Characterizing Efficiency Optimizations in Solidity Smart Contracts

Tamara Brandstätter*, Stefan Schulte†, Jürgen Cito*, Michael Borkowski‡
*TU Wien, Austria, t.brandstaetter@dsg.tuwien.ac.at; juergen.cito@tuwien.ac.at
†TU Wien, Austria, s.schulte@dsg.tuwien.ac.at
‡Institute of Flight Guidance, German Aerospace Center (DLR), Braunschweig, Germany, michael.borkowski@dlr.de

***Abstract*—On many blockchain platforms, gas fees have to be paid for deploying and executing smart contracts. These fees depend on the size of the contract code as well as the needed computational steps and required storage space of a smart contract. Because of the large amount of gas cost paid each day, there is an inherent motivation to optimize smart contract code in order to reduce these cost.**

Within this paper, we discuss the application of 25 strategies for code optimization to Solidity smart contracts. A prototype is developed which detects potential optimizations and partially automatically optimizes the code accordingly. The optimization strategies are evaluated based on 3,018 verified open source smart contracts from *etherscan.io*. We find 471 rule violations in the test data spread across 204 different contract files.

***Index Terms*—Smart contracts, Code optimization, Solidity**

I. Introduction

Since the introduction of the basic principles of blockchains in 2008 [1], blockchain technologies have evolved into a major trend in both research and industry. Basically, a blockchain is an append-only distributed ledger which stores transactions that are propagated through a peer-to-peer (P2P) network. Transactions are grouped in blocks and the blocks are chained using cryptographic hash pointers [2], [3].

While first-generation blockchains like Bitcoin solely focus on providing the functionality for cryptocurrency transactions, second-generation blockchains like Ethereum introduce the possibility to write and deploy computer programs, which are executed by the nodes participating in the according blockchain [4]. These *smart contracts* are usually written in a high-level programming language like Solidity[1]. The source code is compiled into byte code and can then be deployed on the blockchain. The execution takes place in a dedicated environment, e.g., the Ethereum Virtual Machine (EVM) in the case of Ethereum [5]. In short, the EVM is a Turing-complete state machine with a stack-based architecture [6].

Turing-completeness poses some drawbacks when executing code in a blockchain network. For instance, by adding infinite loops to smart contracts, it would be possible to effectively diminish the available computational power. Hence, most second-generation blockchains apply a limit for the computational steps that can be performed within a smart contract [5]. This limit is set for each transaction leading to a smart contract invocation, individually by the sender of a transaction. The limit has an important implication, as it guarantees that each transaction will terminate at some point, i.e., at the latest when the upper limit of computational steps is reached.

While there are differences in the nomenclature and implementation of this limit, most second-generation blockchains follow the notions introduced by Ethereum. Since we make use of Ethereum and Solidity in the work at hand, we will also apply in the following the nomenclature defined by Ethereum.

In Ethereum, the unit for calculating cost is called *gas*. Every operation that can be carried out in the EVM has a defined amount of gas it consumes during deployment and invocations, ranging from 3 amounts of gas for the `ADD` operation to 32,000 amounts of gas for `CREATE`, which creates a new contract on the blockchain [5]. Storing data on the blockchain also is done through operations, and the according gas cost depend on the amount and the type of storage space needed (see Section III).

Importantly, the sender of a transaction does not only define the gas limit for a transaction. She also has to define how much Ether—i.e., the original cryptocurrency supported by Ethereum—she is willing to pay for each amount of gas. After a successful transaction, the sender pays the fee in Ether to the node (i.e., the miner) that executed the smart contract.

With the success of blockchains, the amount of gas paid for smart contracts grows accordingly. As an example, on February 21st 2020, approximately 121,800 US dollars have been spent on gas fees[2]. According to *etherscan.io*, the consumed gas cost per day were mostly in the range of 100,000 to 250,000 US dollars in 2019 and early 2020. At the end of July 2020 and in early August 2020, more than 1,000,000 US dollars were spent on gas fees per day.

Since a significant part of these cost are caused by smart contract code on the blockchain, there is a high potential to reduce these cost by optimizing the code of smart contracts. Therefore, we study the applicability of fundamental code optimization rules to Solidity-based smart contracts, with the goal of cost reduction. We implement promising optimization strategies (called *rules*) and provide a tool which is able to identify optimization potentials in source code, thus allowing developers to improve their smart contracts. To show that this can lead to gas fee savings, we implement automatic optimization for a subset of these rules.

[1] https://github.com/ethereum/solidity

[2] https://etherscan.io

Notably, we apply a defensive white-listing approach throughout our work. This means that only known patterns are identified when we apply rules. While this may lead to false negatives, i.e., rules are not applied even if there is potential for optimization, it also decreases the number of false positives.

The remainder of this paper is organized as follows: In Section II, we discuss the related work, while Section III introduces basic concepts necessary for the understanding of the presented solution. In Section IV, we analyze state-of-the-art optimization strategies and discuss to which degree it makes sense to apply them to Solidity-based smart contracts. Afterwards, we evaluate the implemented rules in Section V. Finally, we conclude the paper in Section VI.

II. Related Work

To the best of our knowledge, the number of approaches aiming at code optimization for smart contracts is still quite small. Notably, the related work discussed below has been conducted for Solidity-based smart contracts, as is also done within the work at hand.

