Asparagus: Automated Synthesis of Parametric Gas Upper-bounds for Smart Contracts

Zhuo Cai, Soroush Farokhnia, Amir Goharshady, S Hitarth

Asparagus: Automated Synthesis of Parametric Gas Upper-bounds for Smart Contracts

ZHUO CAI, SOROUSH FAROKHNIA, AMIR GOHARSHADY, and S. HITARTH,
Hong Kong University of Science and Technology, Hong Kong

Modern programmable blockchains have built-in support for smart contracts, i.e. programs that are stored on the blockchain and whose state is subject to consensus. After a smart contract is deployed on the blockchain, anyone on the network can interact with it and call its functions by creating transactions. The blockchain protocol is then used to reach a consensus about the order of the transactions and, as a direct corollary, the state of every smart contract. Reaching such consensus necessarily requires every node on the network to execute all function calls. Thus, an attacker can perform DoS by creating expensive transactions and function calls that use considerable or even possibly infinite time and space. To avoid this, following Ethereum, virtually all programmable blockchains have introduced the concept of "gas". A fixed hard-coded gas cost is assigned to every atomic operation and the user who calls a function has to pay for its total gas usage. This technique ensures that the protocol is not vulnerable to DoS attacks, but it has also had significant unintended consequences. Out-of-gas errors, i.e. when a user misunderestimates the gas usage of their function call and does not allocate enough gas, are a major source of security vulnerabilities in Ethereum.

We focus on the well-studied problem of automatically finding upper-bounds on the gas usage of a smart contract. This is a classical problem in the blockchain community and has also been extensively studied by researchers in programming languages and verification. In this work, we provide a novel approach using theorems from polyhedral geometry and real algebraic geometry, namely Farkas' Lemma, Handelman's Theorem, and Putinar's Positivstellensatz, to automatically synthesize linear and polynomial parametric bounds for the gas usage of smart contracts. Our approach is the first to provide completeness guarantees for the synthesis of such parametric upper-bounds. Moreover, our theoretical results are independent of the underlying consensus protocol and can be applied to smart contracts written in any language and run on any blockchain.

As a proof of concept, we also provide a tool, called "Asparagus" that implements our algorithms for Ethereum contracts written in Solidity. Finally, we provide extensive experimental results over 24,188 real-world smart contracts that are currently deployed on the Ethereum blockchain. We compare Asparagus against GASTAP, which is the only previous tool that could provide parametric bounds, and show that our method significantly outperforms it, both in terms of applicability and the tightness of the resulting bounds. More specifically, our approach can handle 80.56% of the functions (126,269 out of 156,735) in comparison with GASTAP's 58.62%. Additionally, even on the benchmarks where both approaches successfully synthesize a bound, our bound is tighter in 97.85% of the cases.

CCS Concepts: • **Theory of computation → Program analysis**; **Program verification**; • **Mathematics of computing → Quadratic programming**.

Additional Key Words and Phrases: Smart Contracts, Blockchain, Gas Bounds, Out-of-gas Vulnerabilities

Authors' address: Zhuo Cai, zcaiam@connect.ust.hk; Soroush Farokhnia, sfarokhnia@connect.ust.hk; Amir Goharshady, goharshady@cse.ust.hk; S. Hitarth, hsinghab@connect.ust.hk, Departments of Computer Science and Mathematics, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong.

2475-1421/2023/10-ART253
https://doi.org/10.1145/3622829

ACM Reference Format:
Zhuo Cai, Soroush Farokhnia, Amir Goharshady, and S. Hitarth. 2023. Asparagus: Automated Synthesis of Parametric Gas Upper-bounds for Smart Contracts. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 253 (October 2023), 34 pages. https://doi.org/10.1145/3622829

1 INTRODUCTION AND MOTIVATION

Smart Contracts [Wood et al. 2014]. Smart contracts are programs that are stored and executed on the blockchain. They are immutable after deployment and sovereign, meaning that they can receive and hold funds in the form of a cryptocurrency and the funds can only be retrieved if allowed by the smart contract's code. When a user interacts with a smart contract by calling a function, all nodes on the blockchain network have to run the function call. Thus, all nodes reach a consensus about the state of every smart contract and the payments going in or out of it.

Gas. Allowing arbitrary smart contracts in a Turing-complete language opens the door to many potential vulnerabilities and attacks [Atzei et al. 2017]. Specifically, an attacker can deploy a smart contract that includes a non-terminating loop or one that is otherwise too time-consuming or memory-consuming to execute. They can then create a function call transaction that invokes such costly execution. Then, every node on the network has to execute the transaction in order to guarantee consensus about the final state of the contract and balances of the parties. So, a DoS attack would be catastrophic and can halt the operation of the entire cryptocurrency. To avoid this, Ethereum introduced the concept of *gas* [Wood et al. 2014]. A fixed gas cost is assigned to every atomic operation available in the smart contract programming language. The costs of atomic operations are hard-coded as part of the blockchain protocol. The initiator of a transaction has to pay for its total gas usage over its entire execution. This payment is then transferred to the miner.

Deposits and Out-of-gas Behavior. The user who initiates a transaction (function call to a smart contract) can specify the maximum amount of gas that the transaction is allowed to spend, as well as the price that the user is willing to pay per unit of gas[1]. They should then provide a deposit equal to the product of the two parameters. When the transaction is added to the blockchain, every node in the network executes it, while keeping track of the amount of gas that is being used. If the function call requires more than the allocated gas, then not only the deposit is confiscated but the call itself is reverted and canceled, i.e. all effects of this function call, be it on user's balances or the variables of the smart contracts, will be reverted as if this function call never existed. The only effect is that the initiator loses their deposit.

Our Focus. In this work, we focus on the problem of automated synthesis of parametric bounds on the gas usage of a smart contract. This problem is well-studied in the literature. There are three main motivations for automated bound synthesis: (i) avoiding a loss of deposit due to insufficient gas allocation, (ii) ensuring that a contract can safely call libraries and functions in other contracts, and (iii) avoiding out-of-gas vulnerabilities, where wrong estimations of the gas behavior of a smart contract can lead to catastrophic real-world consequences.

Motivation (i). Avoiding Lost Deposits. In the simplest case, if a user mistakenly underestimates the amount of gas that is needed for the execution of their desired function call and chooses a maximum gas value that is too small, then they would lose their entire deposit when their transaction is mined. This happens even if the estimate was just one unit short of the required gas. Thus, a cautious user should first use static analysis tools to obtain a guaranteed upper-bound on the gas usage of the function they wish to call and then choose the gas limit accordingly.

[1] After the London upgrade in Ethereum, there is a minimum base price that the user must pay or else the transaction is invalid. However, this has no effect on our algorithms, since they all consider gas usage itself and are independent of the gas price.

While losing a gas deposit is irritating, out-of-gas errors can lead to much more severe losses, as further described below. Moreover, the decision on the allocated gas is not always taken by the final user, but often by the programmer of a smart contract.

Motivation (ii). Safe Gas Sharing [Wood et al. 2014]. Bounding the gas usage of a function is further complicated by gas sharing: functions can recursively call other functions, both in the same contract and in other contracts. In these calls, the caller can choose how much of its remaining gas is allocated to the callee. Moreover, it can decide how to handle an out-of-gas exception in the callee, i.e. whether to continue its own execution after the callee failed and was reverted or revert its own execution, too, and throw an exception. Ideally, the caller should provide enough gas to the callee to ensure it can be executed successfully. Otherwise, the smart contract will not have the desired functionality and might even become vulnerable to attacks. Unexpected out-of-gas errors that are propagated can effectively disable the functionality of the smart contract, making it impossible for anyone to successfully call some of the functions [Atzei et al. 2017; Luu et al. 2016]. Thus, it is the caller programmer's responsibility to ensure that, under any circumstance, enough gas is allocated to the callee. Since a smart contract is immutable after deployment, the programmer cannot update the caller's code. Thus, they must first perform a static analysis on the callee's code and obtain a parametric bound on the gas usage of the callee. Then, this bound should be hard-coded in the caller's code, ensuring that every call allocates enough gas to be successfully executed to completion.

Motivation (iii). Identifying Out-of-gas Vulnerabilities. The out-of-gas exceptions can even cause vulnerabilities or be used in intentional attacks [Atzei et al. 2017]. In real-world Ethereum contracts, among all possible kinds of vulnerabilities, out-of-gas exceptions and their mishandling stand out both in terms of frequency of occurrence and the amount of financial damage caused [Liu et al. 2019]. To further motivate our problem, we now provide examples of such attacks and how parametric gas bounds can be used to detect them. We remark that our approach, and gas bounds in general, are not able to fix these vulnerabilities. However, they can indicate the existence of unexpected or undesirable behavior and point to the exact function in the smart contract where it occurs. Thus, a programmer who performs static analysis before deployment of the smart contract would be able to find and fix such issues. On the other hand, a user who intends to interact with a contract can also first run a static analysis to find a gas upper-bound and check if the contract shows suspicious behavior in terms of gas.

Example 1.1 (Gasless Send [Atzei et al. 2017]). As our first motivating example, we consider a famous real-world contract called "King of the Ether throne". This is a dishonest contract that deceives its users and steals their money [Atzei et al. 2017]. It claims to be a simple game similar to a Ponzi scheme. Anyone can join as a player and the players compete for the title of king. Note that a player can itself be a smart contract. To become king, one has to pay an amount to the contract that exceeds the payment of the previous king. This will then unseat the previous king, refund their deposit, and crown the new king. It also pays a small commission to the contract's developer. Thus, the deposit amount required to become king increases monotonically. In practice, the contract was used by individuals to boast about their wealth and also by other developers to advertise their own smart contracts since a sure way of attracting attention to a smart contract was to make it king.

Let's consider a shortened version of this contract's code (Figure 1, left) which has the same vulnerabilities as in the original contract. In this code, the function that has the same name as the contract is a constructor, i.e. `KotET` is called only when the contract is deployed for the first time. It simply sets the king and the owner to be the person who is deploying the contract. When someone sends money to a contract without specifying a function name, a default *fallback* function is called. In this case, this is the nameless function. Thus, players can take part in the game by

```
contract KotET {
 address payable king;
 address payable owner;
 uint claimPrice;

 function KotET() {
  king = owner = msg.sender;
  claimPrice = 100;
 }

 function () payable {
  require(msg.value >= claimPrice);
  uint compensation = ...;
  king.send(compensation); //(*)
  king = msg.sender;
  claimPrice = ...;
 }

 function sweepCommission() {
  owner.send(address(this).balance);
 }
}
```

```
(bool sent, bytes memory data) =
king.call{value: compensation}("");

require(sent);
```

```
contract MaliciousKing {

 KotET victim = ... ; //address of
     the victim on the blockchain

 function attack() public payable {
  victim.call{value: msg.value}("");
 }

 function () payable {
  while(1>0)
  {
   //loop forever
  }
 }

}
```

Fig. 1. A dishonest KotET contract (left), the required changes to make it honest (top right), and a malicious contract attacking the honest version (bottom right).

simply sending money to the contract, which automatically triggers this function. The fallback function first checks to see if the paid amount is enough for claiming the title of king. If not, it simply throws an exception, which has the effect of reverting the function call and refunding the caller. Otherwise, it calculates a compensation for the previous king and pays it out. It then crowns the caller as the new king and also calculates the price another player has to pay to unseat the king. The difference between the amount paid by the new king and the compensation sent to the old king is kept by the contract as a commission. The owner can always call `sweepCommission` and receive the entire balance of the contract.

At first glance, the contract seems honest. However, the devil is in the details. If the king is itself another contract, then the send operation marked by `(*)` calls the king's own fallback function. However, the default behavior of send is to share only a tiny amount of gas. Thus, the send at `(*)` might fail due to an out-of-gas exception. Moreover, this exception is not caught and thus the KotET contract's execution simply continues and the compensation remains in KotET's balance, which can later be claimed by the owner using `sweepCommission`. This attack is rather unintuitive and hard to detect for a human. So, the developer of another smart contract can be excused for not identifying it when they try to make their contract king.

However, if the developer of the other contract performs an automated gas analysis, such as the one provided by this work, it can flag and avoid the attack easily, since it detects an out-of-gas exception at `(*)`. Specifically, in an honest contract, the gas usage of this function should be dependent on the gas usage of the king's fallback function. However, in this case, an approach such as ours reports that the nameless fallback function in KotET has constant gas usage. Moreover, KotET's fallback function, as the caller, uses less gas than the king's fallback function, which is the callee. These two facts both point to an intentional and unavoidable out-of-gas error, which strongly suggests the possibility of an attack.

Let's now consider an honest implementation of KotET, in which the send at `(*)` is modified so that the entire available gas is shared with the callee and exceptions are also propagated, i.e. if the

king's fallback function runs out of gas and the reimbursement of the king fails, then the entire transaction would also be reverted. This can be achieved by replacing the `(*)` line with the code in Figure 1 (top right). Unlike `send`, the default behavior in `call` is to share the entire available gas. So, the code above sends the compensation to the king and also requires that the transfer is successful. If not, it simply throws an exception.

Unfortunately, even this implementation would be vulnerable to gas-based attacks, this time by the king. Indeed, one can create an attack contract as shown in Figure 1 (bottom right). When `attack` is called, `MaliciousKing` takes part in `KotET` by making a deposit and becoming king. However, it will remain king forever and can never be unseated. Whenever a different player pays a higher price to become king, `KotET` calls `MaliciousKing`'s fallback function. This is a trap that executes a non-terminating loop which is guaranteed to run out of gas. So, reimbursing `MaliciousKing` fails and `KotET` reverts the transaction. Hence, `MaliciousKing` always remains king. Again, a parametric gas analysis by `KotET`'s developer would find a linear gas bound of the form $x + c$ for `KotET`'s fallback function, in which x is the gas usage of the king's fallback function. This would immediately indicate a vulnerability to the developer, since the gas usage is dependent on an external parameter x, which is not under the developer's control. This means an external party can cause an out-of-gas exception.

```
contract Bank {
 uint maxAccounts = ...;
 Account accounts[];

 function createAccount(uint r) public
     payable {
  require(accounts.length < maxAccounts);
  Account newAccount;
  newAccount.owner = msg.sender;
  newAccount._balance = msg.value;
  newAccount.referrer = r;
  accounts.push(newAccount);
 }

 function addRewards() public {
  for(uint i=0;i<accounts.length;i++)
   for(uint j=i+1;j<accounts.length;j++)
    if(accounts[j].referrer==i) {
     accounts[i]._balance += 100;
     accounts[j].referrer = j;
    }}}
```

Fig. 2. A Bank contract

Pervasiveness. Similar vulnerabilities, which are often called "gasless send", are extremely common in real-world Ethereum smart contracts. Based on an empirical study in [Prechtel et al. 2019], more than half of the contracts deployed on Ethereum have a variant of this vulnerability. According to [Grech et al. 2018], contracts with gas-based vulnerabilities held more than 2.8 Billion USD in 2018.

