Smart Contracts Refinement for Gas Optimization

Keerthi Nelaturu*, Sidi Mohamed Beillahi†, Fan Long‡, Andreas Veneris*‡
* Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, Canada
† Université de Paris, IRIF, CNRS, Paris, France
‡ Dept. of Computer Science, University of Toronto, Toronto, Canada

***Abstract*—Smart contracts facilitate the execution of programmable code on a blockchain. The cost for executing smart contract code is metered using gas – the exact amount of which is based on the computational complexity of the underlying smart contract. Hence, it is imperative to optimize smart contract code to reduce gas consumption and, in some instances, to even avoid malicious attacks. In this paper, we propose an approach to optimize the gas consumption of smart contracts, specifically loop control structures. We present a prototype implementation of our approach using off-the-shelf tools for Solidity smart contracts. We experimentally evaluate our technique using 72 Solidity smart contracts. Our evaluation demonstrates the average gas cost savings per transaction to be around $23,943$ gas units, or an equivalent 21% decrease in gas costs. Although the approach causes a slight increase in deployment costs due to the additional internal functions, this is only $16,710$ gas units on the average, or a 5% of the total deployment cost. As this overhead remains quite reasonable when compared to the gas cost savings for each transaction, it also confirms the efficacy, practicality and effectiveness of the proposed methodology.**

***Index Terms*—Smart contracts, Gas, Verification, Synthesis**

I. Introduction

Blockchain offers an innovative approach that allows to establish trust in an open environment without the need of a centralized authority to do so. This is because no entity can delete or modify blockchain transactions once they have been recorded. Many consider blockchain as a breakthrough application of cryptography and distributed systems, with use cases ranging from globally deployed cryptocurrencies [1], [2], to Central Bank Digital Currencies [3], supply chains [4], Internet-of-Things networks [5] and insurance [6].

Popular blockchains, most prominently Ethereum [7], allow the execution of application programs, called *smart contracts*, that are stored on the blockchain. Smart contracts offer the autonomy for arbitrarily-complex transactions between untrusted parties in a secure manner without going through a middleman (*e.g.,* commercial financial institutions) using cryptocurrencies. They are already powering a sizable economy: applications include decentralized finance [8] and auctions [9]. A smart contract manages a permanent state stored on the blockchain. It is constituted of a set of functions that manipulate the state. Functions can be called either directly by users or indirectly by other smart contracts, through transactions. They allow to perform arbitrarily-complex operations using cryptoassets stored on the blockchain. Solidity [10] is the most popular Turing-complete high-level programming language for smart contracts, which is designed to target the Ethereum Virtual Machine (EVM) [7]. A smart contract written in Solidity resembles an object in a standard object-oriented programming language, *e.g.,* Java.

An important aspect in smart contract operation is this of *gas*, that is, their underlying computational resource that compensates system participants for executing transactions. A transaction can be terminated if the amount of gas it has consumed exceeds a certain limit fixed by the author of the transaction. The amount of gas for a particular transaction invoking a smart contract function depends on the number of operations in the function and their types. For instance, read and write operations on storage variables consumes much greater amounts of gas than read and write operations for local variables. In a nutshell, the total transaction fee is computed by multiplying the total amount gas by the gas price[1]. Thus, the more computation a smart contract function performs, the more gas and fees it needs to pay for its execution. Therefore, executing smart contracts with functions containing inefficient or unnecessary operations can become a costly expedition. Furthermore, these contracts can be targeted by malicious attackers causing financial losses by exploiting gas related vulnerabilities [11], [12]. A recent study on smart contracts on the Ethereum blockchain reported gas related vulnerabilities in contracts worth more than $2 billion USD [13].

Optimizing smart contracts may reduce gas consumption and therefore the associated cost of executing transactions. It also has the potential to mitigate against malicious exploitation of smart contracts. A common optimization goal for a smart contract aims to reduce the number of operations in general, and in particular, those accessing storage variables while ensuring functional correctness of the underlying code. More specifically, loops can potentially cause an exponential increase in gas consumption that relates to the number of iterations a loop(s) is executed. Evidently, with an increasing use of smart contracts to perform complex operations, the usage of loops in those programs is likely to increase as well. Therefore, it remains objectively important to develop techniques to optimize loop structures in smart contracts.

In this paper, we propose an approach to optimize loop structures in smart contracts. Our optimization consists of a synthesis based technique for generating summaries of loops using predefined templates such as the standard map-reduce operators. Recent work showed that most smart contracts loops can be summarized using map-reduce type operators [14],

[1]Gas price varies and is set by the author of a transaction depending on the current ongoing gas price being used in the blockchain network.