In an early approach, Chen et al. describe the optimization of smart contracts on the source code level [7]. The authors identify seven anti-patterns, categorized into "useless-code-related" and "loop-related" anti-patterns, and provide the *GASPER* tool, which is able to identify three of these anti-patterns. Unfortunately, the GASPER tool is not publicly available so far [8]. In their follow-up work, Chen et al. focus on the identification of inefficient smart contract code on byte code level [9], [10]. Here, the authors present ten anti-patterns, which focus on loops, useless code, wasted disk space, and gas-inefficient operation sequences, implement them in the *GasReducer* tool, and evaluate the outcomes based on 1,500 Ethereum smart contracts. Notably, the authors apply an optimistic optimization approach, leading to a small yet still identifiable number of false positives of 2.5%. In our approach, we apply a white-listing approach for optimizations, which does not lead to false positives. While Chen et al. focus on the byte code level, we apply optimization on the source code level. Nevertheless, the contributions presented by Chen et al. come closest to the work at hand.

Nagele and Schett present the EVM byte code superoptimizer *ebso* which optimizes EVM byte code using constraint solving [6]. For this, ebso provides an encoding of the relevant information from the EVM (including its state), finds different target programs for the byte code, and then selects the target program with minimum gas cost. For this, single blocks of the byte code are taken into account, i.e., instructions which cover different blocks are not further regarded.

Albert et al. present *EthIR*, which allows to generate a control flow graph from the byte code of a smart contract [11]. The tool can help developers to gain a deeper understanding of smart contracts, but does not provide any identification of optimization potential. In subsequent work, the authors present *GASOL*, which is a tool for gas cost analysis and optimization [12]. The main functionality of GASOL is to calculate an upper bound of gas cost for a particular smart contract. Optimization is done solely with regard to storage-related gas cost (namely `SSTORE` and `SLOAD` – see Section III).

Feist et al. provide *Slither*, a tool which can be used to extensively analyze smart contracts [13]. Slither provides among other things the means for automated detection of code optimization opportunities and implements some basic optimization rules, i.e., that eligible variables are declared as constants and that some functions are defined as externals.

In general, work on analyzing algorithms and optimizing code with regard to efficiency is of course not a novel research direction, and has been a topic for at least 50 years [14], [15]. Notably, code optimization has been discussed with regard to different goals, e.g., energy efficiency [16] or reusability and flexibility [17]. However, as mentioned above, our work focuses on optimization with regard to gas fee reduction, since this is a primary goal for smart contracts.

Last but not least, there are a number of papers discussing the analysis of smart contract code with respect to other goals than cost efficiency. Here, the focus is mostly on detecting security vulnerabilities in smart contracts, e.g., [18]. An excellent overview of these approaches is given in [19].

III. Background: Smart Contracts

In Ethereum and similar blockchains, smart contracts are controlled by so-called contract accounts [20]. To invoke a smart contract, Ether has to be sent to the address of the contract account.

Apart from other data, the transaction also contains the sender-defined `STARTGAS` and `GASPRICE` fields. `STARTGAS` defines the upper limit of gas to perform the transaction, thus representing the limitation of computational steps and storage space mentioned in Section I. The `GASPRICE` defines the price the sender is willing to pay for each amount of gas needed to perform the transaction. The actual used gas for a smart contract invocation multiplied by the caller-defined `GASPRICE` gives the gas fee that has to be paid by the caller.

Notably, if a contract account invokes a smart contract, instead of a transaction, a so-called message call is sent by the contract account. The important difference between a transaction and a message call is that the latter does not contain the `GASPRICE`. Instead, the `GASPRICE` is defined by the origin transaction which led to the message call. For the purposes of the work at hand, we assume that smart contract invocations are always done through a transaction.

Before a smart contract can be invoked, it has to be deployed on the blockchain. After deployment, smart contracts in public blockchains like Ethereum are available for all network participants. Whenever a transaction invoking a contract is broadcast to the network, the deployed smart contract is executed by every validating node in the blockchain network. Since blockchains are append-only data structures, a deployed smart contract cannot be altered any longer [20]. This stresses the need to implement gas cost-efficient smart contracts.

When invoking a smart contract, gas cost occur. Every computational step costs some defined amount of gas. Reducing the steps therefore results in lower gas cost. In addition, the

TABLE I
EVM OPERATION GAS COST OVERVIEW (FROM: [5])

Name	Value	Operations/Transactions
G_{zero}	0	Paid for STOP, RETURN and REVERT.
$G_{jumpdest}$	1	Paid for JUMPDEST.
$G_{verylow}$	3	Paid for ADD, SUB, NOT, LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, BYTE, CALLDATALOAD, MLOAD, MSTORE, MSTORE8, PUSH*, DUP*, and SWAP*.
G_{memory}	3	Paid for every additional word when expanding memory.
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
G_{low}	5	Paid for MUL, DIV, SDIV, MOD, SMOD, and SIGNEXTEND.
G_{mid}	8	Paid for ADDMOD, MULMOD and JUMP.
$G_{txdatanonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
G_{sload}	200	Paid for SLOAD.
G_{call}	700	Paid for CALL.
G_{sreset}	5000	Paid for SSTORE when the storage value's zeroness remains unchanged or is set to zero.
$G_{selfdestruct}$	5000	Amount of gas to pay for SELFDESTRUCT.
G_{sset}	20000	Paid for SSTORE when the storage value is set to non-zero from zero.
$G_{transaction}$	21000	Paid as a basic fee for every transaction.
G_{create}	32000	Paid for CREATE.
$G_{txcreate}$	32000	Paid by all contract-creating transactions.
R_{sclear}	-15000	Refund given when the storage value is set to zero from non-zero.
$R_{selfdestruct}$	-24000	Refund given for self-destructing an account.

fees for deploying a smart contract depend on the length of the generated byte code. Thus, reducing the length of the byte code results in lower gas cost.