Example 1.2 (Disabled Functionality, Variant of [Grech et al. 2018]). Consider the smart contract fragment in Figure 2 that implements a bank and provides incentives to its users to refer their friends to open accounts. It has a `createAccount` function, which creates a new account while keeping track of the referrer. It also has an `addRewards` function which computes the rewards of all referrers and updates their balances. Note that this is a gas-inefficient implementation, since `addRewards` has a quadratic runtime. While gas-hungry implementations are highly discouraged, they nevertheless remain common in real-world smart contracts. However, the security problem here is even subtler. In Ethereum, each block, and thus each transaction, is limited to 30 million units of gas. Hence, if the maximum number of accounts is large enough, the `addRewards` function will always run out of gas and get reverted. Effectively, this is equivalent to disabling this function since no one can call it successfully. Of course, the bank can claim that `maxAccounts` is set in a way that ensures `addRewards` never becomes disabled. However, to verify such a claim, a user needs to have a tight upper-bound on the gas usage of `addRewards`. Such an upper-bound should be obtained by a static analysis before the user decides to deposit funds into this smart contract. The upper-bound is necessarily parametric, depending on the length of the array `accounts`. It is also quadratic.

Attack Identification. Parametric gas usage bounds can be used to unmask the attacks mentioned above. In each case, tight gas bounds would show that the contract has the potential to use much more gas than expected or that its gas usage is dependent on an external contract and not bounded or that it uses less gas than one of the functions it calls. These are all cases of highly suspicious and unexpected behavior, which would lead the user to suspect an attack since an honest contract in these cases is expected to have a constant/linearly-bounded gas usage that is independent of all external contracts. Our approach provides a witness (gas bound) showing that the contracts in question are not honest, but it cannot describe the attacks. However, it can locate them, pointing to the exact function in code which has an unexpected gas usage bound. In practice, a cautious user who applies our tool would simply refuse to interact with such contracts.

Problem Formulation. In this work, we focus on the problem of automated synthesis of parametric upper-bounds for a given smart contract $\mathscr{C}$. Our goal is to find a parametric expression B_f for every function f of $\mathscr{C}$, such that B_f, which can potentially depend on the parameters passed to f, the variables stored in $\mathscr{C}$, and the gas usage of other functions called by f, serves as a guaranteed upper-bound on the gas usage of any call to f. Of course, we would like each B_f to be as tight as possible.

Related Works. Due to the importance of avoiding out-of-gas vulnerabilities, this is a well-studied problem and there are several previous tools that attempt to solve it. See Section 6 for a more in-depth comparison. Notably, `solc`, the standard compiler of Solidity, which is the most commonly used programming language for Ethereum smart contracts, already includes a static analyzer that tries to find such upper-bounds and issues a warning to the programmer if it cannot prove that a function has small gas usage. However, these tools are only able to find constant bounds on the gas usage and would simply report $+\infty$ as their bound if no such constant can be found. To the best of our knowledge, the only exception is GASTAP [Albert et al. 2021], which was the only previous tool to provide parametric bounds. We provide an extensive experimental comparison with GASTAP in Section 5.

Undecidability and Limitations. The general case of our problem, where $\mathscr{C}$ can be any contract written in a Turing-complete language, is clearly undecidable since B_f is finite if and only if f terminates and thus halting is a special case of our problem. Therefore, we apply some abstractions and allow only numerical variables in the smart contracts. Specifically, we abstract away the arrays and instead just keep track of their size. These abstractions are sound, but not necessarily complete. Moreover, we focus on synthesizing polynomial upper-bounds B_f. See Section 2 below for more formal treatment. Thus, our approach can only find polynomial bounds over polynomial transition systems. This excludes non-polynomial smart contracts. It also excludes contracts written in functional programming languages such as Scilla [Sergey et al. 2019], which may contain higher-order functions.

Our Contribution. In this work, we consider the problem of finding polynomial gas usage upper-bounds for every function of a given smart contract $\mathscr{C}$. We model our smart contracts as polynomial transition systems (PTSs) and present the following contributions:

- *Theoretical Algorithms.* We provide novel and entirely automated algorithms, using techniques from polyhedral and real algebraic geometry, to synthesize the tightest possible polynomial gas upper-bounds for a given PTS. We show that our approach is not only sound, but also semi-complete, i.e. complete as long as a large enough degree is used for the polynomials. Our theoretical approach is language-independent and can be applied to smart contracts written in any language and run on top of any blockchain protocol.

- *Practical Implementation.* We present a highly-optimized implementation of our algorithm on Ethereum smart contracts which are written in Solidity. Our tool is called *Asparagus*[2] and is open-source, freely available and dedicated to the public domain[3].
- *Experimental Results.* We provide extensive experimental results on 156,735 functions from 24,188 real-world smart contracts, showing that our approach is scalable and applicable to the vast majority of real-world Ethereum contracts (80.56% percent). Moreover, we compare our results with GASTAP, the only previous method that could synthesize parametric bounds. Our experiments clearly demonstrate that Asparagus outperforms GASTAP both in terms of the number of contracts it can handle and the quality and tightness of the bounds. See Section 5 for details.

Novelty. Our approach is novel in a number of ways:

- To the best of our knowledge, this is the first use-case of algebro-geometric theorems and algorithms in a blockchain context.
- Our approach is one of only two to synthesize parametric gas upper-bounds to date. It also significantly outperforms the previous approach (GASTAP) in practice.
- Our algorithms are the first to provide completeness guarantees, showing that we can find the tightest possible polynomial bounds.

Motivation for Polynomial Bounds. In our experimental result, 653 (2.7%) of the benchmarks required quadratic polynomial bounds. These are usually contracts that compute hashes or allocate memory within loops. Thus, no constant or linear bounds exist in these cases and non-linear bounds are required. Polynomial bounds are also desirable for the following additional reasons:

- Certain EVM operations, such as `MSTORE`, incur memory expansion costs which are *quadratic* in size of the used memory. To handle contracts that use a non-constant amount of memory, one has to go beyond linear bounds and introduce polynomials.
- At the time of writing, the gas price is too high, leading many smart contract programmers to avoid loops altogether. This is a major issue that is artificially keeping the contracts simple and not allowing them to exploit the promise of a Turing-complete language. Indeed, the vast majority of functions our experiments had constant gas usage. We expect that with the recent switch to proof-of-stake and proposals for relaxations of the block gas limit, the gas price would decrease significantly, allowing smart contract programmers to write more complicated functionality. Thus, our expectation is to see more contracts with non-linear gas usage in the future.
- Our approach outperforms both `solc` and GASTAP even in the cases when the gas bound is constant. Additionally, our bounds are tighter even when both approaches synthesize constant or linear bounds. These gains are due to the extra expressivity of our polynomial templates.

2 PRELIMINARIES

In this section, we first cover the basics of blockchains and then provide an overview of Ethereum and its gas model. We then present an intermediate representation of an Ethereum smart contract, which was introduced in [Albert et al. 2018]. Finally, we present the concept of a polynomial transition system (PTS). Our algorithm in the next section works with a PTS as its input. Thus, it is applicable to any smart contract, independent of the original language, as long as there is a tool for translating it to PTS. Our tool, Asparagus, gets an Ethereum smart contract written in Solidity as

[2] Automated synthesis of parametric gas upper-bounds for smart contracts

[3] Previous tools for this problem are closed-source and proprietary. GASTAP is only accessible through a web interface and we could not even obtain an executable.

its input, translates it to the intermediate representation of [Albert et al. 2018], then converts the intermediate representation to a PTS, and finally applies our algorithm on the PTS.

2.1 Blockchain Background

This section provides basic background on blockchains and their basic protocols. It can be safely skipped by a reader who is already familiar with this topic. A more detailed discussion of the topics in this section can be found in [Nakamoto 2008; Priyadarshini 2019; Singhal et al. 2018].

Bitcoin and Transactions. Bitcoin was the first electronic currency to provide a trustless and decentralized protocol ensuring everyone on the network could reach a consensus regarding ownership of the funds [Nakamoto 2008]. On the Bitcoin network, the users are pseudonymous and identified by their public keys. The primary objects in Bitcoin are transactions. A user who owns some funds can create a transaction that transfers them to others. Such a transaction should contain a pointer to the previous transactions in which the user obtained the funds (proof of ownership), a signature from the user (proof of consent), and the public keys of recipients. Of course, the transactions cannot create money out of thin air, and hence the total amount of Bitcoin flowing into a transaction should always be at least the amount flowing out of it. The users simply announce their transactions to the network and valid transactions are propagated by a gossip protocol.

Double-spending. Propagating transactions in a peer-to-peer network is not enough for a functioning cryptocurrency. Suppose that Alice owns 1 BTC. She can do *double spending*, i.e. she can create two different transactions Tx_1 and Tx_2. In Tx_1, she transfers the 1 BTC to Bob and in Tx_2 to Carol. She then publishes both transactions at the same time by announcing them to different, and potentially very distant, nodes of the Bitcoin network. Consider a node who hears about Tx_1 first. This node would consider Tx_1 as valid, because Alice owned the 1 BTC she spent, but then when it hears about Tx_2, ignores it as an invalid transaction since the money was already spent in Tx_1 and now belongs to Bob. Conversely, a node who hears about Tx_2 first would believe that Tx_1 is invalid. Hence, after a while, there is a disagreement among the nodes, with one group believing that the 1 BTC belongs to Bob and the opposing group believing it belongs to Carol.

Consensus and Blockchain. To avoid the situation above, it is imperative that all nodes on the network can reach a consensus about not only the set of valid transactions, but also their order. In Bitcoin, the transactions can be grouped into *blocks* of a fixed size. Each block is basically an ordered sequence of transactions, together with some additional metadata. Moreover, every node keeps its own copy of the so-called *blockchain*, which is an append-only linked list of valid blocks. The goal is to ensure that all nodes have the same blockchain. This is achieved by a proof-of-work protocol [Nakamoto 2008], in which adding a block to the blockchain is conditioned upon successfully solving an extremely difficult computational puzzle, i.e. partially inverting a hash function. Some of the nodes, who are called *miners*, compete in creating blocks and solving the computational puzzle, so that they can extend the blockchain[4]. The basic idea is that it is unlikely to have two miners who find a valid solution to the hash inversion problem at approximately the same time and hence the blockchain grows steadily and all nodes agree on its contents. If two or more valid solutions to the proof-of-work puzzle are found at approximately the same time, then there will be a temporary disagreement among the nodes about the state of the blockchain. This is called a *fork*. However, the Bitcoin protocol asserts that the longest chain is the consensus chain. After a while, one of the chains will eventually become longer than others, and then all other chains are dropped and a consensus is restored. It is noteworthy that proof-of-work is not the only viable

[4]The miners are incentivized to do so, because adding a new block pays them a reward. This is also how new units of currency are created. Moreover, each transaction can include a *transaction fee*, which is paid to the miner who successfully adds it to the blockchain.

consensus mechanism and several alternatives exist, including proof-of-stake [Gilad et al. 2017; Kiayias et al. 2017] and proof-of-space [Dziembowski et al. 2015; Park et al. 2018]. In this work, we treat the blockchain and its underlying consensus protocol as black boxes and our results are in no way dependent on how the consensus is reached.

Transaction Scripts and Smart Contracts. As the first example of a working cryptocurrency, Bitcoin [Nakamoto 2008] uses a blockchain protocol to reach a network-wide consensus about the history of transactions. Since consensus is independent of data type of transactions, we can have richer notions of transactions that go beyond simply transferring money. For example, Bitcoin allows *scripts* that establish complicated pre-conditions for spending a particular coin (transaction output), such as requiring signatures from three out of four owners. Ethereum [Wood et al. 2014] took this idea to the extreme and allow arbitrary programs written in a Turing-complete language, which are called *Smart contracts*. Most other modern cryptocurrencies have followed Ethereum and enabled smart contracts. More specifically, a transaction in Ethereum can not only transfer money as in Bitcoin, but also perform other tasks:

- *Deployment of a Contract.* A transaction can contain the code of a program, i.e. a smart contract, and hence deploy it on the blockchain. Simply put, when this transaction is included in a block and then added to the blockchain, every node in the network has access to the smart contract's code, which is now part of the consensus. Moreover, the code is immutable since the blockchain is append-only.
- *Function Calls.* A transaction can "call" a desired function in a smart contract that was deployed by a previous transaction. To do so, the transaction has to provide a pointer to the smart contract, the name of the called function, and the parameters passed to it. A transaction record containing all this data will be added to the blockchain.

Every node on the network keeps track of the smart contracts and their state, i.e. values of the variables in their storage. Moreover, a smart contract is sovereign in the sense that it can receive and hold money in the form of the underlying cryptocurrency. Any funds transferred to a smart contract can only be reclaimed if the smart contract transfers it back, i.e. if a function of the smart contract is called which leads to a transfer command. As mentioned above, contracts are immutable after deployment. The underlying consensus protocol ensures that all nodes on the network reach a consensus about the blockchain, which is, by definition, a consensus about the transactions and their order. So, assuming that the smart contracts are written in an unambiguous language with well-defined semantics, every node can simply execute the transactions in the order that they appear in the blockchain and thus we reach a consensus about the state and balance of every account. In this sense, smart contracts are enforced by the blockchain protocol.

2.2 Ethereum and its Gas Model

This section provides a quick introduction to Ethereum and its gas model, following [Dameron 2019; Wood et al. 2014].

Accounts. In Ethereum, the currency unit is called an "ether" and uses the acronym ETH. An account is an entity with an ether (ETH) balance that can transfer money on Ethereum. It is either controlled by a user or a deployed smart contract.

Transactions. A user can publish an Ethereum transaction to transfer money, deploy a new contract and change the blockchain state.

EVM. The Ethereum Virtual Machine (EVM) deterministically and unambiguously describes the rules of executing transactions on Ethereum. EVM supports a set of operations (opcodes) to interact with input call data, stack, memory, storage and the blockchain world state.

Smart Contracts. A deployed smart contract has the following data fields: *nonce*, *balance*, *codeHash* and *storageRoot*. The code of the smart contract is stored on the blockchain as bytecode (a sequence of opcodes) and immutable. All participants of Ethereum can interact with smart contracts by calling the public functions in their code. The EVM bytecode format is Turing-complete.

Solidity [Ethereum Foundation 2014]. Solidity is an object-oriented, high-level Turing-complete language for implementing Ethereum smart contracts and the most widely-used smart contract language. Solidity supports all the standard data types and data structures such as arrays and mappings. The functions in Solidity contracts can call other functions from the same contract or external public functions by specifying the function signatures and contract addresses.

Gas. Ethereum introduced a fee schedule to avoid the issue of DoS network attacks caused by Turing-complete smart contract computation. The fee schedule is specified in units of *gas*. Each computation in the EVM is associated with a cost in terms of units of gas. Therefore, an executed transaction, which translates to a sequence of computation steps in the EVM, consumes a well-defined amount of gas. The transaction owner should set the *gasLimit* field and purchase the gas at the price of $gasLimit \cdot gasPrice$. The specified *gasLimit* must be no smaller than the actual amount of gas cost in execution, otherwise the gas runs out before the computation terminates. On the other hand, it is also undesirable to set an unnecessarily large *gasLimit*, since miners are reluctant to add them to the blockchain because each block has a total gas limit. On the other hand, users are also reluctant to put down huge deposits.