[15]. In this paper, we apply syntactic transformations on the generated summaries to use local variable instead of repeatedly accessing storage variables. Doing syntactic transformations on summaries is much simpler than doing them on the original loops since summaries follow predefined templates and are more compact. As an added advantage, we also verify the equivalence between the generated contract with the transformed summaries of loops and the original one. To prove the equivalence, we adopt the notion of behavioral refinement between smart contracts from [16] which relates the input-output behavior of contracts' transactions, *i.e.,* transactions parameters and effects on storage variables, ignoring internal details such as local operations and control flow.

We implement our approach by leveraging the synthesizer from [14] to generate summaries of loops. We then extend the synthesizer to perform the syntactic transformations on the generated summaries. Finally, we verify the equivalence between contracts using `solc-verify` [17]. All in all, the proposed implementation correctly optimizes the code of nontrivial smart contracts, and integrates off-the-shelf tools to optimize loop structures in Solidity smart contracts. In detail, we empirically validate our approach on 72 Solidity smart contracts. The experiments demonstrate a 21% decrease in gas costs by applying the proposed optimization approach.

In summary, this paper makes the following contributions:

- We develop an approach to optimize smart contracts gas consumption that combines loop summaries synthesis, syntactic transformations, and equivalence proofs;
- We build an implementation of our approach for Solidity smart contracts that integrates off-the-shelf tools: `SOLIS` [14] and `solc-verify` [17]; and,
- We evaluate our approach, optimizing loop structures for 72 Solidity smart contracts which results in 21% decrease in gas costs.

The rest of the paper is organized as follows. In Section II, we give a motivational overview of our methodology. Section III presents the prior art. In Section IV, we detail the technical elements of our methodology. Section V presents the implementation of our approach for Solidity smart contracts while Section VI contains the empirical results. Finally, Section VII concludes this work.

II. Problem & Method Overview

Here we give an overview of the gas optimization problem investigated in this paper. First, we discuss three refactoring patterns that can be applied to loops to optimize their gas consumption. Then, we illustrate with an example our approach for optimizing smart contracts using the above refactoring patterns while ensuring that the resulting optimized smart contracts are behaviorally equivalent to the original contracts.

Below we list the three loops refactoring patterns which we target in our work:

1) *Loops with repeated storage calls:* A loop can make many calls to the storage based on the number of steps it is supposed to make. This is because read/write calls to the storage variables requires high amounts of gas. For instance, in each iteration of the loop in Figure 1, we require a storage read for loading $total$ value and a storage write for updating $total$ value after the addition operation. A possible optimization, shown in Figure 2, consists of using a local variable to store the $total$ value, and all updates are made to the local variable before re-assigning it to $total$ at the end of the loop. Thus, we reduced the repeated storage calls to just two storage calls.
2) *Loops with a constant comparison:* This pattern refers to the case where there is a comparison operation within a loop that always evaluate to a constant value. For instance, Figure 3 shows an if condition that always evaluate to True. An optimized version consists of eliminating the if condition is shown in Figure 4.
3) *Loops with repeated computations:* Similar to the first pattern 1, but this pattern involves computation that is repeated in each iteration of the loop and could be replaced by a constant value computed once before entering the loop. For instance, the computation $limit * price$ in Figure 5 can be done outside the loop and assigned to a local variable to optimize the gas usage as shown in Figure 6.

```
for (uint i=0;i<length;i++) {
    total += tokens[i];
}
```

Fig. 1. Repeated storage calls

```
uint local = total;
for (uint i=0;i<length;i++) {
    local += tokens[i];
}
total = local;
```

Fig. 2. Refactored loop

```
uint x = 1;
for (uint i=0;i<length;i++) {
    if(x + i > 0) {
        total += tokens[i];
    }
}
```

Fig. 3. Constant comparison

```
uint x = 1;
for (uint i=0;i<length;i++) {
    total += tokens[i];
}
```

Fig. 4. Refactored loop

```
for (uint i=0;i<length;i++) {
  tokens[i] += limit × price;
}
```

Fig. 5. Repeated computations

```
uint local = limit × price;
for (uint i=0;i<length;i++) {
  tokens[i] += local;
}
```

Fig. 6. Refactored loop

A. A Motivating Example

We illustrate our approach using the smart contract in Figure 7. The contract code consists of a single function with a gas-inefficient loop because of unnecessary repeated storage calls to read the value of `active`. Our approach synthesizes a loop summary function to replace the gas-inefficient loop such that the function parameters are the loops parameters and the assigned value as demonstrated by the function `foo_for` in Figure 8. Following, we apply

```
contract C {
    mapping(address => bool) whitelist;
    address[] beneficiaries;
    bool active;

    function foo() public {
        for (uint i=0; i<beneficiaries.length; i++) {
            whitelist[beneficiaries[i]] = active;
        }
    }
}
```

Fig. 7. Unoptimized contract

syntactic transformations that add local assignments operations to copy the values from storage variables (*e.g.*,, `active`) to local variables (*e.g.,* `rvariable`) that are passed in the invocation to the loop summary function `foo_for` as shown in Figure 8. Thus, the intermediary function `foo_for` does all the computations (previously done on storage variable) on the newly passed local variable. Our approach follows a template-based synthesis one which allows to optimize the loop body by using the more compact template summary. Thus, it facilitates applying the syntactic transformations by restricting them to local assignments operations to copy the input and output parameters of the loop summary function.