Another important cost factor is storage, since expanding the storage itself costs gas. Regarding storage, the EVM may utilize (i) the *stack*, which is used for executing a smart contract. The values on the stack are reset once execution of a smart contract is finished. (ii) *Memory*, which is an expandable word-addressable word array. Like the stack, the memory is volatile and therefore resets after the execution is finished. (iii) *Storage*, which provides long-term persistent storage, and is therefore the most expensive storage type [5].

Table I provides an overview of common EVM operations and their gas cost; a complete list can be found in [5]. In general, pure stack operations, e.g., ADD, are rather cheap, while storage- and contract-related operations are expensive. There are also two EVM operations which allow to refund gas, namely SCLEAR and SELFDESTRUCT.

When optimizing code, there might be a trade-off between cost for the deployment of smart contracts in the Ethereum blockchain, and each invocation of a smart contract in the EVM, i.e., the gas cost for deployment may increase because of the optimization, while the invocation cost decrease (or vice versa). We will discuss such trade-offs whenever necessary.

IV. Efficiency Optimization Rules

We analyze the applicability of classical code efficiency optimization strategies in the context of smart contracts. Specifically, we consider the efficiency rules postulated in early work by Bentley [15], which proposes strategies that are grouped into six categories (time-for-space rules, space-for-time rules, loop rules, logic rules, procedure rules, and expression rules).

We decided to analyze these standard rules for two major reasons. First, the application of these rules is supposed to be bound to local transformations that are almost independent from the underlying system specifics (however, we point out certain rules in which optimizations are highly context-specific for smart contracts). Second, given the simplistic nature of the cost model described above, analyzing the efficacy of rules geared towards general-purpose procedural languages, we can establish a baseline for more domain-specific optimizations.

Our first goal is to put these efficiency optimization rules into context for Solidity source code and Ethereum smart contracts. To that end, we provide a general description of each rule and how it could be applied to reduce gas cost, or whether applying the rule has no effect on gas cost or may even lead to cost increase. The second goal of our study is to mark rules with labels with respect to their efficacy for classifying and automatically fixing efficiency issues.

We present an analysis of 25 rules, organized in the six categories, to provide contextualization with respect to Solidity/Ethereum smart contracts and visibly label each rule with one of the following labels:

- **[Non-intrusive Fix]**: The rule can be automatically classified and a non-intrusive, context-independent fix exists. Non-intrusive fixes are changes that semantically preserve functionality without introducing increased code complexity or maintenance cost [21]. We implement these non-intrusive fixes as part of this work.
- **[Automated Rule Classification]**: The rule can be constructed as a context-independent automated classifier. We implement the classifier as part of this work and quantify the presence of these rules. However, we do not provide a non-intrusive (automated) fix.
- **[Context-dependent Rule]**: This label marks rules for which we cannot construct an automated, context-independent classifier, but which can theoretically lead to gas savings. We opt for a conservative definition here where we regard a rule as context-dependent if there is possibility for uncertainty in the classifier (due to context-specific properties). We acknowledge that more context- or domain-specific classifiers could potentially be constructed, but are beyond the scope of this work.
- **[No Gas Savings]**: The rule is not applicable to achieve gas savings.
- **[Included in Compiler]**: The rule is already implemented in the Solidity compiler `solc`[3] and will therefore not lead to further gas savings.

We perform initial analysis with simple code fragments addressing the rules to assess if cost benefits can be achieved during smart contract deployment and invocations. This additionally checks whether a particular rule is already imple-

[3]Version v0.15.3, which was the latest when conducting the work at hand.

mented in `solc`. We leverage the possibility to activate and deactivate compiler-internal optimization in `solc`, and run the original and optimized source code with activated and deactivated optimization.

A. Time-for-Space Rules

The basic idea of time-for-space rules is that space in terms of memory or storage is reduced, while the execution time is increased, since data is not directly available, and needs to be recalculated. In general, smart contracts deployed in the EVM can be assessed to be not very time-critical, since there is an inherent delay when invoking a smart contract [22]. This means that time-critical applications should rather not be deployed in the EVM in the first place. Therefore, adding some execution time should not be an issue in most applications.

*a) **[Context-dependent Rule]** Time-for-Space Rule 1: Packing:* Packing adopts dense storage representations. One approach to implement this is to apply overlaying, i.e., to use the same storage space to store data that is never needed at the same time. For instance, the same variable could be used for different things, if a variable is defined in a function and then never reused. In the EVM, defining less variables directly leads to decreased gas cost, since the operation for setting a storage value from zero to non-zero (`SSET`) costs 20,000 gas units, while the EVM operation for resetting a storage value to another non-zero value (`SRESET`) costs 5,000 gas units.

Hence, packing could be suitable to enable more cost-efficient smart contract code. However, it is not trivial to implement this rule defensively: The identification of packing possibilities generates a false positive whenever the usage of a variable is not found.

*b) **[Context-dependent Rule]** Time-for-Space Rule 2: Interpreters:* This rule aims at reducing the space required by using interpreters for compact representations. The simplest application of an interpreter are subroutines, i.e., part of a function is defined as a subroutine and can then be called.

Integrating subroutines may lead to decreased deployment cost, because the size of a smart contract is potentially decreased if subroutines are inserted into the code. However, the invocation cost may increase, since calling a subroutine is usually (a little bit) more expensive than calling code directly (see also Procedure Rule 1: Collapsing Procedure Hierarchies).

Identifying subroutines is closely related to "extract method" refactoring [23], and a complex task which provides a certain degree of uncertainty if a routine has really been identified or not. Hence, we decided not to further pursue this rule in the scope of this work.