Gas Usage Formula. The Ethereum protocol fixes a specific gas cost for every one of the EVM opcodes using a complicated formula to address the different computational workloads of different opcodes. See [Dameron 2019, Page 14] for a complete table of gas costs. We follow these costs in our implementation but their specific values have no bearing on the theory. In summary:

- Most opcodes have fixed constant gas costs. For example, stack operation opcodes `PUSHx`, `POPx` and `SWAPx` have a gas cost of 3 units each.
- Some opcodes have a few different gas costs given different conditions. The `SSTORE` opcode is an example, which saves a word to storage. The gas cost is 20,000 if it is overwriting a zero with a non-zero value, and 5,000 otherwise. Moreover, if the operation clears a non-zero to a zero, then 15,000 units of gas are refunded, but this refund is made only if the current transaction uses more than 15,000 units of gas. Since these conditions are unnecessarily complicated and depend on the dynamic worldstate, and since they are not specified by the smart contract itself, we always replace them with their highest possible value to preserve soundness, e.g. we always assume that an `SSTORE` takes 20,000 units of gas.
- The gas costs of some operations depend on the size of a particular stack or array. For example, taking the hash of a piece of data that is n bytes long has a dynamic cost that is linearly dependent on n. To handle these cases, we always keep track of the sizes of the stacks/arrays as separate variables. Notably, while these costs are not constants, they are always polynomials (but not necessarily linear) with respect to their input size n.

2.3 Intermediate Representations and Polynomial Transition Systems

GASTAP [Albert et al. 2021]. GASTAP is a tool to process EVM bytecode into an intermediate rule-based representation (RBR) and estimate gas bounds of function calls. The part of the GASTAP project which obtains the RBR is called EthIR and is publicly available [Albert et al. 2018]. Our algorithm works on a polynomial transition system (PTS, defined further below). However, the translation from RBR to PTS is much simpler than a direct translation from EVM bytecode. So, in practice, we first use [Albert et al. 2018] to translate EVM to RBR and then convert the RBR to PTS.

```
contract VotingContract {

 struct Proposal {bytes32 name;  uint voteCount;}
 Proposal[] public proposals;

 function winningProposal() public returns (uint
     winningProposal){
  uint winningVoteCount = 0;
  for (uint p=0;p<proposals.length;p++){
   if (proposals[p].voteCount>winningVoteCount) {
    winningVoteCount = proposals[p].voteCount;
    winningProposal = p;
   }}}}
```

```
contract NestedLoop{

function main(uint a,uint b)
     public returns(uint)
 {
  uint count = 0;
  for (uint i=0;i<a;++i)
   for (uint j=0;j<b;++j)
    ++count;
  return count;
 }
}
```

Fig. 3. Two Example Contracts

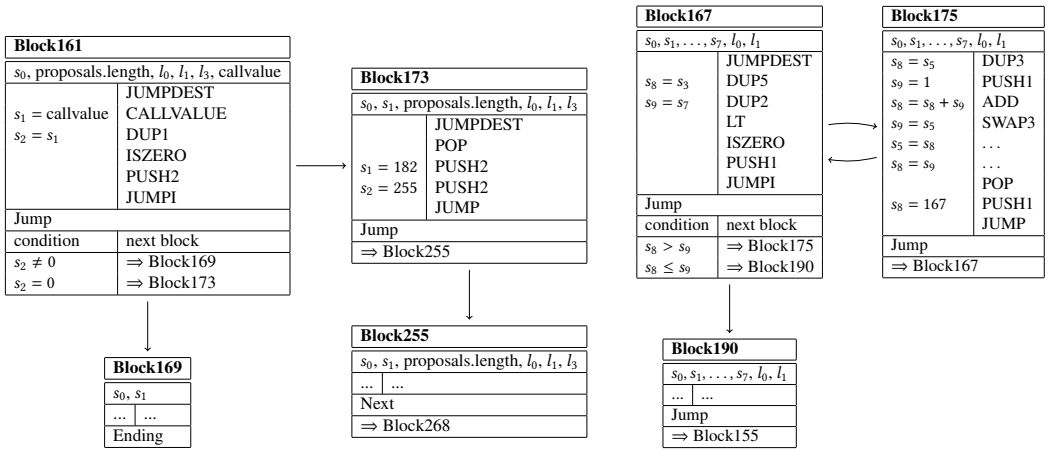

Fig. 4. Part of the Rule-based Representations (RBR) of *VotingContract* (left) and *NestedLoop* (right)

EVM to CFG. GASTAP firstly converts an EVM bytecode to a control flow graph (CFG). The EVM language uses a local stack, a volatile memory, and a persistent storage that is part of the blockchain state. Given the compiled smart contract, GASTAP groups the code into EVM basic blocks, which are maximal sequences of straight-line code. Each EVM block ends with either a `JUMP`/`JUMPI` operation where program execution jumps to a specified location, or an ending instruction like `RETURN` or `REVERT`. Each EVM block has an address according to the location where it appears in the bytecode. When a block jumps, static analysis is applied to parse the jump destination and form an edge to the destination block. As a result, a stack-sensitive control flow graph is created. This CFG is then used to translate the program into RBR and PTS formats below. See [Albert et al. 2018] for detailed syntax and semantics of RBR.

Example 2.1. Figure 3 shows two Ethereum smart contracts, written in Solidity, which will serve as our running examples. Figure 4 shows part of the RBR representation of these contracts, as well as the edges in their CFGs.

Polynomial Update Functions, Assertions and Transitions. Let $\mathbb{V} = \{v_1, \ldots, v_k\}$ denote a finite set of variables. We denote the set of all polynomial expressions with real coefficients over

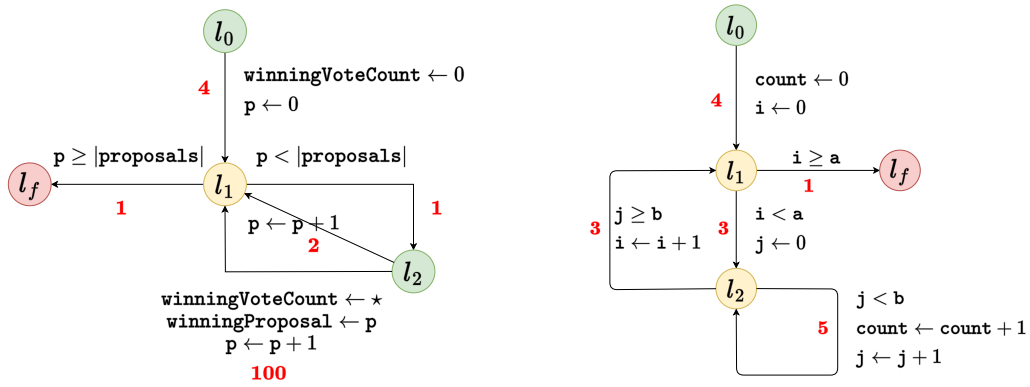

Fig. 5. Simplified PTS Representations T_1 (left) and T_2 (right) of Running Examples. The numbers in red denote the cost of each transition. In these examples, the costs are constant, but our approach supports arbitrary polynomial costs. Cutpoints are shown in yellow.

the variables $\mathbb{V}$ by $\mathbb{R}[\mathbb{V}]$. A polynomial *update function* $U : \mathbb{V} \to \mathbb{R}[\mathbb{V}]$ maps each variable in $\mathbb{V}$ to a polynomial over $\mathbb{V}$. A *polynomial assertion* G is a logical formula formed by a boolean combination of the assertions of the form $f \geq 0$ and $f > 0$, where $f \in \mathbb{R}[\mathbb{V}]$. A *polynomial transition* $\tau = (U, G)$ is a pair of an update function and a guard. With a slight misuse of notation, we sometimes write τ_U instead of U and τ_G to denote G.

Valuations and Notation. A *valuation* $\sigma : \mathbb{V} \to \mathbb{R}$ assigns a real value to each variable in $\mathbb{V}$. A polynomial f can be evaluated at any valuation σ naturally by substituting the value of each variable in f and computing it. We will denote this value by $f(\sigma)$. Let U be an update function, then $U(\sigma)$ denotes a valuation in which $U(\sigma)(v) = U(v)(\sigma)$ for each v. If F is a polynomial or polynomial assertion, we define $U(F)$ as the result of substituting each occurrence of a variable v in F by the polynomial $U(v)$.

Composition of Transitions. Given two polynomial transitions $\tau_1 = (U_1, G_1)$ and $\tau_2 = (U_2, G_2)$, we denote their composition by $\tau_1 \circ \tau_2$ and define it as (U, G) where $U(v) = U_1(U_2(v))$, and $G = G_1 \wedge U_1(G_2)$.

PTS. A *Polynomial Transition System* (PTS) T is a tuple $(\mathbb{V}, \mathbb{L}, l_0, l_f, \delta)$ where $\mathbb{V}$ is a set of variables, $\mathbb{L}$ is a set of locations, l_0 and l_f are the initial and final locations, respectively, and δ is a partial function that maps each pair of locations (l, l') to a polynomial transition (U, G).

Example 2.2. Figure 5 provides a simplified PTS representation for each of our example contracts of Figure 3. In practice, our tool automatically converts the CFG provided by [Albert et al. 2018] to an RBR. See Section 3 for details. However, the resulting PTSs are usually not human-readable, so Figure 5 is not a direct output of our tool, but a simplified version.

Runs. Given an initial valuation $\sigma : \mathbb{V} \to \mathbb{R}$, a *run* R of the PTS T is an alternating sequence of locations and valuations $R := \sigma_0, l_0, \sigma_1, l_1, \sigma_2, l_2, \sigma_3 \ldots$ satisfying the following constraints:

(1) $\sigma_i = \tau_U(\sigma_{i-1})$ for every $i \geq 1$, where $\tau := \delta(l_{i-1}, l_i)$.
(2) $\sigma_i \models \tau_G$ for every $i \geq 1$. Here, we again have $\tau := \delta(l_{i-1}, l_i)$.

Example 2.3. Consider the initial valuation $\sigma_0 = \{\mathtt{a} \mapsto 0, \mathtt{b} \mapsto 0, \mathtt{count} \mapsto -1, \mathtt{i} \mapsto 10, \mathtt{j} \mapsto 0\}$ in the PTS of Figure 5 (right). An example run starting from this valuation is:

$$\{\mathtt{a} \mapsto 0, \mathtt{b} \mapsto 0, \mathtt{count} \mapsto -1, \mathtt{i} \mapsto 10, \mathtt{j} \mapsto 0\}, l_0$$
$$\{\mathtt{a} \mapsto 0, \mathtt{b} \mapsto 0, \mathtt{count} \mapsto 0, \mathtt{i} \mapsto 0, \mathtt{j} \mapsto 0\}, l_1$$
$$\{\mathtt{a} \mapsto 0, \mathtt{b} \mapsto 0, \mathtt{count} \mapsto 0, \mathtt{i} \mapsto 0, \mathtt{j} \mapsto 0\}, l_f.$$

Cutsets. Given a PTS T, a *cutset* of T is a set of locations C such that (i) C contains l_0 and l_f, and (ii) every cycle in the underlying graph of T passes through some location in the set C. In other words, removing C from T will reduce it to a directed acyclic graph. Each location in C is called a *cutpoint*.

Example 2.4. In Figure 5 (left), the set $\{l_0, l_1, l_f\}$ is a cutset. Similarly, in Figure 5 (right), $\{l_0, l_2, l_f\}$ is a cutset. It is easy to verify that removing these cutsets would eliminate all cycles. Note that the edge labeled **5** is a cycle on its own, thus l_2 has to be included in every cutset.

Basic Paths. Given a cutset C, a path $\Pi := l_1, \tau_1, \ldots, l_{m-1}, \tau_{m-1}, l_m$ in the underlying graph of T is said to be *basic* if it starts and ends at cutpoints and does not pass through any cutpoints in between. It is straightforward to observe that the number of all possible basic paths in a PTS is finite as there could be only a finite number of basic paths between each pair of cutpoints. We extend our transition functions to paths and define $\tau(\Pi) := \bigcirc_{i=0}^{m}(\tau_i)$.

Example 2.5. Using the cutsets from the previous example, the path l_1, **1**, l_2, **2**, l_1 is a basic path of the left PTS and l_2, **5**, l_2 and l_0, **4**, l_1, **3**, l_2 are examples of basic paths in the right PTS.

Invariant Maps. Given a cutset C of the PTS T, we call a map $\mathbb{I}$ that maps each cutpoint to a polynomial assertion an *invariant map* if for any run $R = \sigma_0, l_1, \sigma_1, l_2, \sigma_2, \ldots$ of T that starts from some cutpoint l_1 and initial valuation $\sigma_0 \models \mathbb{I}(l_1)$, we have $\sigma_i \models \mathbb{I}(l_i)$ whenever $l_i \in C$. In other words, if the initial valuation satisfies the invariant at the start cutpoint, then whenever the run R reaches a cutpoint l_i, the valuation of the variables satisfies the invariant at l_i.

Inductive Invariants. An invariant map $\mathbb{I}$ over cutset C is said to be an *inductive invariant map* if for every pair of locations $l, l' \in C$ and every basic path $\Pi := l, \tau_0, l_1 \ldots l_{m-1}, \tau_m, l'$ from l to l' we have $\mathbb{I}(l) \wedge \tau_G \implies \tau_U(\mathbb{I}(l'))$, where $\tau = \bigcirc_{i=0}^{m}(\tau_i)$. In other words, if a valuation satisfies the invariant at the cutpoint l and also satisfies the transition conditions τ_G, then the updated valuation obtained using τ_U satisfies the invariant at the cutpoint l'.

Invariant Generation. Invariant generation and more specifically, the automated synthesis of linear/polynomial inductive invariants, is an orthogonal and well-studied problem with practically efficient tools such as [Chatterjee et al. 2020; Colón et al. 2003]. As such, in the sequel, we assume that every PTS comes with invariants generated using one of these tools.

Abstractions in the Translation from RBR to PTS. We translate a smart contract from the RBR format to a PTS in the standard manner, i.e. creating one location for every line or every basic block of code and following the operations and guards as in the control flow graph. See the next section for an example. However, this process necessarily leads to some imprecision:

- All variables in a PTS are real-valued. Hence, integer variables are converted to real.
- Some operations are inherently not applicable to real variables. We handle these by relying on non-determinism. For example, if we have `c := a%b`. In the PTS, the variable `c` will get a non-deterministic value and the invariant $0 \leq \mathtt{c} \leq \mathtt{b} - 1$ is added. We handle integer division similarly.
- The real variables in a PTS are unbounded, whereas the variable types available in real-world smart contracts are bounded. We add these bounds, e.g. $-2^{31} \leq \mathtt{x} \leq 2^{31} - 1$ for a 32-bit integer `x`, to the invariants.