In the final step of our technique, we check whether the optimized contract is behaviorally equivalent to the original contract. More specifically, our behavioral (*i.e.,* functional) equivalence consists of a relation between the states of the two contracts supporting a proof that the optimized contract mimics every method invocation of the original contract. Since in our optimization we do not alter the state representation of the original contract than the relation supporting the proof of equivalence is always a conjunction of equalities between respective state variables as shown below:

$$\mathsf{Equiv} \stackrel{\text{def}}{=} \texttt{C0.beneficiaries} = \texttt{C.beneficiaries} \wedge \texttt{C0.active} = \texttt{C.active} \wedge \texttt{C0.whitelist} = \texttt{C.whitelist} \quad (1)$$

To prove the equivalence, we construct a product contract in Figure 9 from the two contracts, where for each two public methods (*e.g.,* `foo`) of the two contracts we create a corresponding public method in the product contract. Then, an equivalence relation proof is valid iff it is a valid invariant of the product contract. Here, we assume that contracts are deterministic – a valid assumption by-definition for contemporary blockchain smart contracts. In Figure 9, we define the synchronized product of the two contracts in Figures 7 and 8 using the inheritance mechanism of Solidity.

III. Prior Art

There are many works on the analysis or functional verification of smart contracts, *e.g.*, systems for detecting vulnerabilities in smart contracts based on static analysis and symbolic execution engines [13], [18]–[20], or systems for proving full functional correctness of smart contracts using techniques such as interactive theorem provers, SMT solvers, or predicate

```
contract C0 {
    mapping(address => bool) whitelist;
    address[] beneficiaries;
    bool active;

    function foo() public {
        bool rvariable = active;
        uint initial = 0;
        uint loopcondition = beneficiaries.length;
        foo_for(rvariable, initial, loopcondition);
    }

    function foo_for(bool rvariable, uint initial,
        uint loopcondition) internal {
        for (uint i=initial; i<loopcondition; i++) {
            whitelist[beneficiaries[i]] = rvariable;
        }
    }
}
```

Fig. 8. Optimized contract

```
invariant C0.beneficiaries == C.beneficiaries
invariant C0.active == C.active
invariant C0.whitelist == C.whitelist
contract P is C, C0 {

    constructor() public {
        C();
        C0();
    }

    function checkFoo() public {
        C.foo();
        C0.foo();
    }
}
```

Fig. 9. Product contract

abstraction [21]–[24]. Recently, the high volatility in gas prices has pushed for gas optimization techniques [13], [25]–[29]. Along those lines, this paper focuses on developing a novel gas optimization approach for smart contracts written in a high-level programming language. For our methodology, we use existing techniques and tools for smart contracts synthesis and verification [14], [16], [17]. In the remaining we review techniques that optimize smart contract gas consumption.

There are many tools for detecting gas-inefficient patterns and gas related vulnerabilities in smart contracts [13], [25]–[34]. In [13], the authors develop a tool that detects gas-related vulnerabilities. However, they do not propose any automated optimization to eliminate these vulnerabilities. Cheng et al.'s [25] identify multiple patterns such as dead code, opaque predicates, expensive operations in loops that can be optimized. They develop a tool called `GASPER` to apply the optimization to smart contracts at the bytecode level. They also extend their initial work to create GasReducer [35] that uses parallelized symbolic execution and handles additional optimization patterns. Albert et al. [26] develop a tool called `GASOL` to analyze gas consumption and to optimize smart contract code by calculating an upper bound of gas consumption. Their optimizations are specific to storage-related operations. In [29], [34], the authors target Solidity smart contracts and propose static analysis techniques based on syntactic transfor-