B. Space-for-Time Rules

Space-for-time rules are based on the idea to store redundant information to decrease the runtime of a system. We regard space-for-time rules to not be generally applicable for reducing gas cost since storage is expensive in blockchains. This is the case for data structure augmentation and the storage of precomputed results. Nevertheless, there are two space-for-time rules which could be applicable to smart contracts when aiming at gas cost optimization:

*a) **[Included in Compiler]** Space-for-Time Rule 1: Caching:* The caching of the most frequently accessed data in the volatile memory (instead of the persistent storage) could lead to gas savings, since accessing the memory (`MLOAD`) is way cheaper than accessing the storage (`SLOAD`). Our analysis has shown that this optimization strategy has already been implemented in `solc`.

*b) **[Context-dependent Rule]** Space-for-Time Rule 2: Lazy Evaluation:* This rule aims at avoiding unnecessary evaluations, e.g., calculations or expression checks [24]. As a simplified example for memorization (which is a prerequisite for lazy evaluation), let us use the calculation of Fibonacci numbers. These numbers could either be calculated beforehand, which would be the above mentioned storage of precomputed results, or numbers are calculated when needed and stored at that point of time (e.g., in a lookup table), thus avoiding recalculations. We conducted an initial analysis to assess if this rule could lead to gas cost reduction. In this analysis, an additional mapping was introduced into a standard function, and filled and reused during function calls. This led to higher deployment cost, but lower invocation cost for the implemented smart contract.

In general, this rule could be applied if the number of invocations is high enough to justify the overhead of storing precomputed values in a lookup table, following the lazy evaluation principle. To apply this rule, expertise on the specific application is required, which means that the automatic classification and fixing according to this rule is difficult to achieve without introducing domain-specific heuristics.

C. Loop Rules

The loop rules discussed in the following paragraphs apply general best practices when programming loops. As will be shown in the following, *all* discussed loop rules are generally applicable to Solidity smart contracts.

*a) **[Non-intrusive Fix]** Loop Rule 1: Code Motion out of Loops:* This rule describes that repeated calculations that are inside a loop and do not depend on a loop variable can be moved outward of the loop. Thus, a calculation is performed only once instead of in each loop iteration. This may lead to additional deployment cost, if a new variable needs to be defined. However, the invocation cost for the optimized function is expected to be much cheaper. The only exception is if a loop has only one iteration. Therefore, calculations that do not depend on the loop variable should be moved outside the loop. This can always be applied to Solidity smart contracts.

*b) **[Automated Rule Classification]** Loop Rule 2: Combining Tests:* The goal of this rule is to decrease the number of tests for a loop. In the best case, only one test condition needs to be applied. As in Loop Rule 1, this may lead to higher deployment cost, while the invocation cost of a function are decreased. This rule is generally applicable with regard to the identification of potential combinations. However, automated optimization is context-dependent.

*c) **[Non-intrusive Fix]** Loop Rule 3: Loop Unrolling:* In some cases, it might be meaningful to remove small loops

to save the cost of modifying the loop variables and of checking the loop condition. Usually, this leads to higher deployment cost, while the invocation cost are reduced. This makes Loop Rule 3 generally eligible for optimizing Solidity smart contracts. It should be noted that automatic identification and subsequent optimization is not trivial if loops are nested or conditions are not automatically optimized. Therefore, in our implementation, we will differentiate between simple and complex loops (see Section V).

*d) **[Non-intrusive Fix]** Loop Rule 4: Transfer-driven Loop Unrolling:* This rule extends Loop Rule 3 by only externalizing trivial assignments made within a loop, i.e., the loop is still in the source code, but trivial assignments are moved outside the loop. For instance, this can be achieved by eliminating superfluous variables inside a loop. This should lead to savings in gas cost for both smart contract deployment and invocation, since storage and memory space are decreased.

*e) **[Non-intrusive Fix]** Loop Rule 5: Unconditional Branch Removing:* The basic idea of this rule is to remove unconditional branches at the end of a loop. Instead, the loop should be rotated in order to have a conditional branch at the end. For instance, this can be achieved by replacing a for-loop or while-loop with a do-while-loop. This removes a conditional jump at the beginning and an unconditional jump at the end of the loop. Instead, there is only a conditional jump at the end, thus saving gas cost.

*f) **[Automated Rule Classification]** Loop Rule 6: Loop Fusion:* Loop fusion combines multiple loops that apply to the same set of elements. This may lead to decreased deployment cost, since the smart contract code becomes shorter. Also, the invocation of a smart contract may become cheaper, since only one loop (instead of several loops) needs to be iterated. As with the other five loop rules, we therefore further regard this rule in our implementation (see Section V).

D. Logic Rules

This category of rules deals with logic evaluations that test the program state. The rules describe how code logic can be modified without semantic changes to increase cost efficiency.

*a) **[Non-intrusive Fix]** Logic Rule 1: Exploit Algebraic Identities:* The idea of this rule is to replace expensive expressions with semantically equivalent, yet cheaper, expressions. One example is the application of De Morgan's law, i.e., $\neg a \vee \neg b \equiv \neg(a \wedge b)$. By replacing the first term $\neg a \vee \neg b$ with the second term $\neg(a \wedge b)$, it is possible to get rid of a `NOT` operation in Solidity. Of course, there are further examples for exploiting algebraic identities.

The rule is generally applicable, however, it is not always trivial to identify these parts of optimization automatically, because they heavily depend on the use case. Still, easily applicable examples like De Morgan's law exist.

*b) **[Context-dependent Rule]** Logic Rule 2: Short-circuiting Monotone Functions:* This rule can be applied when a monotone function is tested for a threshold. The idea is to introduce a break into a function as soon as the result is known, avoiding excessive calculations. A typical example of the short-circuit evaluation is that instead of evaluating both expressions *A* and *B*, we only evaluate the first expression. So, if *A* is already false, *B* is not evaluated, because the overall expression cannot become true any longer. However, this cannot be applied if *B* has some important side effects.