- There is no support for arrays, stacks or strings in a PTS. Thus, we replace each array with a variable that keeps track of its size. We handle stacks and strings similarly.
- Some contracts have calls to external functions that were not available to us. The results of these calls are also handled by non-determinism. Moreover, for each such external function, we define a new variable that models its total gas cost and use it in our parametric bounds.

Note that the points above do not affect the soundness of our approach and hence all of our obtained bounds are correct. Moreover, the gas cost we assign to every PTS transition is the actual gas cost of the same transition in the original contract and is not at all affected by the translation to PTS. Our experimental results in Section 5 demonstrate the applicability of our approach to real-world smart contracts. Hence, while it is theoretically possible to write adversarial smart contracts to which our approach is not applicable, the vast majority of real-world smart contracts can be modeled as polynomial transition systems.

3 OUR ALGORITHM

In this section, we provide an overview of our approach and illustrate it on the two running examples of Figure 3. To increase readability, we leave out some of the mathematical details and present them separately in the next section. The central idea behind our algorithm is to synthesize a *remaining gas polynomial* at every line of a cutset which models an upper-bound on the possible gas usage if the program starts executing from that line. This is a straightforward extension of the classical concept of ranking functions. We then write a set of entailments between polynomial inequalities which model the requirements of our remaining gas polynomial. Finally, we use tools from polyhedral and real algebraic geometry to translate these entailments into quadratic constraints which are in turn passed to an external solver.

Our algorithm starts with a smart contract and translates it into a set of quadratic constraints using the steps shown in Figure 6. Our central theoretical contribution is the preservation of both soundness and completeness from the PTS all the way to the final solution. In other words, imprecisions can only be introduced in the translations from the smart contract to RBR or from RBR to PTS.

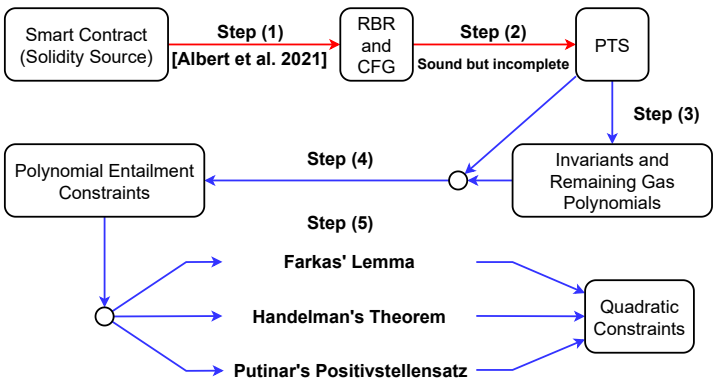

Fig. 6. The Steps of Our Algorithm. Sound and Semi-complete steps are shown in blue, whereas sound steps which might break completeness are in red. See Theorem 4.9 for a more detailed treatment of soundness and completeness.

Our Algorithm. Our algorithm consists of the following six steps:

(1) **Converting the Smart Contract to RBR.** To enable the static analysis of the smart contract, we use the intermediate representation of [Albert et al. 2021]. We compile the smart contract using the standard `solc` compiler to EVM bytecode, then we construct the control-flow graph of the bytecode and convert it to the RBR format, which explicitly represents the local and state variables, the operand stack, and blockchain data. This increases the number of variables that we use in RBR but is essential to facilitate further static analyses. For this step, we rely entirely on the standard compiler and [Albert et al. 2021].

Example. Figure 4 shows parts of the RBRs and CFGs of the two running examples as a graph wherein each vertex is a basic block and each edge is a transition. This is the output of the first step of our algorithm.

(2) **Transforming the RBR to Polynomial Transition System.** We parse the RBR file and create a corresponding polynomial transition system (PTS) $T = (\mathbb{V}, \mathbb{L}, l_0, l_f, \delta)$ as follows:
   (a) **Soundness-preserving Over-approximations.** Since the PTS can only have polynomial operations like addition, subtraction, and multiplication, we must perform a few modifications which preserve the soundness of our approach. As an example, if we encounter a non-polynomial expression such as `x := a%b` in the RBR, we replace it with `x := y` where `y` is a fresh variable with the invariant $0 \leq \mathtt{y} \leq b - 1$. In other words, we are over-approximating non-polynomial operations by non-determinism. The soundness directly follows because our subsequent analysis is then guaranteed to work for all values of `y`, and hence it will also work for the actual value of `x`.
   (b) **Establishing a Gas Consumption Polynomial.** We associate a *gas consumption polynomial* $C(\tau) \in \mathbb{R}[\mathbb{V}]$ to each transition τ in the PTS T. The polynomial $C(\tau)$, when evaluated at a valuation σ, provides the amount of gas that will be consumed on taking the transition τ starting from the valuation σ. In practice, this is a direct encoding of the Ethereum gas table and does not add any imprecision. This is because the gas costs of all EVM opcodes are polynomial expressions. We note that the gas usage assigned to each transition is not necessarily constant and can be any polynomial. Handling polynomial costs is necessary for a faithful modeling of operations such as `SSTORE` which have a quadratic gas usage.
   (c) **Gas Consumption Polynomial of a Path.** Let $\Pi := l_1, \tau_1, l_2, \tau_2, l_3$ be a path in T. The total gas consumed on taking the path Π can be obtained by adding the consumption polynomial of the first transition to the consumption polynomial of the second transition, in which the values of all variables are updated according to the first transition. Formally, we define the consumption $C(\Pi) = C(\tau_1) + \tau_{1U}(C(\tau_2))$. It is straightforward to extend this definition to paths of arbitrary length.

Example. Figure 7 shows part of the transition system obtained for *VotingContract* . This contains not only a renamed version of every variable in the original contract, but also new operations which are not readily apparent in the Solidity code, such as keeping track of the function call stacks and their sizes. As another example, an array in Solidity is compiled to a hash-based map in EVM bytecode. Thus, accessing an element of the array requires a call to a hash function. Since this hash function is not a polynomial, we abstract it away and replace it with non-determinism in our PTS, thus over-approximating the behavior of the original contract. Also note that at this point every transition τ has a well-defined cost $C(\tau)$.
As shown in Figure 7, the polynomial transition system corresponding to a contract can become quite complicated due to the internal implementation details of Solidity. Thus, to make our illustration human-readable, we consider a simplified PTS for each of our contracts, as shown in Figure 5. Note that this is only for illustration and our algorithm works on the

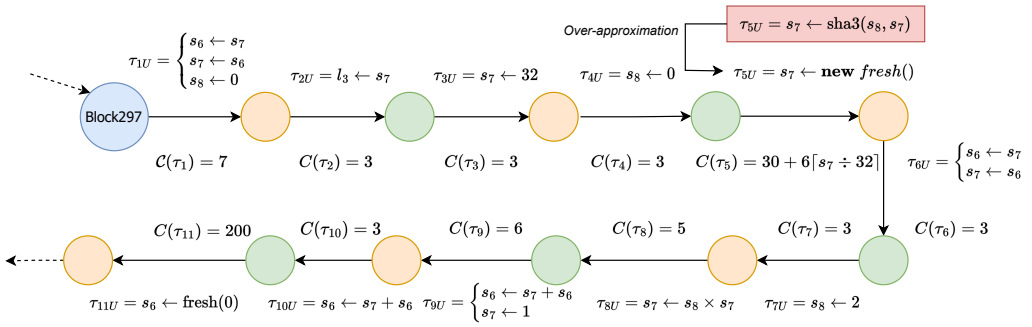

Fig. 7. Part of the PTS for *VotingContract*

original PTS above. In the PTS for *VotingContract* , we have a variable |`proposals`| which keeps track of the length of the array. l_1 corresponds to the `for` loops and l_2 to the `if`. Note that l_2 has two outgoing edges, which are chosen non-deterministically. This is because the branching in the original contract depends on the values saved in the array. However, these values are abstracted away and only the array's length is saved in the PTS. Thus, the transition becomes non-deterministic. Similarly, the assignment to `winningVoteCount` is based on a value in the array and is hence replaced by non-determinism.

(3) **Adding Symbolic Invariants and Remaining Gas Polynomials.** Let C be a cutset of T and $\mathbb{I}$ be an invariant map for C. Note that $\mathbb{I}$ can have both concrete and symbolic polynomials, i.e. polynomial templates with unknown coefficients, as invariants. Our tool can use invariants generated by [Chatterjee et al. 2020; Colón et al. 2003] or generate its own if the given invariants are not strong enough.

(a) **Remaining Gas Polynomial.** We associate a so-called *Remaining Gas Polynomial* B_l to each cutpoint $l \in C$. The idea is that we want to ensure that for every basic path Π starting and ending at the cutpoints l and l', respectively, if we start with an amount of gas that is at least B_l, then the total gas *left* after running through the path is at least $B_{l'}$. The mapping B is said to be a remaining gas polynomial map over cutset C. Note that we do not know the B_l's. Thus, our algorithm simply creates a symbolic template for each of them, i.e. a symbolic polynomial in which the coefficients are unknown real values that have to be synthesized in the future. Our algorithm can create similar templates for the invariants, if they are not provided, and synthesize them similarly[5].
Recall that whenever we reach the location l, the invariant $\mathbb{I}(l)$ must hold. Therefore, we can formalize our requirements on B_l and $B_{l'}$ by taking any basic path Π from l to l' and requiring:

$$\mathbb{I}(l) \wedge \tau_G \implies B_l - C(\Pi) \geq \tau_U(B_{l'}) \tag{1}$$

where $\tau := \tau(\Pi)$ is the composition of all transitions of Π.

(b) **Remaining Gas at the Final Location.** The remaining gas polynomial of the final location l_f is always set to 0. In other words, we ensure that if we reach the final location l_f, no matter what path we took to reach there from the initial location l_0, if we start with gas at least B_{l_0} evaluated at initial valuation, then we will always be left with non-negative gas on termination.

[5]To generate the templates, the algorithm needs to know the maximum degree d that might be used for the polynomials. In practice, we simply try different values of d, starting from 1 and going up, until the algorithm succeeds.

Example. For the first example, we are looking for a linear bound. Thus, at every location $l_i \neq l_f$ of the PTS T_1, our algorithm computes the following remaining gas polynomial:

$$B_{l_i} := t_{i,0} + t_{i,1} \cdot \texttt{winningVoteCount} + t_{i,2} \cdot \texttt{p} + t_{i,3} \cdot |\texttt{proposals}| + t_{i,4} \cdot \texttt{winningProposal}.$$

At l_f we have $B_{l_f} = 0$. Here, the $t_{i,j}$'s are new unknowns, which are called *template variables*. Our goal is to find concrete values for the template variables such that the B_{l_i}'s satisfy the requirements of the remaining gas polynomials as in (1).
For the PTS T_2, our goal is to obtain a quadratic bound, thus our algorithm symbolically computes the following template expression at every location $l_i \neq l_f$:

$$\begin{gathered} B_{l_i} := t_{i,0} + t_{i,1} \cdot \texttt{count} + t_{i,2} \cdot \texttt{a} + t_{i,3} \cdot \texttt{b} + t_{i,4} \cdot \texttt{i} + t_{i,5} \cdot \texttt{j} + \\ t_{i,6} \cdot \texttt{count}^2 + t_{i,7} \cdot \texttt{count} \cdot \texttt{a} + t_{i,8} \cdot \texttt{count} \cdot \texttt{b} + t_{i,9} \cdot \texttt{count} \cdot \texttt{i} + t_{i,10} \cdot \texttt{count} \cdot \texttt{j} + \\ t_{i,11} \cdot \texttt{a}^2 + t_{i,12} \cdot \texttt{a} \cdot \texttt{b} + t_{i,13} \cdot \texttt{a} \cdot \texttt{i} + t_{i,14} \cdot \texttt{a} \cdot \texttt{j} + \\ t_{i,15} \cdot \texttt{b}^2 + t_{i,16} \cdot \texttt{b} \cdot \texttt{i} + t_{i,17} \cdot \texttt{b} \cdot \texttt{j} + \\ t_{i,18} \cdot \texttt{i}^2 + t_{i,19} \cdot \texttt{i} \cdot \texttt{j} + t_{i,20} \cdot \texttt{j}^2 \end{gathered}$$

In other words, the template for the remaining gas polynomial at every location includes all the possible monomials of degree at most 2 over the variables. Moreover, each monomial appears with an unknown coefficient $t_{i,j}$ that should be synthesized.

(4) **Reduction to Entailment Constraints.** In this step, we reduce our main problem of synthesizing the remaining gas polynomial for each cutpoint to solving polynomial entailment constraints of the following standard form:

$$f_1 \bowtie_1 0 \wedge f_2 \bowtie_2 0 \wedge \ldots \wedge f_r \bowtie_r 0 \implies g \bowtie_0 0,$$

in which g and every f_i are polynomials over the PTS variables, whose *coefficients* might be symbolic and contain template variables. Moreover, we have $\bowtie_i \in \{\geq, >\}$.
Recall that $\mathbb{I}$ is a symbolic inductive invariant map and B a symbolic remaining gas polynomial map over the cutset C. The polynomials in both $\mathbb{I}$ and B are over the PTS variables. However, their coefficients are not concrete real numbers, but rather symbolic expressions over the template variables. This is why we call them symbolic polynomials.
Let P be the set of all basic paths in the PTS T. Let Φ be the set of constraints generated until this point of the algorithm. For each basic path $\Pi \in P$ going from l to l', and having transition effect τ, we add the following constraint to Φ :

$$\mathbb{I}(l) \wedge \tau_G \implies \tau_U(\mathbb{I}(l')) \wedge B_l - \tau_U(B_{l'}) \geq C(\Pi).$$

The first conjunct $\tau_U(\mathbb{I}(l'))$ of the conclusion makes sure that the $\mathbb{I}$ is an inductive invariant map, and the second conjunct $B_l - \tau_U(B_{l'}) \geq C(\Pi)$ of the conclusion ensures that B is a valid remaining gas polynomial map.