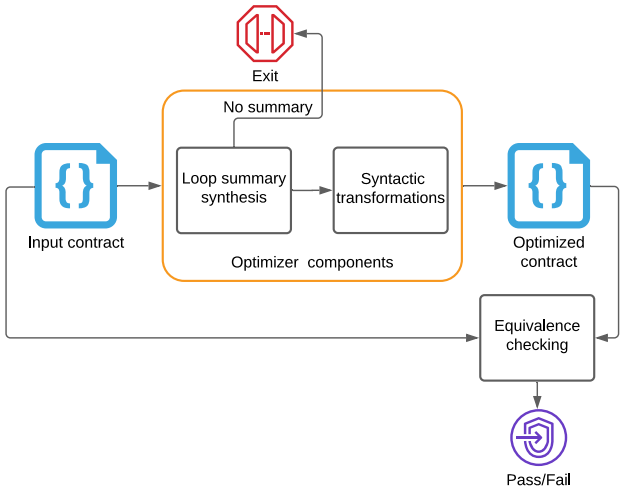

Fig. 10. Optimizer Components

mations that follow a predefined set of optimization patterns similar to the ones we discussed in Section II. For instance, in [29] a Python based optimizer is developed to detect the predefined set of gas-inefficient patterns and replace them with optimized code. Both [29], [34] do not formally proof equivalence between the optimized contract and the original contract. In contrast, our approach includes an equivalence checking step to ensure equivalence between the two contracts using simulation relation based proof. Furthermore, note that techniques based on static analysis are less precise (*i.e.*, in terms of the number of false positives) than our synthesis based technique. This is because they use syntactic approximation of the variables accessed and bounded loop unrolling. Thus, in a bounded loop unrolling it might seem that a block of code is unreachable while this might not be valid in general.

IV. Proposed Methodology

We now describe our methodology for gas optimization. Given a smart contract, our approach generates an optimized contract that has the same state representations (storage variables remain unchanged) as the original contract and is behaviorally equivalent to the original contract as well. In Figure 10, we show the three main steps that constitute our methodology for gas optimization. The first step constitutes of synthesizing a compact summary for each loop using an approximate synthesis technique based on predefined templates like map-reduce operators. Our loop summarization allows one to optimize code and it also reduces gas consumption. For instance, it allows to remove unnecessary conditionals according to the second pattern discussed in Section II. Then, the second step constitutes of syntactic transformations where we remove unnecessary accesses to storage variables, which allow to reduce the gas consumption further. Our final step consists of checking the behavioral equivalence between the input contract and the optimized contract generated using the previous two steps. In the rest of the section, we describe the steps of our methodology in more details.

A. Loop Summary Synthesis

To synthesize a concise summary of a loop structure we adopt a template-based synthesis technique. In particular, we use the set of templates proposed by [14] which consists of map-reduce operators such as `Map`, `Fold`, and `Zip`. This is because it was shown in [14] that most smart contracts loops can be summarized using the above set of templates. For instance, the loop in the contract shown in Figure 7 can be summarized by a map operator function that assigns a constant value (`active`) to a range of elements within a Solidity map (`whitelist`). The function `foo_for` in Figure 8 corresponds to the synthesized map operator function parameterized by the constant value and the range of elements. Using our template-based synthesis technique allows one to derive a loop summary where dead code (*i.e.*, code that is never executed) within a loop body and unnecessary conditionals are eliminated. Thus, we are able to carry gas optimization during the synthesis step of our methodology.

B. Syntactic Transformations

After the previous semantics transformations based on synthesis, in this step we use syntactic transformations to apply further optimizations. In particular, our syntactic transformations are guided by the goal to eliminate the first and third gas-inefficient patterns that we discussed in Section II. These patterns mainly target the reduction of gas consumption associated with the usage of storage variables. Having a concise loop summary generated in the previous step will facilitate applying the syntactic transformations. Our syntactic transformations consist of mainly eliminating repetitive calls to a contract storage variables by storing copies of storage variables in local variables. Following, the values stored in local variables are moved back to the corresponding storage variables. For instance, in Figure 8 the value of the storage variable `active` is stored in a local variable `rvariable` that is passed as parameter to the map operator function `foo_for`. Note that if we consider the number of iterations a loop will go through, eliminating repetitive calls to storage variables allows to save substantial amounts of gas cost that can reach more than 20,000 gas units.

Figure 8 shows the optimized contract code after applying the above two steps of our gas optimization methodology. Next, we describe how to check whether this optimized contract is behaviorally equivalent to the original contract.

C. Equivalence Checking

We reduce the problem of verifying that the optimized contract is behaviorally equivalent to the original contract to proving bisimulation relation between the two contracts. The bisimulation relation consists of the conjunction of equality between corresponding state variables of the two contracts. This is because in the previous optimization steps we do not alter the state variables of contracts. Then, we prove

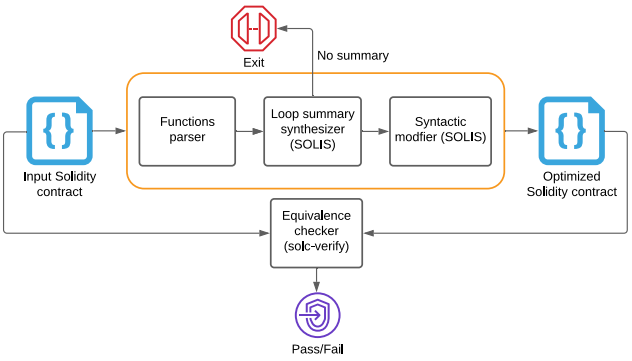

Fig. 11. Implementation Architecture

```
contract C {
    uint initial;
    uint loopcondition;
    uint gvariable;
    uint[] somearray;
    uint somecondition;

    function foo() public {
        for (uint i= initial ; i< loopcondition ; i++) {
            uint local = 100;
            gvariable += somearray[i] + local ;
        }
    }
}
```