Utilizing this rule may lead to cost reductions during smart contract invocations, since a contract can potentially avoid to be executed in its entirety. However, for more complex expressions that may introduce side-effects, it is difficult to construct an automated (general-purpose) classifier that avoids false positives.

*c) **[Context-dependent Rule]** Logic Rule 3: Reordering Tests:* Tests, i.e., conditions, can be arranged in different orders. This rule exploits this by rearranging tests so that "cheap" tests, i.e., tests most likely evaluated to *true*, are placed before expensive tests, which are more likely to be evaluated to *false*. For instance, this can be done in an if-then-else clause. Again, the problem is the necessity of estimating which condition is most likely evaluated to *true*, which requires domain expertise.

*d) **[No Gas Savings]** Logic Rule 4: Precompute Logical Functions:* The aim of this rule is to replace the calculation of a logical function by a lookup table. As discussed in Section IV-B, the issue is, however, that storage is expensive in the EVM, and it is therefore generally not meaningful (if the goal is to save gas) to use additional space in order to save time. Precomputation may save some gas in future invocations, but the additional deployment cost are usually too high and access to the precomputed results is still very expensive.

*e) **[Non-intrusive Fix]** Logic Rule 5: Boolean Variable Elimination:* The basic idea of this rule is to get rid of Boolean variables and to replace them with an if-then-else condition. For instance, we can replace a Boolean variable, which stores the result of a logical expression, with the actual logical expression, whenever the Boolean variable is originally used in a condition. This saves deployment cost, since the variable is not needed any longer, and may also lead to less cost during smart contract invocation.

E. Procedure Rules

This category of optimization strategies deals with the underlying structure of a program organized in procedures.

*a) **[No Gas Savings]** Procedure Rule 1: Collapsing Procedure Hierarchies:* Collapsing procedure hierarchies describes that procedure calls are replaced by inserting code directly into a smart contract (with appropriate binding of variables). Notably, this rule is the counterpart to Time-for-Space Rule 2: Interpreters that has already been discussed in Section IV-A. Following the idea of collapsing procedure hierarchies, the deployment of a smart contract is more expensive, since additional source code is needed. However, the invocation cost may decrease.

We conducted some pre-analysis for this rule by replacing iterative procedure calls by inlining code that shows the additional deployment cost being too high to yield any savings.

*b) **[Context-dependent Rule]** Procedure Rule 2: Exploit Common Cases:* This rule follows the goal to treat frequent cases efficiently. Thus, it is related to Space-for-Time Rule 1: Caching (see Section IV-B), which we found to be applicable to Solidity smart contracts. Identifying the occurrence of this rule and its application is a complex task, since it is necessary to have the domain expertise to identify the most frequent cases, and to implement accordingly efficient code for this special case. Naturally, the application of this rule is only meaningful if it is possible to apply a more efficient (while correct) code fragment. Thus, while it would make sense to apply this rule whenever a special piece of code for the most common cases leads to less gas cost, classification and fixing are very context-dependent.

*c) **[No Gas Savings]** Procedure Rule 3: Coroutines:* This rule optimizes code by adding coroutines [25]. Since the EVM does not support coroutines, this rule is not applicable.

*d) **[Automated Rule Classification]** Procedure Rule 4: Transformations on Recursive Procedures:* The transformation of recursive procedures into iterative procedures may lead to gas cost savings. In many cases, this will lead to increased deployment cost, e.g., since additional variables need to be defined. However, there might be less cost during smart contract invocation.

*e) **[No Gas Savings]** Procedure Rule 5: Parallelism:* Parallelism describes that tasks are carried out in parallel, e.g., using threads. Since there is no support for concurrency in the EVM [26], [27], this rule cannot be applied here.

F. Expression Rules

Expression rules describe the optimization of expressions, such as reusing results or replacing expensive expressions with cheaper ones. As will be discussed in the following paragraphs, this includes the adaptation of some of the already discussed rules to expressions.

*a) **[Included in Compiler]** Expression Rule 1: Compile-time Initialization:* This rule adapts principles of the previously discussed Loop Rule 1: Code Motion out of Loops, which we assessed to be applicable to Solidity smart contracts. Expression Rule 1 could for instance be applied by replacing an expression directly with its result, if possible. Our analysis has shown that this rule is already implemented in `solc`.

*b) **[Context-dependent Rule]** Expression Rule 2: Exploit Algebraic Identities:* The basic functionality of this rule has already been discussed in Logic Rule 1: Exploit Algebraic Identities. As discussed in Section IV-D, this rule is in general applicable to Solidity smart contracts, but it is not always trivial to identify these optimization parts in an automated fashion. Hence, it is necessary to analyze in more detail the algebraic identities which could be applied in Solidity. Unfortunately, this goes beyond the scope of the work at hand.

*c) **[Context-dependent Rule]** Expression Rule 3: Common Sub-expression Elimination:* This rule is related to Space-for-Time Rule 2: Lazy Evaluation. Its basic idea is to avoid the repeated calculation of the same expression by saving the result of the first invocation and then reusing this result later on. Hence, it makes sense to apply this rule if the number of accesses is large enough to justify the overhead of storing the precomputed values in a lookup table. As with lazy evaluation, the applicability of this rule needs to be answered separately for each use case. Therefore, the rule is not further regarded within the work at hand.