Example. We give the constraints for one basic path. Other basic paths are handled similarly. In T_1, consider the basic path that starts at l_1, goes to l_2 with a cost of 1 and then back to l_1 with a cost of 2. This basic path can only be taken if $\texttt{p} < |\texttt{proposals}|$. Moreover, it increases the value of p by 1. Thus, our algorithm computes the following constraint:

$$\mathbb{I}(l_1) \wedge \texttt{p} < |\texttt{proposals}| \implies \mathbb{I}(l_1)[\texttt{p} \leftarrow \texttt{p} + 1] \wedge B_{l_1} - B_{l_1}[\texttt{p} \leftarrow \texttt{p} + 1] \geq 3.$$

Suppose that the invariant at l_1 is $\mathbb{I}(l_1) := \texttt{p} \geq 0 \wedge \texttt{p} \leq |\texttt{proposals}| + 1$. The algorithm expands the constraint above and obtains:

$$\begin{gathered} \texttt{p} \geq 0 \wedge \texttt{p} \leq |\texttt{proposals}| + 1 \wedge \texttt{p} < |\texttt{proposals}| \implies \\ \textbf{p + 1} \geq 0 \wedge \textbf{p+1} \leq |\texttt{proposals}| + 1 \wedge \end{gathered}$$

$$t_{1,0} + t_{1,1} \cdot \texttt{winningVoteCount} + t_{1,2} \cdot \texttt{p} + t_{1,3} \cdot |\texttt{proposals}| + t_{1,4} \cdot \texttt{winningProposal} -$$
$$t_{1,0} - t_{1,1} \cdot \texttt{winningVoteCount} - t_{1,2} \cdot \textbf{(p + 1)} - t_{1,3} \cdot |\texttt{proposals}| - t_{1,4} \cdot \texttt{winningProposal} \geq 3$$

The algorithm simplifies the constraint using standard algebraic operations and obtains:

$$\texttt{p} \geq 0 \ \wedge \ \texttt{p} < |\texttt{proposals}| \implies$$
$$\texttt{p} + 1 \geq 0 \ \wedge \ \texttt{p} + 1 \leq |\texttt{proposals}| + 1 \ \wedge \ -t_{1,2} \geq 3.$$

(5) **Reducing Entailment Constraints to Quadratic Constraints.** In the last step, the problem was reduced to solving a system of entailment constraints of the following form:

$$\bigwedge_i f_i \geq 0 \implies g \geq 0.$$

Note that our approach can also handle strict inequalities. However, we focus on non-strict inequalities in the presentation. In the constraint above g and each f_i are symbolic polynomials over the variables $\mathbb{V}$ whose coefficients are symbolic expressions over the set $\mathbb{T}$ of template variables. Our goal is to find concrete values for each template variable $t \in \mathbb{T}$, such that the constraints above hold for every valuation over $\mathbb{V}$. So, this is a formula in the first-order theory of the reals and hence decidable.

Unfortunately, if we pass our constraints directly to a Non-linear Real Arithmetic (NRA) solver, or an SMT solver, it will have to deal with the notoriously difficult problem of quantifier elimination, since our formula has a quantifier alternation. Solvers for the first-order theory of the reals are famously inefficient and cannot handle even toy examples.

Our main algorithmic breakthrough in this step is to employ certain theorems from polyhedral and real algebraic geometry (Detailed in Section 4) to eliminate this quantifier alternation and reduce the problem to a set of quadratic constraints. This reduced instance does not involve any quantifier alternation and is usually much easier to solve using NRA solvers. Moreover, we will obtain a set of quadratic constraints over the variables in $\mathbb{T}$ only and all variables in $\mathbb{V}$ will be eliminated from the formula.

We now show the reduction for a single entailment constraint. Since our constraints are composed conjunctively, our algorithm applies the same reduction to every one of them and then conjunctively combines the quadratic constraints corresponding to each entailment constraint. We consider several cases:

(a) **Linear.** If both the premise and the conclusion of an entailment constraint consist only of linear inequalities over the variables, then we employ Farkas' Lemma to reduce the problem to quadratic constraints (Theorem 4.1).

Example. We illustrate how we use Farkas' Lemma to handle the following constraint:

$$t_0 \cdot x + y - 2 \geq 0 \wedge 2 \cdot x - t_1 \cdot y + 1 \geq 0 \implies 4 \cdot x - y - 3 \geq 0$$

where $t_0, t_1 \in \mathbb{T}$ are unknown template variables. Suppose we can somehow achieve

$$4 \cdot x - y - 3 \equiv \lambda_0 + \lambda_1 \cdot (t_0 \cdot x + y - 2) + \lambda_2 \cdot (2 \cdot x - t_1 \cdot y + 1)$$

with $\lambda_0, \lambda_1, \lambda_2$ being non-negative real numbers. This equality essentially guarantees that whenever $t_0 \cdot x + y - 2$ and $2 \cdot x - t_1 \cdot y + 1$ are non-negative, $4 \cdot x - y - 3$ will also be non-negative, since we are writing the conclusion as a linear combination of the premises and only using non-negative multipliers λ_i. We can simplify the RHS:

$$4 \cdot x - y - 3 \equiv (\lambda_1 \cdot t_0 + 2 \cdot \lambda_2) \cdot x + (\lambda_1 - \lambda_2 \cdot t_1) \cdot y + (\lambda_0 - 2 \cdot \lambda_1 + \lambda_2).$$

However, this equality should hold for all values of x and y. Hence, this equivalence will be true iff the corresponding coefficients on both sides are equal, i.e., $\lambda_1 \cdot t_0 + 2 \cdot \lambda_2 = 4$,

$\lambda_1 - \lambda_2 \cdot t_1 = -1$, and $\lambda_0 - 2 \cdot \lambda_1 + \lambda_2 = -3$. These are precisely our quadratic constraints. Note that these constraints do not involve any of the PTS variables x and y, and there is also no quantifier alternation. Any solution to this set of quadratic constraints will give us a model for t_0 and t_1 that satisfies the original entailment.
Passing this system of quadratic constraints to an external solver, one possible solution could be $t_0 = 1, t_1 = 3, \lambda_0 = 0, \lambda_1 = 2, \lambda_2 = 1$. Plugging these values back, we obtain the following valid constraint which holds for all values of x and y:

$$\boxed{1} \cdot x + y - 2 \geq 0 \wedge 2 \cdot x - \boxed{3} \cdot y + 1 \geq 0 \implies 4 \cdot x - y - 3 \geq 0$$

(b) **Linear Premise and Non-linear Conclusion.** If the premise is linear but the conclusion is non-linear, then we use Handelman's Theorem (Theorem 4.3) to reduce the entailment constraint to quadratic constraints.

Example. Consider the following constraint: $t_0 \cdot x - 2 \geq 0 \wedge x - t_1 \cdot y \geq 0 \implies x^2 - 8 \cdot y + 4 \geq 0$, where $t_0, t_1 \in \mathbb{T}$ are unknown constants.
We cannot apply the same idea as in Farkas' Lemma since no linear combination of $t_0 \cdot x - 2$ and $x - t_1 \cdot y$ can generate a non-linear polynomial like $x^2 - 8 \cdot y + 4$. The intuition behind applying Handelman's Theorem is that as we know $t_0 \cdot x - 2$ is non-negative, then we can also assume that every power of this polynomial is also non-negative. More specifically, we can assume that $t_0^2 \cdot x^2 - 4 \cdot t_0 \cdot x + 4$ is also non-negative. We can also multiply non-negative powers of our assumed-to-be-non-negative premises together. Suppose we can write $x^2 - 8 \cdot y + 4 \equiv \lambda_0 + \lambda_1 \cdot (t_0 \cdot x - 2) + \lambda_2 \cdot (x - t_1 \cdot y) + \lambda_3 \cdot (t_0^2 \cdot x^2 - 4 \cdot t_0 \cdot x + 4)$ with each λ_i being a non-negative real as in the previous case. Using a similar argument as before, this will guarantee the validity of the constraint. In practice, our algorithm generates all possible products of the LHS inequalities up to a user-defined degree d. However, for brevity, we only included some of these products in this example.
Simplifying and equating the coefficients on both sides, we get $\lambda_3 \cdot t_0^2 = 1$ (coefficients of x^2), $\lambda_1 \cdot t_0 + \lambda_2 - 4 \cdot \lambda_3 \cdot t_0 = 0$ (coefficients of x), $-\lambda_2 \cdot t_1 = -8$ (coefficients of y), and $\lambda_0 - 2 \cdot \lambda_1 + 4 \cdot \lambda_3 = 4$ (constant terms). One possible solution to this system of quadratic constraints is $t_0 = 1, t_1 = 2$ and $\lambda_0 = 0, \lambda_1 = 0, \lambda_2 = 4, \lambda_3 = 1$. This leads to the following valid entailment, which holds for all values of x and y:

$$\boxed{1} \cdot x - 2 \geq 0 \wedge x - \boxed{2} \cdot y \geq 0 \implies x^2 - 8 \cdot y + 4 \geq 0$$

(c) **Non-linear Premise.** Finally, in the most general case, if both the premise and the conclusion might be non-linear, we use Putinar's Positivstellensatz (Theorem 4.3) to reduce the entailment to quadratic constraints.

Example. Consider the following constraint:

$$t_0 \cdot x^3 \geq 0 \implies x^9 + y^2 \geq 0$$

We can not apply Farkas' Lemma for the same reason as the previous example, nor we can apply Handleman's Theorem because no power of $t_0 \cdot x^3$ will generate a polynomial containing y. Intuitively, it is easy to observe that setting $t_0 = 1$ gives us a valid concrete entailment

$$\boxed{1} \cdot x^3 \geq 0 \implies x^9 + y^2 \geq 0$$

because we can write $x^9 + y^2 \equiv x^3 \cdot (x^3)^2 + y^2$. Note that $(x^3)^2$ and y^2 are both sums of squares, therefore they are always non-negative. As x^3 is also assumed to be non-negative in the premise, the conclusion should also be non-negative whenever the premise is non-negative.

Node	Remaining Gas Polynomial
l_0	$101 \cdot \lvert\texttt{proposals}\rvert + 106$
l_1	$101 \cdot \lvert\texttt{proposals}\rvert - 101 \cdot \texttt{p} + 102$
l_2	$101 \cdot \lvert\texttt{proposals}\rvert - 101 \cdot \texttt{p} + 101$
l_f	0

Node	Remaining Gas Polynomial
l_0	$5 \cdot \texttt{a} \cdot \texttt{b} + 11 \cdot \texttt{a} + 5 \cdot \texttt{b} + 16$
l_1	$5 \cdot \texttt{a} \cdot \texttt{b} + 11 \cdot \texttt{a} - 5 \cdot \texttt{b} \cdot \texttt{i} + 5 \cdot \texttt{b} - 11 \cdot \texttt{i} + 12$
l_2	$5 \cdot \texttt{a} \cdot \texttt{b} + 11 \cdot \texttt{a} - 5 \cdot \texttt{b} \cdot \texttt{i} + 5 \cdot \texttt{b} - 5 \cdot \texttt{j} - 11 \cdot \texttt{i} + 9$
l_f	0

Table 1. Synthesized solution for T_1 (left) and T_2 (right)

When we apply Putinar's Positivstellensatz, we simply multiply each polynomial in the premise with a symbolic sum of squares of sufficiently high degree and sum them together. In other words, the multipliers λ_i are no longer non-negative scalar variables, but instead polynomials that are guaranteed to be sum-of-squares. We can write this guarantee as a system of quadratic constraints, too. Then we equate the coefficients of both sides, just as in the previous cases, to obtain a system of quadratic constraints. Solving this system gives a valid concrete constraint.

(6) **Solving the Quadratic Constraints.** The previous step has already reduced the problem to a system quadratic constraints over template variables $\mathbb{T}$ and newly-introduced multipliers λ_i. We pass this system to an external NRA solver. If the NRA solver can find a solution to the system, then we substitute the value of the template variables in $\mathbb{T}$ in our symbolic remaining gas polynomial map B to obtain a concrete parametric bound on the gas usage. Finally, we return the concrete gas polynomial at the start location as the answer.

Example. Table 1 shows one solution of the quadratic system for each of T_1 and T_2. Thus, our approach is able to prove that the total gas cost is at most $101 \cdot \lvert\texttt{proposals}\rvert + 106$ for T_1 and $5 \cdot \texttt{a} \cdot \texttt{b} + 11 \cdot \texttt{a} + 5 \cdot \texttt{b} + 16$ for T_2. Note that the solutions are not unique, but any solution of the system leads to a valid bound on the gas usage of the PTS.

Objective Function. Our algorithm encodes all the requirements on the template variables $t_{i,j}$ first as entailment constraints and then as a system of quadratic constraints. Thus, every solution of this system, when plugged back into the template for B_{l_0} leads to a valid bound on the gas usage of the PTS. In our experiments, we do not use an objective function, but one can generally specify any objective function based on the unknown template variables $t_{i,j}$. This would lead to a Quadratically-Constained Quadratic Programming (QCQP) instance.

Pseudocode. Algorithm 1 provides a pseudocode of our approach.

Algorithm 1: Asparagus

Input: A smart contract A
Input: Start block S, degree bound d for the polynomials
Result: Remaining Gas Polynomial B_{l_0} at the initial location corresponding to S

$T \leftarrow$ parsePTS(A) ▷ Generate the PTS T for A (Steps 1 and 2) – Not guaranteed to preserve completeness.;
$C \leftarrow$ getCutset(T);
$I \leftarrow$ TemplateInvariantMap(C, T, d) ▷ Step 3;
$B \leftarrow$ TemplateRemainingGasMap(C, T, d);
$QS \leftarrow$ **true**;
foreach *Basic Path* Π *in* T *w.r.t.* C **do**
  $(\phi \Rightarrow \psi) \leftarrow$ GetConstraintPair(Π, T) ▷ Step 4;
  if ϕ, ψ *are both linear* **then**
    $QS_c \leftarrow$ ReduceWithFarkas(ϕ, ψ) ▷ Step 5a ;
  end
  else if ϕ *is linear* **then**
    $QS_c \leftarrow$ ReduceWithHandelman(ϕ, ψ, d) ▷ Step 5b;
  end
  else
    $QS_c \leftarrow$ ReduceWithPutinar(ϕ, ψ, d) ▷ Step 5c;
  end
  $QS \leftarrow QS \wedge QS_c$
end
Model $\leftarrow$ NRA_Solver(QS) ▷ Step 6;
if Model *is UNSAT* **then**
  return Failed
end
return ConcretePolynomial$(B_S,$ Model$)$ ▷ The concrete remaining gas polynomial;

4 MATHEMATICAL DETAILS

In this section, we fill in the mathematical details of our algorithm using Farkas' Lemma, Handelman's Theorem and Putinar's Positivstellensatz. The previous section has already illustrated how we use these theorems in our algorithm.

Notation. We first set down some basic notation:

(1) Given a boolean formula $\phi := \bigwedge_i (f_i \geq 0)$ over the variables $\mathbb{V}$, we define

$$\mathrm{SAT}(\phi) := \{\sigma : \mathbb{V} \to \mathbb{R} \mid \bigwedge_i \sigma \models f_i \geq 0\}$$

(2) A polynomial $h \in \mathbb{R}[\mathbb{V}]$ is said to be a *sum of squares* (SOS) iff it can be written as a finite sum $h \equiv \sum_i g_i^2$ where $g_i \in \mathbb{R}[\mathbb{V}]$ are arbitrary polynomials.

4.1 Farkas Lemma

We start by stating Farkas' Lemma, which is a well-known result in polyhedral geometry and linear algebra.