Fig. 12. Loop locations to be optimized

the validity of the bisimulation relation between the two contracts by checking that it is an inductive invariant for a composition of contracts, which is formalized using the standard synchronized product construction. For instance, Figure 9 contains the product contract `P` of the two contracts `C` and `C0` discussed in Section II, which is defined using the inheritance mechanism of Solidity. Since the public method `foo` is defined in both contracts, the method `checkFoo` represents synchronous invocations of `foo` in `C` and `C0`. With this verification step, we will be able to certify the behavioral integrity of the accomplished optimizations in the previous two steps.

V. Implementation

In this section we describe an implementation of our proposed methodology for Solidity smart contracts. Figure 11 shows the high-level architecture of our implementation which consists of four main components: functions *parser*, a loop summary *synthesizer*, a syntactic *modifier*, and an equivalence *checker*. As input, our implementation requires a Solidity smart contract.

Given an input contract, the functions parser constructs an intermediary contract for each function of the given contract that contains a loop. This is because the tool, `SOLIS`, used by our loop summary synthesizer accepts only Solidity contracts with a single function. In turn, for each intermediary contract, the synthesizer generates a loop summary using a predefined set of templates. Afterwards, the syntactic modifier transforms unnecessary storage accesses using intermediary local variables to store the values of storage variables. An optimized contract is constructed by combining the optimized intermediary contracts. Finally, the equivalence checker validates that the optimized contract is behaviorally equivalent to the input contract.

A. Functions Parser

Our functions parses a set of intermediary contracts from the input contract. Each intermediary contract is associated with one function from the input contract. We use Solidity Parser Python library [36] to extract the abstract syntax tree (AST) of the input contract. We then use the AST to identify functions with loops and generate an intermediary contract for each function.

B. Loop Summary Synthesizer

To synthesize a loop summary for an intermediary contract, we use `SOLIS` [14], an automated tool for synthesizing loop summaries for Solidity contracts. `SOLIS` generates a loop summary by enumerating candidate summaries in a fixed set of templates defined as in a domain specific language called `CONSUL`. Possible operations in `CONSUL` are `Map`, `Fold`, and `Zip` functions. `SOLIS` uses bounded model checking to ensure that a candidate loop summary is valid by checking whether it is equivalent to the initial loop. We extend `CONSUL` along three principle axes. First, we extend it to support arithmetic operations other than addition. Second, we add more Solidity language features such as *msg.sender*. Three, we extend it to handle functions and loops with complicated structures, *e.g.*, nested loops and loops inside `if` conditions.

C. Syntactic Modifier

For the syntactic transformations, we identify specific locations in the loop structure where we store the values of storage variables in local variables. At the end of the loop, the updated values of the local variables are moved back to the storage variables. For instance, in Figure 12, we highlight the particular locations where the above transformations can be applied: initialization, loop condition, variables accessed within the loop, and variables declared in the loop. We extend `SOLIS` to apply the syntactic transformations as postprocessing step for the intermediary contracts that contain synthesized loops summaries. Once the final Solidity optimized intermediary contracts are generated, we merge them to devise the final optimized contract.

D. Equivalence Checker

`SOLIS` automatically checks for equivalence between an intermediary contract and the generated intermediary contract before the syntactic transformations. However, it remains to check for equivalence between an intermediary contract and the generated intermediary contract after syntactic transformations, and the input contract and the final optimized contract.

Note that checking the equivalence between the input contract and the final optimized contract is sufficient.

Our equivalence checking consists of the reduction from bisimulation checking to deductive verification that the bisimulation relation between two contracts is a valid inductive invariant of the product of the two contracts. For this purpose, we use `solc-verify` [17], a modular verification tool for Solidity smart contracts. `solc-verify` reduces Solidity contract verification to Boogie verification which in turn reduced to SMT solving.

VI. Experiments

This section presents the results of an experimental evaluation of the prototype implementation of our approach. The workstation we use for our experiments is equipped with an Intel Core i3-4170 3.7GHz CPU, 16GB of DDR3 RAM, 512GB SSD running Linux Ubuntu 14.04LTS operating system in a local network environment.

We have three separate applications running for the experimental setup: (i) automated tool based on `SOLIS` to create the optimized contract given an input Solidity contract, (ii) Truffle [37] and Ganache [38] local network to evaluate gas costs by running minimal transactions on the input and output contracts, and (iii) `Solc-verify` to check the equivalence between the input and output contracts. The open-source code for the implementation using python is available at Github[2].