*d) **[Context-dependent Rule]** Expression Rule 4: Pairing Computation:* The goal of this rule is to combine expressions into pairs that are similar and can be evaluated together. These expressions are then implemented in a procedure. Pairing computation is related to Loop Rule 6: Loop Fusion, where several loops are combined, as well as Procedure Rule 2: Exploit Common Cases, where a new code fragment is introduced for common cases. For instance, instead of separately finding the minimum and maximum from a dataset, this could be combined into one expression. While in general useful for the cost optimization of Solidity smart contracts, the application of this rule is hampered by the difficulty to automatically identify application cases.

*e) **[Context-dependent Rule]** Expression Rule 5: Exploit Word Parallelism:* Following this rule allows to execute different operations on words in parallel, e.g., different Boolean operations. For this, it is necessary to put related bits next to each other in a word. However, the identification of related bits is context-dependent.

V. Evaluation

We empirically characterize rule violations and gas cost savings based on a representative dataset of Ethereum smart contracts written in Solidity. For this, we implemented the *python-solidity-optimizer* as an open source tool that classifies context-independent rules and, where applicable, provides non-intrusive fixes to measure gas cost savings. The python-solidity-optimizer is available at Github[4].

Figure 1 provides an overview of the system structure of the python-solidity-optimizer. In brief, the tool uses the python-solidity-parser[5] to generate a parse tree for each file from a repository of smart contracts defined by the user, applies the rules to these contracts, and generates smart contracts which are optimized (where non-intrusive fixes are applicable), provide mark-up information for the user, or both.

Notably, the python-solidity-optimizer allows to extend the set of rules by new rules, i.e., any researcher can use the tool to apply their own set of rules.

A. Metrics and Applied Rules

We measure for how many of the rules we can find optimization potential in the test dataset (see below) – this is done for the rules labeled **[Non-intrusive Fix]** and **[Automated Rule Classification]** (see below). For each violation of a rule (i.e., for each optimization potential), we mark this in the source code by adding a comment above the line of code where the violation has been found. In addition to the identification of

[4] https://github.com/TamaraBrandstaetter/python-solidity-optimizer
[5] https://github.com/ConsenSys/python-solidity-parser

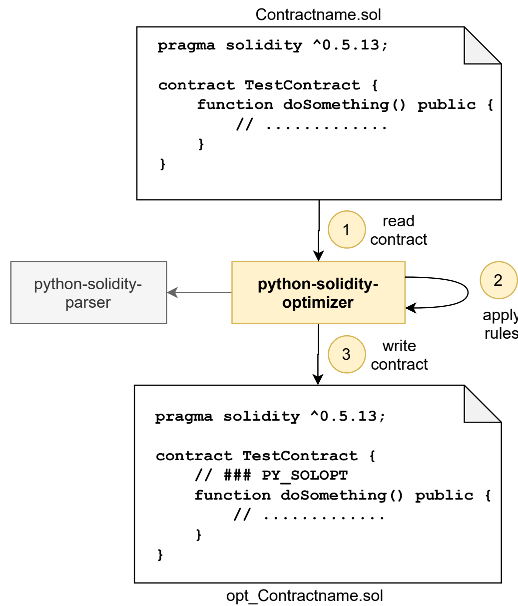

Fig. 1. Schematic overview of the `python-solidity-optimizer` that applies efficiency transformations on the original source code file and returns a new, optimized file

rule violations, we apply the non-intrusive fixes we identified. For these rules, we measure the gas cost savings.

By focusing on non-intrusive fixes, we make sure to avoid false positives with the disadvantage that we may not be able to detect all rule violations and miss some optimization potential.

We briefly reiterate and elaborate the rules we implemented in our prototype for classification and optimization. We detect and automatically provide non-intrusive fixes for the following rules:

- **Loop Rule 1: Code Motion out of Loops**: Repeated calculations are moved outside the loop and therefore only calculated once.
- **Loop Rule 3: Loop Unrolling for Simple Loops**[6]: Removing the loop and unrolling the content for each iteration reduces gas cost by saving the cost of modifying variables and checking conditions.
- **Loop Rule 4: Transfer-driven Loop Unrolling**: Removing trivial assignments (aliases) within a loop and replacing them with the values themselves reduces the required space.
- **Loop Rule 5: Unconditional Branch Removing**: Using a do-while loop instead of a while- or for-loop. This rule is implemented for the simple loops mentioned in Loop Rule 3, since it is necessary to decide if the loop is executed at least once in order to replace the loop with a do-while loop. Applying this rule removes a conditional jump operation at the beginning of the loop.
- **Logic Rule 1: Exploit Algebraic Identities**: Gas savings by replacing expensive expressions with semantically equivalent cheaper expressions, e.g., application of De Morgan's law.
- **Logic Rule 5: Boolean Variable Elimination**: Direct evaluation of a condition instead of storing the result in a (costly) Boolean variable.

Furthermore, violations are detected (but not automatically optimized) for the following rules:

- **Loop Rule 2: Combining Tests**: Potential to reduce the number of evaluated conditions within a loop in order to contain (in the best case) only one condition. Optimization is context-dependent and therefore not automatically applied.
- **Loop Rule 3: Loop Unrolling for Complex Loops**[7]: Removing the loop and unrolling the content for each iteration.
- **Loop Rule 6: Loop Fusion**: Combining successive loops across the same collections to one can save computation and space. Not automatically optimized, since each variable has to be checked across the loops to see whether it is possible to merge them without changing the semantics of the smart contract.
- **Procedure Rule 4: Transformations of Recursive Procedures**: Iterative algorithms usually provide cheaper invocations than recursive ones, since a smaller number of variables need to be modified and less conditions need to be checked. Therefore, iterative algorithms should always be preferred to recursive ones. However, optimizing this automatically is not a trivial task. Per se, this rule can be applied by identifying if a function with the same name is called within a function. Since it is possible to define several functions with the same name and the same number of parameters, but different parameter types, within one contract, this can easily lead to false positives.