Theorem 4.1 (Farkas' Lemma [Farkas 1902]). *Let $f_1, \ldots f_r$ and g be any linear polynomials over the set of variables $\mathbb{V}$. The boolean implication*

$$f_1 \geq 0 \wedge f_2 \geq 0 \ldots \wedge f_r \geq 0 \implies g \geq 0$$

holds for every valuation of $\mathbb{V}$ ***if and only if*** *there exist $\lambda_0, \lambda_1, \ldots, \lambda_r$, such that*

$$g \equiv \lambda_0 + \sum_{i=1}^{r} \lambda_i \cdot f_i$$

where all λ_i's are non-negative reals. Moreover, $f_1 \geq 0 \wedge f_2 \geq 0 \ldots \wedge f_r \geq 0$ is unsatisfiable ***if and only if*** *$-1 \equiv \lambda_0 + \sum_{i=1}^{r} \lambda_i f_i$ for some non-negative real λ_i's.*

Example 4.2. Let $f_1 := -x - 2 \cdot y + z - 5$, $f_2 := x + 3 \cdot y - 2 \cdot z - 10$, $f_3 := 2 \cdot x + 4 \cdot y + 5 \cdot z + 5$ and $g := 3 \cdot y + 5 \cdot z - 15$. We can readily see that $g \equiv 2 \cdot f_1 + f_2 + f3$. Therefore, the implication $\phi := f_1 \geq 0 \land f_2 \geq 0 \implies g \geq 0$ holds. Note that Farkas' lemma is, in a sense, both sound and complete. If the entailment holds, then it guarantees that suitable λ_i's exist.

Pseudocode. Algorithm 2 provides a more formal presentation of how Farkas' lemma is used to reduce entailments to a system of quadratic constraints.

Algorithm 2: ReduceWithFarkas

Input: An entailment constraint $f_1 \geq 0 \land f_2 \geq 0 \ldots \land f_r \geq 0 \implies g \geq 0$ over the set $\mathbb{V}$ of PTS variables
Result: A system QS of quadratic constraints
Create fresh symbolic variables $\lambda_0, \lambda_1, \ldots, \lambda_r$;
$QS \leftarrow \lambda_0 \geq 0 \land \lambda_1 \geq 0 \land \ldots \land \lambda_r \geq 0$;
$eq \leftarrow \lambda_0 + \sum_{i=1}^{r} \lambda_i \cdot f_i - g$ ▷ Computed symbolically;
foreach *monomial* $m \in \mathbb{V} \cup \{1\}$ **do**
    $\texttt{coeff}_m \leftarrow$ the coefficient of m in eq;
    $QS \leftarrow QS \land \texttt{coeff}_m = 0$;
end
return QS;

4.2 Handelman's Theorem

For our next theorem, we first need the concept of a monoid.

Monoid. Let $F := \{f_1, f_2, \ldots, f_r\}$ be a set of polynomials over $\mathbb{V}$. We define the monoid of F as the set:

$$monoid(F) := \{f_1^{k_1} \cdot f_2^{k_2} \cdots f_r^{k_r} \mid k_1, \ldots, k_r \in \mathbb{N}\}. \tag{2}$$

In other words, the monoid includes all polynomials that can be obtained as a multiplication of polynomials in F. We also define

$$monoid_d(F) := \{f \mid f \in monoid(F) \land \deg(f) \leq d\}.$$

Theorem 4.3 (Handelman's Theorem [Handelman 1988]). *Let $f_1, \ldots f_r$ be any linear polynomials and g be an arbitrary polynomial over $\mathbb{V}$ such that $SAT(f_1 \geq 0 \land f_2 \geq 0 \land \ldots \land f_r \geq 0)$ is a non-empty compact set. The boolean implication*

$$f_1 \geq 0 \land f_2 \geq 0 \land \ldots \land f_r \geq 0 \implies g \geq 0$$

holds over real numbers ***if and only if*** *there exist λ_i, m_i such that $g \equiv \sum_{i=1}^{r} \lambda_i \cdot m_i$, in which each λ_i is a non-negative real number and each $m_i \in monoid(\{f_1, \ldots, f_r\})$.*

Example 4.4. Let $f_1 := -x - 2 \cdot y - 5$, $f_2 := 2 \cdot x + 3 \cdot y - 2$, and $g := 2 \cdot y - 2 \cdot x \cdot y - 3 \cdot y^2$. We only use the monoid of degree 2 in this example: $monoid_2(\{f_1, f_2\}) = \{1, f_1, f_2, f_1^2, f_2^2, f_1 \cdot f_2\}$. It can be verified that $g \equiv 2 \cdot f_1 \cdot f_2 + f_2^2 + 12 \cdot f_2$. Therefore, by Handelman's Theorem, the implication $\phi := f_1 \geq 0 \land f_2 \geq 0 \implies g \geq 0$ holds for every possible real valuation of x and y.

Pseudocode. Algorithm 3 shows how Handelman's theorem reduces entailments to a system of quadratic constraints.

Algorithm 3: ReduceWithHandelman

Input: An entailment constraint $f_1 \geq 0 \wedge f_2 \geq 0 \ldots \wedge f_r \geq 0 \implies g \geq 0$ over the set $\mathbb{V}$ of PTS variables
Input: A degree bound d
Result: A set of quadratic inequalities
$M_d \leftarrow monoid_d(f_1, \ldots, f_r)$;
$r \leftarrow |M_d|$;
Create fresh symbolic variables $\lambda_0, \lambda_1, \ldots, \lambda_r$;
$QS \leftarrow \lambda_0 \geq 0 \wedge \lambda_1 \geq 0 \wedge \ldots \wedge \lambda_r \geq 0$;
$eq \leftarrow \lambda_0 + \sum_{i=1}^{r} \lambda_i \cdot M_d[i] - g$ ▷ Computed symbolically;
foreach *monomial* m *of* $\mathbb{R}[\mathbb{V}]$ **do**
  $\texttt{coeff}_m \leftarrow$ the coefficient of m in eq;
  $QS \leftarrow QS \wedge \texttt{coeff}_m = 0$;
end
return QS;

4.3 Putinar's Positivstellensatz

Finally, our strongest hammer is Putinar's Positivstellensatz. Positivstellensatz is German for "positive locus theorem" and denotes a theorem that characterizes positive polynomials over semi-algebraic sets.

Theorem 4.5 (Putinar's Positivstellensatz [Putinar 1993]). *Let $f_1, \ldots f_r$ and g be polynomials of any degree over $\mathbb{V}$ such that $SAT(f_i)$ is compact for at least one f_i. The boolean implication*

$$f_1 \geq 0 \wedge f_2 \geq 0 \ldots \wedge f_r \geq 0 \implies g > 0$$

holds over real numbers ***if and only if*** *g can be written as*

$$g \equiv h_0 + \sum_{i=1}^{r} h_i \cdot f_i \tag{3}$$

in which each h_i is a sum of squares.

Example 4.6. As an example, let

$$f_1 := x + y - 1 \qquad f_2 := -x^2 - y^2 + 10$$
$$f_3 := -2 \cdot x^3 - 4 \cdot x \cdot y^2 + 10 \qquad g := 2 \cdot x^2 \cdot y - 4 \cdot x \cdot y^2 + 10$$

We can choose the sum of squares $h_0 = 2 \cdot x^2, h_1 = 2 \cdot x^2, h_2 = 0$ and $h_3 = 1$, and write $g \equiv h_0 + h_1 \cdot f_1 + h_2 \cdot f_2 + h_3 \cdot f_3$. Therefore, the boolean implication $\phi := f_1 \geq 0 \wedge f_2 \geq 0 \wedge f_3 \geq 0 \implies g \geq 0$ holds.

SOS Templates. Recall that, in order to apply Putinar's Positivstellensatz, we needed to multiply each polynomial f_i with a sum of squares h_i. In our algorithm, we used a template for h_i. We now discuss how such a template can be generated automatically. Given a degree bound $2 \cdot m$, we can use Theorems 4.7 and 4.8 below to generate a general symbolic polynomial template for a SOS of degree $2 \cdot m$.

Theorem 4.7 ([Blekherman et al. 2012]). *Let V_m be the vector of all monomials of degree at most m over the variables $\mathbb{V} = \{x_1, \ldots, x_n\}$. A polynomial $h \in \mathbb{R}[\mathbb{V}]$ of degree $2 \cdot m$ is a sum of squares* ***if and only if*** *there exists a symmetric positive semi-definite matrix Q such that $h = V_m^T \cdot Q \cdot V_m$.*

Theorem 4.8 (Cholesky Decomposition, [Watkins 2004]). *A square symmetric matrix Q is positive semi-definite* ***if and only if*** *there exists a lower-triangular matrix L with non-negative diagonal entries such that $Q = L \cdot L^T$.*

Hence, it suffices to generate a template for the lower-triangular matrix L, introducing new template variables for each of the entries. We can then simply compute a template for h by setting

$$h := V_m^T \cdot L \cdot L^T \cdot V_m.$$

Pseudocode. Based on the theorems above, Algorithm 4 provides a more formal presentation of how Putinar's Positivstellensatz reduces entailments to a system of quadratic constraints.

Algorithm 4: ReduceWithPutinar

Input: An entailment constraint $f_1 \geq 0 \wedge f_2 \geq 0 \ldots \wedge f_r \geq 0 \implies g > 0$ over the set $\mathbb{V}$ of PTS variables
Input: An even degree bound d
Result: A system QS of quadratic constraints

$V_{d/2} \leftarrow monoid_d(\mathbb{V})$ ▷ All monomials of degree at most $d/2$ over $\mathbb{V}$;
$w \leftarrow |V_{d/2}|$;
$QS \leftarrow \texttt{true}$;
foreach $0 \leq i \leq r$ **do**
  $L \leftarrow$ a $w \times w$ lower-triangular matrix whose every non-zero entry at position (j,k) is a fresh symbolic variable $\lambda_{i,j,k}$;
  foreach $1 \leq j \leq w$ **do**
    $QS \leftarrow QS \wedge \lambda_{i,j,j} \geq 0$ ▷ Non-negative diagonal entries;
  end
  $h_i \leftarrow V_{d/2}^T \cdot L \cdot L^T \cdot V_{d/2}$ ▷ Computed symbolically;
end
$eq \leftarrow h_0 + \sum_{i=1}^{r} h_i \cdot f_i - g$ ▷ Computed symbolically;
foreach *monomial* m *of* $\mathbb{R}[\mathbb{V}]$ **do**
  $\texttt{coeff}_m \leftarrow$ the coefficient of m in eq;
  $QS \leftarrow QS \wedge \texttt{coeff}_m = 0$;
end
return QS;

4.4 Summary of the Mathematical Tools

The following table summarizes how our algorithm applies the three theorems above in order to reduce a constraint of the form $f_1 \geq 0 \ldots \wedge f_r \geq 0 \implies g \geq 0$ to a system of quadratic constraints.

Theorem	$F := f_1, \ldots, f_r$	g	Equality Used
Farkas' Lemma	Linear	Linear	$g \equiv \lambda_0 + \sum_{i=1}^{r} \lambda_i \cdot f_i$ where each λ_i is a non-negative real
Handelman's Theorem	Linear	Non-linear	$g \equiv \sum_i \lambda_i \cdot m_i$ where $m_i \in monoid_d(F)$
Putinar's Positivstellensatz	Non-linear	Arbitrary	$g \equiv h_0 + \sum_i h_i \cdot f_i$ where each h_i is an SOS

Table 2. Summary of the theorems used to reduce an entailment to quadratic constraints

We remark that Putinar's Positivstellensatz is more general than Farkas' Lemma and Handelman's theorem. Putinar can handle antecedents and consequents that are both polynomial, whereas Handelman requires linear antecedents and Farkas additionally requires linear consequents. In cases where more than one theorem is applicable, the quality of the bounds will not be affected and they can synthesize the exact same set of bounds. In other words, no theorem is tighter. This is because they are all complete. In practice, we prefer to first use Farkas if possible, and switch to Handelman only if Farkas fails, and then to Putinar if Handelman fails. Our tool makes heuristic choices to decide which entailments should be kept in Handelman and which escalated to Putinar. The preference for Farkas over Handelman over Putinar is because there is a tradeoff between generality and the number of new variables introduced by each approach. Farkas introduces the

fewest number of extra variables, namely the λ_i's in Theorem 4.1, whereas Handelman has to generate a coefficient for each monoid element, i.e. λ_i's in Theorem 4.3, and Putinar needs to set up a template in which we have a new variable for each non-zero entry of the lower-triangular matrix L of Theorem 4.8.

4.5 Soundness, Completeness and Complexity

We end this section by stating that our algorithm is both sound and semi-complete.

Theorem 4.9. *Given a PTS T over variables with bounded values, a cutset C of T, a polynomial inductive invariant $\mathbb{I}$, and a template for a remaining gas polynomial B_l at each cutpoint $l \in C$, the algorithm of Section 3 has the following desired properties:*

- Soundness*: Every solution to the system of quadratic constraints in Step 6 leads to valid remaining gas polynomials when plugged back into the template $\{B_l\}_{l\in\mathbb{L}}$. We say that a set of remaining gas polynomials are valid if they satisfy Equation (1).*
- Semi-completeness*: For every valid set $\{B_l^*\}_{l\in\mathbb{L}}$ of remaining gas polynomials, if the degree d chosen by the user is large enough, i.e. if polynomials of large enough degree are used in the templates, then there exists a solution of the system of quadratic constraints in Step 6 that when plugged into the templates $\{B_l\}_{l\in\mathbb{L}}$ leads to $\{B_l^*\}_{l\in\mathbb{L}}$.*

Proof. The entailment constraints generated in Step 4 precisely model the requirements of Equation (1). Thus, this step is sound. In Step 5, each of the three possible reductions are sound. In case 5a, the RHS is written as a linear combination of the LHS polynomials with non-negative coefficients λ_i. If this is successful, then it is trivial that the entailment holds, since a linear combination of non-negative inequalities with non-negative coefficients is guaranteed to be non-negative. In case 5b, we are first considering products of the non-negative inequalities on the LHS. Such products are of course non-negative. We then write the RHS as a linear combination of these products with non-negative coefficients. Thus, the same argument as in the previous case applies. Finally, case 5c is also similar to case 5a, except that the coefficients which were non-negative real constants are now replaced by sum-of-square polynomials. However, the same argument stands since sum-of-square polynomials are always non-negative. One can essentially repeat the same argument for corner cases involving strict inequalities. See [Goharshady 2020, Chapter 7] for details.