A. DataSet Collection

For our experiments, we collect a benchmark of smart contracts constituted of two data-sets. The first data-set is constituted of 22 contracts from the benchmark contracts used in [14]. The second data-set is based on the set of contracts used in [29], which we extract from Etherscan [39]. In total we extract around 72 contracts that are supported by our prototype containing functions with loop constructs. Apart from the restriction on loops, we also check for the Solidity version to be v0.5.0. The versions before had breaking changes that make `solc-verify` and `SOLIS` incompatible for compiling and analyzing the contracts.

B. Results

We run our tool with a benchmark of 72 Solidity smart contracts. In Table I, we report the results of the experimental evaluation for 23 contracts which have been selected based on a set of criteria. Contracts need to adhere to at least one of the following criteria below:

- contracts having more than one function;
- contracts containing maximum lines of code;
- contracts that account for the maximum gas cost savings; or,
- contracts with deployment costs of more than $20,000$ gas units.

The complete dataset with the cost savings and the increase in the deployment costs can be found on the Github repository

[2]https://github.com/nelaturuk/loop-optimizer

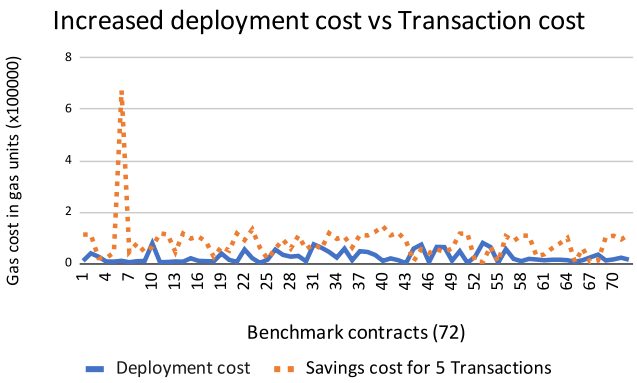

Fig. 13. Comparison between deployment cost and transaction cost

with the implementation. The first two columns of Table I list data concerning the smart contracts. The last four columns list data concerning the application of our tool to these contracts. We list the approximate gas cost savings. We calculate the deployment costs for the original and the optimized contracts. The invocation costs depict the transaction-related gas costs. In the case of arrays or mappings being used, we set the initial size to be 10 which is very low compared to the real-world applications. Depending on the complexity of the code, the average gas cost savings is $23,943$ gas units per transaction for the contracts that we tested. We also highlight the increased deployment cost due to the additional local variables and the internal functions. The average increase in deployment cost is around $16,710$ gas units. This cost will grow or fall depending on the contract's size and the complexity of the implementations. The deployment cost is a one-time fee. The transaction savings on the other hand are vital as end-users pay the gas fees for further access to the deployed contract. Figure 13 shows the comparison between one-time deployment cost and transaction costs aggregated for 5 transactions. The 72 benchmark contracts are represented on x-axis. The y-axis signifies the gas costs for both deployment and transactions.

As part of the evaluation, we verify all the 72 contracts in our data-sets for bisimulation equivalence using `solc-verify`. We are able to confirm the correspondence for around 50 contracts that did not include arithmetic calculations. A significant drawback observed with solc-verify was the semantic restrictions for predefined data types with loop constructs due to which we are not able to verify some of the contracts with arithmetic calculations.

C. Limitations

1) Program Synthesis: In the current setup, the set of templates supported by `CONSUL` is constituted of only `Map`, `Fold`, and `Zip` which restricts the number of contracts that can be optimized using our methodology. This limitation is one of the observations made during our evaluations. We also could not extract summaries for contracts with complex loop structures due to `CONSUL` implementation restrictions.

TABLE I
GAS COSTS FOR DEPLOYMENT AND TRANSACTIONS

Contract Name	Lines of Code	Number of Functions	Deployment Cost			Invocation Costs		
			Original	Optimized	Addional Cost	Original	Optimized	Savings
SimpleAddition	36	1	90543	100221	+9678	82874	60453	-22421
SimpleMath	60	4	151729	190711	+38982	82874	60453	-22421
IfCheck	33	1	116131	142021	+25890	39414	35273	-4141
MappingWithIf	51	1	147775	154849	+7074	43542	39401	-4141
DeclarationInFor	30	1	119139	126021	+6882	97344	88891	-8453
NestedFor	20	1	117545	127427	+9882	474666	250456	-224210
AddressList	27	1	188359	192697	+4338	66243	57436	-8807
Airdrop	37	1	359969	368363	+8394	73838	58339	-15499
ALinuxGold	61	2	439885	448730	+8845	85575	76659	-8916
Allocations	35	1	107523	185596	+78073	83182	70321	-12861
ArrayTools	22	1	95901	100233	+4332	83843	60453	-23390
AshToken	23	1	104547	109753	+5206	82904	60453	-22451
BeneficiaryOptions	66	2	404778	412140	+7362	40582	31689	-8893
DividendManager	58	2	262298	268797	+6499	64725	41511	-23214
EthGain	43	3	206846	226652	+19806	646727	627425	-19302
EticaRelease	140	2	1289607	1299321	+9714	71725	50872	-20853
Future	38	4	554900	563423	+8523	477235	460444	-16791
Etheropt	31	1	175540	183466	+7926	72811	67768	-5043
IPO	50	2	1566755	1604980	+38225	526418	513412	-13006
ISDT	30	1	531584	545125	+13541	78993	69100	-9893
KYCVerification	30	1	129519	135421	+5902	444590	421341	-23249
League	32	2	1360814	1414840	+54026	89369	71305	-18064
LitionRegistry	26	1	169316	189552	+20236	68061	41632	-26429
LIXToken	47	1	97879	99606	+1727	85527	73334	-12193