We do not see these lists of rules as exhaustive. Rather, we see these rules as a baseline for a lower-bound of optimization potential when applying context-independent analysis and adaptation of smart contracts.

To calculate the actual gas savings achieved due to the implemented rules, we deploy all contracts that are automatically optimized by our prototype in a test environment with the help of the Remix IDE[8]. Again, we use `solc` v0.5.13 for compilation. We deploy the contracts before and after the optimization to compare the gas usages.

Since the required gas also depends on parameters when deploying contracts, we use the same parameters for both contracts (i.e., optimized and not optimized) to make them comparable.

If a contract is used within another contract as a dependency, that particular source code is also included in the source file. Therefore, violations in the dependency contract can be counted multiple times, depending on how often they are

[6]The prototype detects all rule violations of that kind – see below. Only simple loops that only contain expressions are automatically optimized.

[7]Loops that contain nested loops or conditions are not automatically optimized because this would lead to even higher deployment cost.

[8]https://remix.ethereum.org/

used. Since the used dependency also has to be deployed, we cannot filter out these violations. Even though the rule might be violated by another smart contract developer, the gas consumption affects the new contract, too.

After the deployment, we call all functions that are optimized by our prototype to compare the gas usages of the function calls. Again, we use the same parameters for both function calls to make them comparable. Some functions in our dataset, particularly those used in games, use randomization. We rewrite those randomized functions to return a fixed value, guaranteeing the same execution for the initial and the optimized version of the contract.

B. Dataset

We apply the implemented rules to 3,018 smart contracts[9] from *etherscan.io*[10]. etherscan.io is a well-known online service that offers the possibility to verify source code of smart contracts. The goal behind the service is to create transparency and to strengthen the user's confidence in using a smart contract. Data from etherscan.io has been used for the evaluations in a number of research papers, e.g., [28], [29], [30].

We explicitly selected the 3,018 *verified* and *open source* contracts that were available to us at etherscan.io when conducting the evaluations. The set of open source smart contracts at etherscan.io contains a number of popular and commonly-used applications, which are developed as a joint effort by the Ethereum community, e.g., SafeMath[11]. We therefore expect that the quality of most of the smart contracts used in our evaluation is already at a high level. This way, we can evaluate if the implemented rules lead to significant gas savings even for such cases.

C. Results

Figure 2 provides an overview of the rule violations (i.e., for rules labeled **[Non-intrusive Fix]** or **[Automated Rule Classification]**) detected in our dataset of 3,018 verified smart contracts. The yellow bars show the number of violations found for a particular rule, the gray bars show the number of affected contract files in which the violations occur (in order to assess multiple violations in one smart contract).

In total, we identify 471 rule violations within the test dataset. Contracts tend to contain multiple violations. The most common case is that two violations occur in the same file, but one smart contract file contained as much as 12 violations. The 471 violations are spread across 204 input files from the dataset, yielding a violation rate of about 6.76%.

Table II provides an overview of the cost savings when applying the six implemented automatic optimizations, i.e., the rules labeled **[Non-intrusive Fix]**. For each of the applied rules, we calculate the total and average gas cost for deployment and for invocations of a smart contract. This is

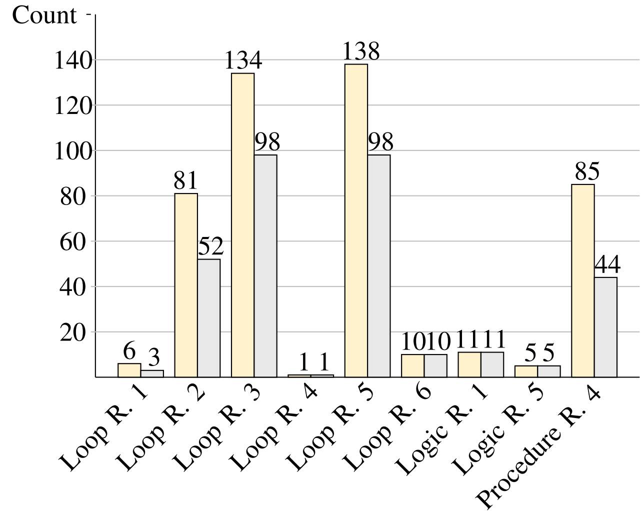

Fig. 2. Rule violation count in *etherscan.io* contracts

done for both the original files from etherscan.io and the optimized versions of the smart contracts. The calculation of the average cost is based on the number of functions a violation is detected in. Thus, if a function contains the same violation several times, the function nevertheless only counts once for the calculation of the average.

Loop Rule 1: Code Motion of Loops finds a total of six violations in three smart contracts. As it can be seen in Table II, gas cost for both deployments and invocations can be achieved. On average, this leads to savings of 0.197% for the deployment and 0.101% for each invocation of the smart contracts in the test dataset.

Loop Rule 2: Combining Tests finds a total of 81 violations in 52 smart contracts. We encounter a total of 4,469 loops in the test dataset. Out of these, 3,943 are for-loops, with 13 violations of this rule coming from this loop type. Furthermore, we identify 508 while-loops and 18 do-while-loops in the test dataset, leading to 66 violations for the while-loops and two violations for the do-while-loops. The analysis shows that there are more likely multiple tests in a while-loop, leading to the 13% violation rate for this rule, while the other two loop types are less affected.

Loop Rule 3: Loop Unrolling is regarded in two different ways: Once for simple loops, for which the prototype detects violations and automatically optimizes the code, and for complex loops, where violations are detected, but not automatically optimized (see Section V-A).