For completeness, note that Step 4 is a precise encoding of Equation (1) and thus complete. Step 5a is complete due to Farkas' lemma (Theorem 4.1) which guarantees that whenever the entailment constraint holds, the RHS can be written as a linear combination of the LHS with non-negative coefficients λ_i. Similarly, Step 5b is complete due to Theorem 4.3 guaranteeing that if the entailment holds, then the RHS can be written as a combination of products of the LHS. Finally, the completeness of Step 5c is due to Theorem 4.3 in which the validity of the entailment $f_1 \geq 0 \wedge f_2 \geq 0 \ldots \wedge f_r \geq 0 \implies g > 0$ guarantees $g \equiv h_0 + \sum_{i=1}^{r} h_i \cdot f_i$, where each h_i is a sum-of-squares. Of course, the products generated in Step 5b and the sums-of-squares in Step 5c should have a high enough degree, since the theorems do not have a degree bound. This is why we call this semi-completeness, i.e. completeness as long as the degree d chosen for the templates is large enough. Finally, we note that both Theorem 4.3 and Theorem 4.3 have a compactness requirement. This holds naturally in our case since the variables in a smart contract are always bounded. Assuming that the largest possible value for a variable x is m, we can add the invariant $-m \leq x \leq m$ to every line of the program. This puts our valuations inside a bounded hypercube. Finally, by the Heine-Borel theorem, a subset of $\mathbb{R}^n$ is compact if and only if it is closed and bounded. Thus, the hypercube is compact. □

Theorem 4.10. *Given the same inputs as in the previous theorem, the runtime of our reduction to a system of quadratic constraints (Steps 4 and 5) is polynomial in the size n of the PTS and depends exponentially on the number of variables and the degree bound d.*

Proof. Step 4 generates one entailment constaint for each transition in the PTS. Thus, its runtime is $O(n)$. For each entailment constraint, we have the following analysis:

- Step 5a (Farkas' lemma) generates at most $|\mathbb{V}|$ fresh variables λ_i and the same number of quadratic equalities.
- Step 5b creates all monomials of degree up to d as in Equation (2). The number of such monomials is $\binom{d+|\mathbb{V}|}{d}$. There is at most one quadratic equality generated per monomial.
- Step 5c creates a constant number of sum-of-square polynomials h_i as in Equation (3). To generate a template for each such polynomial, we need to first generate all monomials of degree at most $d/2$ as in Theorem 4.7 and then form the L matrix of Theorem 4.8 which has $O(\binom{d/2+|\mathbb{V}|}{d/2}^2)$ entries. We will have at most one quadratic equality for each monomial.

The time spent in generating each quadratic equality is bounded by a polynomial in its size. Thus, the total runtime is polynomial in the size n of the PTS but depends exponentially on the number of variables in $\mathbb{V}$ and the degree bound d. □

5 EXPERIMENTAL RESULTS

Implementation. We implemented our algorithm of Section 3 for Ethereum smart contracts and named our automated tool ***Asparagus***[6]. Our implementation is in Python 3 and can obtain linear and polynomial gas usage upper-bounds for public functions of any given contract written in Solidity. We used Slither [Feist et al. 2019] for parsing, `solc` 0.4.25 for compilation [Ethereum Foundation 2014], Z3 [de Moura and Bjørner 2008] and Mathsat [Cimatti et al. 2013] to solve the final systems of quadratic constraints, and EthIR [Albert et al. 2018] to generate RBR intermediate representation. The implementation is open-source and dedicated to the public domain with a CC0 (no rights reserved) license. It is available as an archived artifact attached to this article.

Benchmarks and Experimental Setting. We compare Asparagus with GASTAP [Albert et al. 2021], which is the only previous tool able to generate parametric bounds, as well as the `solc` compiler's built-in static analyzer [Ethereum Foundation 2014]. As benchmarks, we took the dataset provided by GASTAP. Since both GASTAP and Asparagus use EthIR in their pipeline, we excluded any benchmark on which EthIR failed to generate RBR. Surprisingly, such benchmarks existed and we could not replicate and verify the experimental claims of GASTAP [Albert et al. 2021]. We suspect this might have to do with EthIR updates that were aimed at supporting newer versions of Solidity, and have likely reduced its applicability significantly in comparison to what GASTAP reports. Irrespective of the cause, our results have significant mismatches with GASTAP's claims. In total, we report experimental results on 24,188 contracts, containing 156,735 functions. Figure 8 shows the distribution of contract lengths in our benchmark suite and Figure 9 reports the number of basic paths in the analyzed functions.

Machine and Runtimes. All computations were performed on an Intel Xeon Gold 5115 CPU (2.40GHz, using 16 cores) running Ubuntu 20.04 and 64 GB of RAM. Our average runtime on each function was 5.18 seconds, leading to a total runtime of almost 225.8 hours for 156,735 functions. This demonstrates that our approach is scalable and easily applicable to real-world contracts.

Comparing Bounds. In cases where the bounds are constants, we can simply compare them. We say a parametric bound A is tighter than another bound B if for any initial state σ that satisfies

[6] Automated synthesis of parametric gas upper-bounds for smart contracts

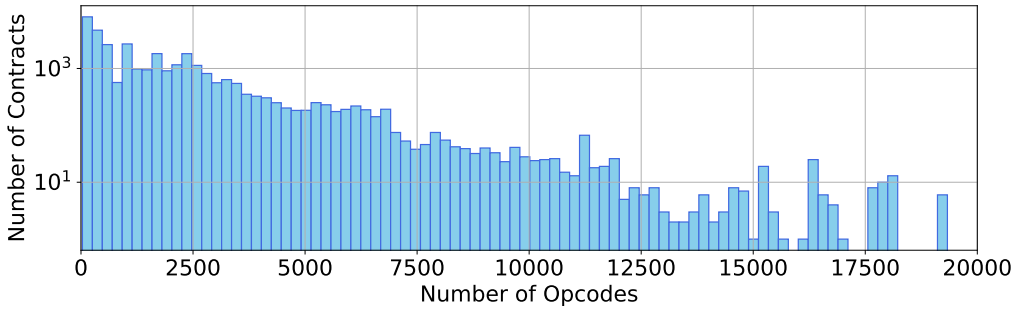

Fig. 8. Distribution of Contract Lengths among the Benchmarks.

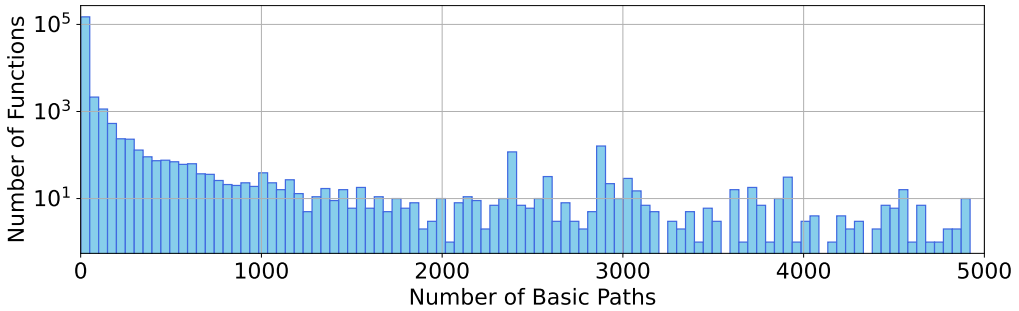

Fig. 9. Number of Basic Paths in the Analyzed Functions.

the precondition, i.e. the invariant at the beginning point of the function, we have $A(\sigma) \leq B(\sigma)$ and additionally, $A(\sigma)$ is strictly less in at least one such state. To compare bounds, we again use the theorems of Farkas, Handelman and Putinar to (i) check whether $\mathbb{I}(l_0) \Rightarrow A \leq B$ holds, and (ii) synthesize a particular state that satisfies $\mathbb{I}(l_0) \wedge A < B$. If this fails, we report that the bounds are incomparable.

Correctness of Our Bounds. We took the bounds reported by other tools on face value. However, for our own bounds, we plugged them back into the three theorems (Farkas, Handelman, Putinar) and verified that they satisfy the requirements. Thus, we are confident that numerical or other errors in the solvers have not compromised our results.

Comparison with `solc`. We first compared Asparagus and `solc` on the entire dataset of 156,735 functions. Our tool successfully synthesized gas usage upper-bounds for **80.56%** of the instances compared to a success rate of **64.15%** for `solc`. Note that `solc` can only synthesize constant bounds and hence fails on all benchmarks that require parametric bounds. Table 3 provides a comparison of success rates and coverage of the two tools. In cases where both approaches could successfully find a bound, our bound was better in 81,425 cases. Figure 10 provides a comparison of the *constant* bounds obtained by the two tools.

Comparison with GASTAP. We also compared Asparagus with GASTAP. Unfortunately, we did not have the desired degree of access to GASTAP in order to perform a complete comparison due to the following reasons:

	Asparagus		solc	
	#	%	#	%
Constant Gas Bound	124,987	–	100,550	–
Parametric Gas Bound	1,282	–	0	–
Solved	126,269	80.56%	100,550	64.15%
Failed	30,466	19.43%	56,185	35.84%

Table 3. Number and percentage of benchmark functions solved by Asparagus and solc on 156,735 functions.

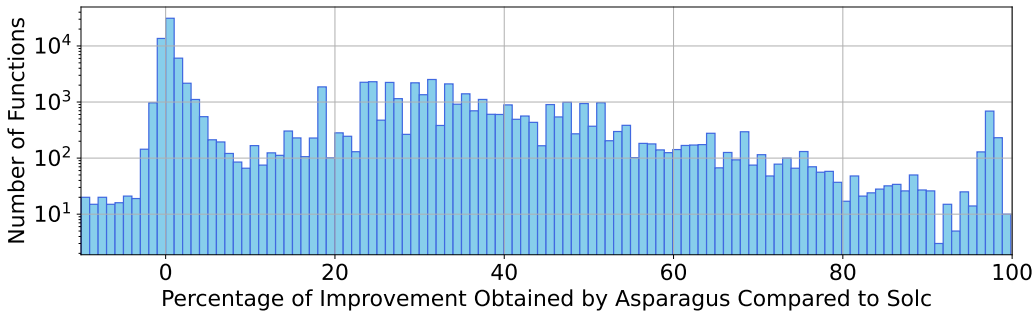

Fig. 10. Comparsion of the Constant Bounds obtained by Asparagus and solc.

- GASTAP did not publish its gas upper-bound results for the dataset.
- GASTAP is a closed-source and proprietary piece of software that can only be accessed by an online web interface[7] with no API. To perform the comparison, we had to reverse-engineer their http requests and even then, most of our requests failed, even after sending them a dozen times each. We manually checked some of the failed requests on their online interface, and confirmed that the failures are due to GASTAP.

Due to these limitations, we could successfully run GASTAP on only 1,682 contracts, consisting of 10,186 functions. We thus report a comparison on these 10,186 functions. On these benchmarks, Asparagus successfully synthesized a gas upper-bound for **86.94%** of the functions compared to **58.62%** for GASTAP and **61.61%** for solc, demonstrating Asparagus' effectiveness. Table 4 provides a more detailed comparison and Figure 11 compares the *constant* bounds obtained by the two tools.

	Asparagus		GASTAP		solc	
	#	%	#	%	#	%
Constant Gas Bound	8,404	–	5,846	–	6,276	–
Parametric Gas Bound	452	–	126	–	0	–
Solved	8,856	86.94%	5,972	58.62%	6,276	61.61%
Failed	1,330	13.05%	4,214	41.37%	3,910	38.38%

Table 4. Number and percentage of benchmark functions solved by Asparagus, GASTAP and solc on 10,186 functions.

[7] costa.fdi.ucm.es/gastap

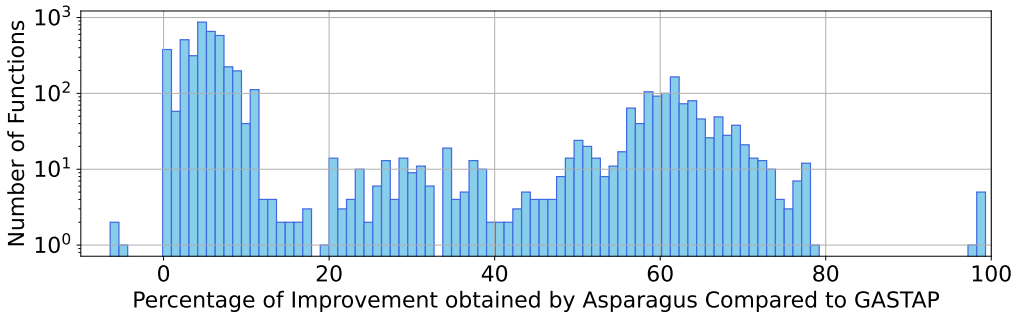

Fig. 11. Comparsion of the Constant Bounds obtained by Asparagus and GASTAP.

Notably, we compared the bounds in cases where both Asparagus and GASTAP were successful (5,916 functions). Asparagus' synthesized upper-bounds were strictly tighter in 5,789 cases, representing **97.85%**. GASTAP provided tighter bounds for only 45 functions, representing **0.76%** of the instances. In other cases, the bounds were either equal or incomparable.

Non-linear Bounds. The vast majority of real-world contracts have constant or linear gas usage. In total, only 653 of our benchmarks required a polynomial bound and an application of the theorems of Handelman and Putinar. The rest of the instances were solved by Farkas' Lemma. Our implementation supports polynomials of any degree. We have experimented with nested loops requiring up to hexic bounds and successfully synthesized results on hand-crafted examples. However, none of the real-world benchmarks in our dataset needed bounds of degree higher than 2.

Summary. Asparagus is able to handle significantly more real-world benchmarks than both `solc` and GASTAP. Since GASTAP was the only previous tool that could generate parametric bounds, Asparagus is now the most widely-applicable tool for such bounds to the best of our knowledge. Moreover, we consistently generate tighter bounds than both GASTAP and `solc`.

Contracts beyond Asparagus' Capability. As mentioned above, Asparagus successfully handled 80.56% of the cases in our benchmark suite (See Table 3). This is considerably more than `solc` and GASTAP. The remaining benchmarks were beyond the capability of our tool, due to non-polynomial operations and bounds. Our PTS formalism can handle polynomial assignments, guards, templates and bounds. Thus, we replace non-polynomial behavior, such as the integer mod operation, with non-determinism, cf. Steps 1 and 2 of the algorithm. This can lead to failures in synthesizing bounds. Moreover, in some cases, no polynomial bound exists at all. Our completeness result guarantees that our approach will succeed if (i) the contract can be faithfully modeled as a PTS with polynomial assignments, invariants and templates, and (ii) the PTS has a polynomial gas usage bound. In practice, all real-world contracts on which Asparagus failed were violating condition (i) and had non-polynomial inherently-integer operations such as gcd or mod.