2) Equivalence Checker: For equivalence checking, we use `solc-verify` in our experiments. Except for arithmetic calculations within the loop, we confirmed equivalence for all the other structures. The issue with the arithmetic calculations arises because of the encoding of unsigned Solidity integers as mathematical integers in Boogie since they are well supported by SMT solvers. Due to this limitation, we were not able to check equivalence for certain contracts.

In summary, any limitations are centered around tools used by our implementation rather than inherent elements of the proposed methodology. However, with the increasing maturity of the various program analysis tools for smart contracts, one should expect future tools to address those limitations.

VII. CONCLUSION

A methodology for optimizing loop constructs in smart contracts in terms of the gas costs is presented. By generating summaries for loops using a DSL and syntactically modifying the contracts, we synthesize optimized contracts. We additionally verify the equivalence of the generated contracts with the original contracts using refinement proofs. We implement our methodology in a prototype tool and use this software to optimize loops in 72 Solidity smart contracts. Our evaluation shows that its optimizations produce significant gas cost savings. In the future we might extend our work to identify and gather more gas-efficient patterns for control structures other than loops for which we can use templates based synthesis to generate concise summaries. Furthermore, for verifying equivalence, our implementation assumes manually-provided invariants and pre/post-conditions, which can be automated for many contracts of interest using standard invariant-generation techniques, *e.g.*, [24].

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." 2008. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[2] V. Buterin, "What is ethereum?" *Ethereum Official webpage. Available: http://www.ethdocs.org/en/latest/introduction/what-is-ethereum.html*, 2016.

[3] N. Pocher and A. Veneris, "Privacy and transparency in cbdcs: A regulation-by-design aml/cft scheme," in *IEEE International Conference on Blockchain and Cryptocurrency (ICBC 2021)*, 2021.

[4] "Carrefour says blockchain tracking boosting sales of some products." 2019. [Online]. Available: https://www.reuters.com/article/us-carrefour-blockchain-idUSKCN1T42A5

[5] J. Meijers, G. D. Putra, G. Kotsialou, S. S. Kanhere, and A. Veneris, "Cost-effective blockchain-based iot data marketplaces with a credit invariant," in *EEE International Conference on Blockchain and Cryptocurrency (ICBC 2021)*, 2021.

[6] "Blockchain is empowering the future of insurance." 2016. [Online]. Available: https://techcrunch.com/2016/10/29/blockchain-is-empowering-the-future-of-insurance/

[7] Ethereum, 2021. [Online]. Available: https://ethereum.org

[8] "Decentralized finance (defi)." 2021. [Online]. Available: https://ethereum.org/en/defi/

[9] "Christie's auctions first digital-only artwork for $70m." 2021. [Online]. Available: https://www.theguardian.com/artanddesign/2021/mar/11/christies-first-digital-only-artwork-70m-nft-beeple.

[10] Solidity, 2021. [Online]. Available: https://docs.soliditylang.org

[11] "Swc-126: Insufficient gas griefing." 2021. [Online]. Available: https://swcregistry.io/docs/SWC-126

[12] "Swc-128: Dos with block gas limit." 2021. [Online]. Available: https://swcregistry.io/docs/SWC-128

[13] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: surviving out-of-gas conditions in ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 116:1–116:27, 2018. [Online]. Available: https://doi.org/10.1145/3276486

[14] B. Mariano, Y. Chen, Y. Feng, S. K. Lahiri, and I. Dillig, "Demystifying loops in smart contracts," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 262–274. [Online]. Available: https://doi.org/10.1145/3324884.3416626

[15] A. Li, J. A. Choi, and F. Long, "Securing smart contract with runtime validation," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 438–453. [Online]. Available: https://doi.org/10.1145/3385412.3385982

[16] S. M. Beillahi, G. F. Ciocarlie, M. Emmi, and C. Enea, "Behavioral simulation for smart contracts," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 470–486. [Online]. Available: https://doi.org/10.1145/3385412.3386022

[17] Á. Hajdu and D. Jovanovic, "solc-verify: A modular verifier for solidity smart contracts," in *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Chakraborty and J. A. Navas, Eds., vol. 12031. Springer, 2019, pp. 161–179. [Online]. Available: https://doi.org/10.1007/978-3-030-41600-3_11

[18] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 67–82. [Online]. Available: https://doi.org/10.1145/3243734.3243780

[19] J. He, M. Balunovic, N. Ambroladze, P. Tsankov, and M. T. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 531–548. [Online]. Available: https://doi.org/10.1145/3319535.3363230

[20] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 254–269. [Online]. Available: https://doi.org/10.1145/2976749.2978309

[21] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in isabelle/hol," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, J. Andronick and A. P. Felty, Eds. ACM, 2018, pp. 66–77. [Online]. Available: https://doi.org/10.1145/3167084

[22] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," in *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*, ser. Lecture Notes in Computer Science, M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. A. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, Eds., vol. 10323. Springer, 2017, pp. 520–535. [Online]. Available: https://doi.org/10.1007/978-3-319-70278-0_33

[23] Á. Hajdu and D. Jovanovic, "solc-verify: A modular verifier for solidity smart contracts," *CoRR*, vol. abs/1907.04262, 2019. [Online]. Available: http://arxiv.org/abs/1907.04262

[24] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. T. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1661–1677. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00024

[25] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, M. Pinzger, G. Bavota, and A. Marcus, Eds. IEEE Computer Society, 2017, pp. 442–446. [Online]. Available: https://doi.org/10.1109/SANER.2017.7884650

[26] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "GASOL: gas analysis and optimization for ethereum smart contracts," in *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, ser. Lecture Notes in Computer Science, A. Biere and D. Parker, Eds., vol. 12079. Springer, 2020, pp. 118–125. [Online]. Available: https://doi.org/10.1007/978-3-030-45237-7_7

[27] E. Albert, P. Gordillo, A. Rubio, and M. A. Schett, "Synthesis of super-optimized smart contracts using max-smt," in *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, ser. Lecture Notes in Computer Science, S. K. Lahiri and C. Wang, Eds., vol. 12224. Springer, 2020, pp. 177–200. [Online]. Available: https://doi.org/10.1007/978-3-030-53288-8_10

[28] J. Nagele and M. A. Schett, "Blockchain superoptimizer," *CoRR*, vol. abs/2005.05912, 2020. [Online]. Available: https://arxiv.org/abs/2005.05912

[29] T. Brandstätter, S. Schulte, J. Cito, and M. Borkowski, "Characterizing efficiency optimizations in solidity smart contracts," in *2020 IEEE International Conference on Blockchain, Blockchain 2020, Rhodes Island, Greece, November 2-6, 2020*. IEEE, 2020, pp. 281–290. [Online]. Available: https://doi.org/10.1109/Blockchain50366.2020.00042

[30] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang, "Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2020.

[31] J. Correas, P. Gordillo, and G. Román-Díez, "Static profiling and optimization of ethereum smart contracts using resource analysis," *IEEE Access*, vol. 9, pp. 25 495–25 507, 2021. [Online]. Available: https://doi.org/10.1109/ACCESS.2021.3057565

[32] V. Pérez, M. Klemen, P. López-García, J. F. Morales, and M. V. Hermenegildo, "Cost analysis of smart contracts via parametric resource analysis," in *Static Analysis - 27th International Symposium, SAS 2020, Virtual Event, November 18-20, 2020, Proceedings*, ser. Lecture Notes in Computer Science, D. Pichardie and M. Sighireanu, Eds., vol. 12389. Springer, 2020, pp. 7–31. [Online]. Available: https://doi.org/10.1007/978-3-030-65474-0_2

[33] A. A. ZARIR, G. A. OLIVA, Z. M. J. JIANG, and A. E. HASSAN, "Developing cost-effective blockchain-powered applications: A case study of the gas usage of smart contracts transactions in the ethereum blockchain platform," *ACM Trans. Softw. Eng. Methodol*, vol. 1, no. 1, 2020.

[34] C. Signer, "Gas cost analysis for ethereum smart contracts," Master's thesis, ETH Zurich, Department of Computer Science, 2018.

[35] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang, "Towards saving money in using smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, A. Zisman and S. Apel, Eds. ACM, 2018, pp. 81–84. [Online]. Available: https://doi.org/10.1145/3183399.3183420

[36] Consensys, "Python solidity parser," https://github.com/ConsenSys/python-solidity-parser, accessed on 03/09/2021.

[37] Truffle, 2021. [Online]. Available: https://www.trufflesuite.com/truffle

[38] Ganache, 2021. [Online]. Available: https://www.trufflesuite.com/ganache

[39] Etherscan, 2021. [Online]. Available: https://etherscan.io/