6 RELATED WORKS

Works on Smart Contracts. Since smart contracts are immutable and safety-critical, i.e. handling millions or possibly even hundreds of millions of dollars, any security vulnerability can lead to catastrophic financial losses [Atzei et al. 2017]. The most successful attack to date is the DAO hack, which managed to steal more than 60 million dollars from a contract called the DAO. The damage was partially reverted by a hard fork of the Ethereum blockchain [Siegel 2016], causing significant disagreement in the community and leading to a division of Ethereum into two separate

cryptocurrencies, the other being Ethereum classic. There are currently many program analysis and vulnerability detection tools for smart contracts [Atzei et al. 2017; Sayeed et al. 2020], both using static analysis [Feist et al. 2019; Kalra et al. 2018; Luu et al. 2016] and dynamic [Nguyen et al. 2020; Nikolic et al. 2018; Rodler et al. 2019]. The following works consider gas-related vulnerabilities and gas bounds:

- MadMax [Grech et al. 2018] automatically detects gas-related vulnerabilities using a combination of control-flow analysis and declarative structural queries.
- GasChecker [Chen et al. 2021] detects ten gas-inefficient programming patterns in bytecode by using symbolic execution.
- V-Gas [Ma et al. 2022] uses feedback-directed fuzz testing to automatically generate inputs that can potentially lead to high overall gas costs.
- Gas Gauge [Nassirzadeh et al. 2021] checks if the contract is at risk of DoS based on exceeding the block gas limit.
- GASOL [Albert et al. 2020] detects gas-expensive code segments and automatically optimizes them. This is achieved by trying to replace multiple accesses to the same position in storage with gas-efficient memory accesses. These transformations are applied at the level of Solidity. In [Correas et al. 2021], the authors generalized GASOL to handle expressions that involve more than one cost model. This enabled them to develop a theoretical framework that has further applications such as the detection of various gas-related vulnerabilities.
- The standard Solidity compiler `solc` [Ethereum Foundation 2014] has a built-in static analyzer that generates gas upper-bounds. However, it can only handle constant bounds and when parametric bounds are needed, it simply returns $+\infty$. The same limitation applies to several other tools, such as [Marescotti et al. 2018].
- Despite the amazing progress in detecting and repairing gas-related vulnerabilities through manual security audits, manual approaches cannot keep pace with the growing complexity, size, and number of smart contracts and also lead to an unacceptably high rate of false negatives [Nassirzadeh et al. 2021]. Nevertheless, manual security audits remain an integral part of the smart contract development process.
- To the best of our knowledge, GASTAP [Albert et al. 2021] is the only previous tool that automatically synthesizes parametric gas bounds for Ethereum smart contracts. We have provided a detailed experimental comparison with GASTAP in Section 5. Finally, it is noteworthy that GASTAP's parametric bounds are limited to the linear case, whereas our approach provides sound and semi-complete synthesis for polynomial bounds of any degree.

Farkas' Lemma in Static Analysis. Template-based static analysis using Farkas' lemma is a standard technique applied in several previous works, such as [Colón et al. 2003; Liu et al. 2022; Sankaranarayanan et al. 2004] for invariant generation and [Chatterjee et al. 2017b; Podelski and Rybalchenko 2004] to synthesize ranking functions or their probabilistic extensions and thus prove termination. Our gas usage bounds can be seen as a generalization of ranking functions with weights, i.e. every transition has to reduce the ranking function's value not by a unit, but by its gas usage. As such, our approach for the linear case is a direct extension of [Podelski and Rybalchenko 2004]. This being said, 653 of the functions in our benchmark suite require polynomial bounds of higher degrees. Moreover, the gas usage of some opcodes in the EVM, such as those allocating memory, has a non-linear (quadratic) dependence on the size of an array. Thus, an approach based solely on Farkas' lemma, which can only handle linear transition systems and costs, is not general enough for our setting.

Positivstellensätze in Static Analysis. Extending Farkas-based algorithms from linear to polynomial programs using Positivstellensätze is a direction that has been studied for many families

of static analyses, including termination and runtime analysis [Chatterjee et al. 2016, 2017a, 2019, 2022, 2021b, 2023, 2021a; Huang et al. 2018, 2019; Sun et al. 2023; Wang et al. 2021], invariant generation [Chatterjee et al. 2020], reachability analysis [Asadi et al. 2021], syntax-guided synthesis [Goharshady et al. 2023] and stochastic shortest paths in MDPs [Chatterjee et al. 2018].

Resource Analysis. Our work can also be seen as a tailored version of the general resource analysis problem, where the resource is gas and the programs are smart contracts. Resource analysis, and especially the problem of finding or verifying bounds on the worst-case execution time of a program, have been extensively studied in the literature using many different approaches, e.g. [Albert et al. 2009; Atkey 2011; Bygde 2010; Çiçek et al. 2017, 2020; Hoffmann et al. 2012, 2017; Wang et al. 2019b; Wilhelm et al. 2008]. Indeed, GASTAP [Albert et al. 2021] is based on the COSTA project [Albert et al. 2009]. To the best of our knowledge, none of the previous works in resource analysis are applicable to our setting of polynomial transition systems with a polynomial cost associated to each transition (bytecode operation). However, they might lead to useful gas bounds for smart contracts, possibly by using a different formalism. Unfortunately, a direct experimental comparison was only possible with GASTAP, since the other tools mentioned above work on different programming languages.

Scilla [Sergey et al. 2019]. Scilla is a functional programming language designed to write safe smart contracts. It is also much more verification-friendly than both Solidity and the EVM. One of the main design choices in Scilla is to ensure predictable and easy-to-compute gas consumption for every function. Intuitively, this is achieved by abandoning recursion and loops in favor of structural folds. Additionally, the semantics of Scilla are designed such that one can easily and modularly add a monadic gas accounting.

TiML [Wang et al. 2019a]. Another approach to prevention of gas-based vulnerabilities is provided by the TiML functional programming language and its Ethereum Virtual Machine variant TiEVM. The idea is to encode resource bounds in the type system, ensuring that a smart contract that passes type-checking would never run out of gas. In contrast to our approach, TiML considers a functional, rather than imperative, view of the smart contracts. Moreover, it does not synthesize bounds, but rather verifies the correctness of user-provided gas bounds encoded in its type system.

7 CONCLUSION

In this work, we presented Asparagus, a novel approach to automatically synthesize parametric gas usage upper-bounds for smart contracts, using tools and theorems from polyhedral and real algebraic geometry. To the best of our knowledge, this is the first identified use-case of algebro-geometric methods in the blockchain. Moreover, our approach is the first to successfully synthesize polynomial bounds. Finally, our experimental results demonstrate the applicability of our approach to a large set of standard real-world smart contracts which are currently deployed on the Ethereum blockchain.

8 ACKNOWLEDGMENTS AND NOTES

The authors are grateful to the anonymous reviewers for detailed comments which significantly improved this work. The research was partially supported by Hong Kong Research Grants Council ECS Project 26208122. Z. Cai was supported by the Hong Kong PhD Fellowship Scheme (HKPFS). Following the norms of theoretical computer science, authors are listed in alphabetical order.

REFERENCES

Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, German Puebla, Diana V. Ramírez-Deantes, Guillermo Román-Díez, and Damiano Zanardini. 2009. Termination and Cost Analysis with COSTA and its User Interfaces. In *PROLE*. 109–121.

Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2020. GASOL: Gas analysis and optimization for Ethereum smart contracts. In *TACAS (2)*. 118–125.

Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2021. Don't run on fumes - Parametric gas bounds for smart contracts. *J. Syst. Softw.* 176 (2021), 110923.

Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. 2018. EthIR: A framework for high-level analysis of Ethereum bytecode. In *ATVA*. 513–520.

Ali Asadi, Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Mohammad Mahdavi. 2021. Polynomial reachability witnesses via Stellensätze. In *PLDI*. ACM, 772–787.

Robert Atkey. 2011. Amortised resource analysis with separation logic. *Log. Methods Comput. Sci.* 7, 2 (2011).

Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on Ethereum smart contracts. In *POST*. 164–186.

Grigoriy Blekherman, Pablo A Parrilo, and Rekha R Thomas. 2012. *Semidefinite optimization and convex algebraic geometry*. SIAM.

Stefan Bygde. 2010. *Static WCET analysis based on abstract interpretation and counting of elements*. Ph. D. Dissertation. Mälardalen University.

Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *CAV*, Vol. 9779. 3–22.

Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2017a. Non-polynomial Worst-Case Analysis of Recursive Programs. In *CAV*, Vol. 10427. 41–63.

Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2019. Non-polynomial Worst-Case Analysis of Recursive Programs. *ACM Trans. Program. Lang. Syst.* 41, 4 (2019), 20:1–20:52.

Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. 2020. Polynomial invariant generation for non-deterministic recursive programs. In *PLDI*. 672–687.

Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Nastaran Okati. 2018. Computational Approaches for Stochastic Shortest Path on Succinct MDPs. In *IJCAI*. 4700–4707.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Tobias Meggendorfer, and Dorde Zikelic. 2022. Sound and Complete Certificates for Quantitative Termination Analysis of Probabilistic Programs. In *CAV*, Vol. 13371. 55–78.

Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, Jiri Zárevúcky, and Dorde Zikelic. 2021b. On Lexicographic Proof Rules for Probabilistic Termination. In *FM*, Vol. 13047. 619–639.

Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, Jiri Zárevúcky, and Dorde Zikelic. 2023. On Lexicographic Proof Rules for Probabilistic Termination. *Formal Aspects Comput.* 35, 2 (2023), 11:1–11:25.

Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, and Dorde Zikelic. 2021a. Proving non-termination by program reversal. In *PLDI*. 1033–1048.

Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. 2017b. Stochastic invariants for probabilistic termination. In *POPL*. 145–160.

Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiapu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. 2021. GasChecker: Scalable analysis for discovering gas-inefficient smart contracts. *IEEE Trans. Emerg. Top. Comput.* 9, 3 (2021), 1433–1448.

Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational cost analysis. In *POPL*. 316–329.

Ezgi Çiçek, Mehdi Bouaziz, Sungkeun Cho, and Dino Distefano. 2020. Static resource analysis at scale. In *SAS*. 3–6.

Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT solver. In *TACAS*. 93–107.

Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. 2003. Linear invariant generation using non-linear constraint solving. In *CAV*. 420–432.

Jesús Correas, Pablo Gordillo, and Guillermo Román-Díez. 2021. Static profiling and optimization of Ethereum smart contracts using resource analysis. *IEEE Access* 9 (2021), 25495–25507.

Micah Dameron. 2019. Beige paper: An Ethereum technical specification. (2019).

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An efficient SMT solver. In *TACAS*. 337–340.

Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. 2015. Proofs of space. In *CRYPTO*. 585–605.

Ethereum Foundation. 2014. https://docs.soliditylang.org/

Julius Farkas. 1902. Theory of simple inequalities. *Journal for pure and applied mathematics (Crelles Journal)* 1902, 124 (1902), 1–27.

Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A static analysis framework for smart contracts. In *WETSEB*. 8–15.

Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *SOSP*. 51–68.

Amir Goharshady. 2020. *Parameterized and algebro-geometric advances in static program analysis*. Ph. D. Dissertation. Institute of Science and Technology Austria.

Amir Kafshdar Goharshady, S. Hitarth, Fatemeh Mohammadi, and Harshit J. Motwani. 2023. Algebro-geometric Algorithms for Template-Based Synthesis of Polynomial Programs. In *OOPSLA*. 727–756.

Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 116:1–116:27.

David Handelman. 1988. Representing polynomials by positive linear functions on compact convex polyhedra. *Pacific J. Math.* 132, 1 (1988), 35–62.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.* 34, 3 (2012), 14:1–14:62.

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *POPL*. 359–373.

Mingzhang Huang, Hongfei Fu, and Krishnendu Chatterjee. 2018. New Approaches for Almost-Sure Termination of Probabilistic Programs. In *APLAS*, Vol. 11275. 181–201.

Mingzhang Huang, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. 2019. Modular verification for almost-sure termination of probabilistic programs. In *OOPSLA*, Vol. 3. 129:1–129:29.

Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing safety of smart contracts. In *NDSS*.

Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *CRYPTO*. 357–388.

Chao Liu, Jianbo Gao, Yue Li, and Zhong Chen. 2019. Understanding out of gas exceptions on Ethereum. In *BlockSys*, Vol. 1156.

Hongming Liu, Hongfei Fu, Zhiyong Yu, Jiaxin Song, and Guoqiang Li. 2022. Scalable linear invariant generation with Farkas' lemma. In *OOPSLA*, Vol. 6. 204–232.

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *CCS*. 254–269.

Fuchen Ma, Meng Ren, Fu Ying, Wanting Sun, Houbing Song, Heyuan Shi, Yu Jiang, and Huizhong Li. 2022. V-Gas: Generating high gas consumption inputs to avoid out-of-gas vulnerability. *ACM Trans. Internet Technol.* (2022).

Matteo Marescotti, Martin Blicha, Antti E. J. Hyvärinen, Sepideh Asadi, and Natasha Sharygina. 2018. Computing exact worst-case gas consumption for smart contracts. In *ISoLA (4)*. 450–465.

Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

Behkish Nassirzadeh, Huaiying Sun, Sebastian Banescu, and Vijay Ganesh. 2021. Gas gauge: A security analysis tool for smart contract out-of-gas vulnerabilities. *CoRR* abs/2112.14771 (2021).

Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: an efficient adaptive fuzzer for solidity smart contracts. In *ICSE*. 778–788.

Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *ACSAC*. 653–663.

Sunoo Park, Albert Kwon, Georg Fuchsbauer, Peter Gazi, Joël Alwen, and Krzysztof Pietrzak. 2018. SpaceMint: A cryptocurrency based on proofs of space. In *FC*. 480–499.

Andreas Podelski and Andrey Rybalchenko. 2004. A complete method for the synthesis of linear ranking functions. In *VMCAI*. 239–251.

David Prechtel, Tobias Groß, and Tilo Müller. 2019. Evaluating spread of 'gasless send' in Ethereum smart contracts. In *NTMS*. 1–6.

Ishaani Priyadarshini. 2019. Introduction to blockchain technology. *Cyber security in parallel and distributed computing: concepts, techniques, applications and case studies* (2019), 91–107.

Mihai Putinar. 1993. Positive polynomials on compact semi-algebraic sets. *Indiana University Mathematics Journal* 42, 3 (1993), 969–984.

Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *NDSS*.

Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2004. Constraint-based linear-relations analysis. In *SAS*, Vol. 3148. 53–68.

Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. 2020. Smart contract: Attacks and protections. *IEEE Access* 8 (2020), 24416–24427.

Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer smart contract programming with Scilla. In *OOPSLA*. 1–30.

David Siegel. 2016. Understanding The DAO Attack. https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/

Bikramaditya Singhal, Gautam Dhameja, Priyansu Sekhar Panda, Bikramaditya Singhal, Gautam Dhameja, and Priyansu Sekhar Panda. 2018. Introduction to blockchain. *Beginning Blockchain: A Beginner's Guide to Building Blockchain Solutions* (2018), 1–29.

Yican Sun, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. 2023. Automated Tail Bound Analysis for Probabilistic Recurrence Relations. In *CAV*, Vol. 13966. 16–39.

Jinyi Wang, Yican Sun, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. 2021. Quantitative analysis of assertion violations in probabilistic programs. In *PLDI*. 1171–1186.

Peng Wang et al. 2019a. *Type system for resource bounds with type-preserving compilation.* Ph. D. Dissertation. Massachusetts Institute of Technology.

Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin, and Wenjun Shi. 2019b. Cost analysis of nondeterministic probabilistic programs. In *PLDI*. 204–220.

David S Watkins. 2004. *Fundamentals of matrix computations*. Vol. 64. John Wiley & Sons.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.* 7, 3 (2008), 36:1–36:53.

Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* (2014), 1–32.

Received 2023-04-14; accepted 2023-08-27