Four violations regarding simple loops are found, while more complex loops show 130 violations. This means that about 2.9% of all loops are affected by this rule.

Following this rule, loops are automatically unrolled if they would run a maximum of four iterations. This limit has been set in order to take into account the higher deployment cost if a loop is unrolled (see Table II). In fact, as can be seen in the table, the deployment cost for this rule increase by 43,591 gas on average (or 4.771%), while the invocation cost

[9]The list of used smart contracts can be found at https://github.com/TamaraBrandstaetter/python-solidity-optimizer.

[10]https://etherscan.io

[11]https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol

TABLE II
DEPLOYMENT GAS COST OVERVIEW GROUPED BY RULES

	Deployment		Invocation	
	Total	Average	Total	Average
	Loop Rule 1			
Original	2,082,519	694,173	570,469	142,617
Optimized	2,078,417	692,806	569,891	142,473
Difference	**-4,102**	**-1,367**	**-578**	**-144**
	Loop Rule 3			
Original	3,654,299	913,575	176,443	44,111
Optimized	3,828,662	957,166	174,761	43,690
Difference	**174,363**	**43,591**	**-1,682**	**-421**
	Loop Rule 4			
Original	481,510	481,510	30,569	30,569
Optimized	479,302	479,302	30,519	30,519
Difference	**-2,208**	**-2,208**	**-50**	**-50**
	Loop Rule 5			
Original	204,034,849	2,081,988	4,420,417	35,363
Optimized	203,908,250	2,080,696	4,404,346	35,235
Difference	**-126,599**	**-1,292**	**-16,071**	**-129**
	Logic Rule 1			
Original	30,814,434	2,801,312	404,838	36,803
Optimized	30,810,740	2,800,976	404,805	36,800
Difference	**-3,694**	**-336**	**-33**	**-3**
	Logic Rule 5			
Original	14,140,841	2,828,168	152,509	30,502
Optimized	14,134,321	2,826,864	152,459	30,492
Difference	**-6,520**	**-1,304**	**-50**	**-10**

only decrease on average by 421 gas (or 0.954%). Thus, we are able to redeem the additional deployment cost by 104 invocations on average. If unrolling loops with an iteration number higher than four, the deployment cost would increase even more, so that the number of invocations to balance out the higher deployment cost would on average be higher than 300. Since the internal optimizer of `solc` applies optimizations only if they pay off within 200 invocations, this is also the limit we take into account here.

Loop Rule 4: Transfer-driven Loop Unrolling is only applicable in one case in the test dataset, leading to 0.459% less deployment cost and 0.164% less invocation cost. However, because of the limited number of occurrences, these savings cannot be generalized.

Loop Rule 5: Unconditional Branch Removing is the most-applied rule within the whole test dataset, with 138 violations found in 98 smart contracts. As pointed out above, this rule was applied to the same simple for-loops as considered in Loop Rule 3. About 3% of all loops in the test dataset violated this rule. As can be seen in Table II, both deployment (on average by 0.062%) and invocation cost (0.364%) are decreased when optimizing smart contracts following this rule.

Loop Rule 6: Loop Fusion is violated ten times in ten different smart contracts, which is in line with our expectations, since multiple looping over the same collection is rarely used.

Logic Rule 1: Exploit Algebraic Identifies finds a total of 11 violations in 11 smart contracts, with rather small gas savings for both deployment (0.012%) and invocations (0.008%). When examining this rule further, we observe that only 31 if-statements in the whole dataset contain multiple negations. However, out of these, 35.4% violate the implemented rule.

Logic Rule 5: Boolean Variable Elimination is violated five times across five contracts. The relative cost savings are on average 0.046% per deployment and 0.033% per invocation.

Procedure Rule 4: Transformations of Recursive Procedures is only implemented in order to identify rule violations, optimizations are not automatically conducted (see above). A total of 85 violations in 44 different smart contract files are found in the test dataset.

D. Discussion

With 6.76% of all smart contracts showing at least one rule violation, the number of violations in the test dataset is quite significant. In fact, the high number of affected smart contracts was a little bit of a surprise to us, since we expected that the smart contracts in the test dataset would be relatively well-maintained. This notion is based on the fact that the 3,018 validated open source smart contracts involve a number of popular and commonly-used smart contracts, which we did not expect to violate the rules very often. Other studies have shown that popular contracts used very frequently in transactions do show a smaller amount of optimization potential [19]. We therefore expect an even higher violation rate if the test dataset would be extended to further smart contracts.

While the number of rule violations in the test dataset is surely significant, it could be argued that the gas savings per deployment and invocation are quite small. On average, 1,213 gas is saved when deploying an optimized smart contract, and 123 gas for each invocation. While the absolute savings for each deployment and invocation might be small, gas cost in most of 2019 and 2020 in the Ethereum blockchain alone are in the range of 100,000 to 250,000 US dollars each day, or even higher (see Section I). Therefore, even small cost savings can sum up to high absolute numbers.

As a side effect of the study conducted, we are able to identify that `solc` (v0.15.3) only includes a very limited subset of the investigated optimization rules, namely Space-for-Time Rule 3: Caching and Expression Rule 1: Compile-time Initialization. Based on the results of our experiments, it is surely promising to explore the integration of further optimization rules into `solc`.

Last but not least, the application of the discussed rules may also lead to negative side-effects, e.g., by decreasing the readability of the smart contract code. However, it should be noted that our python-solidity-optimizer optimizes code directly before compilation into byte code, i.e., a developer is still able to see the original, not-optimized code.

VI. CONCLUSIONS

Within this paper, we examined how basic optimization strategies from the field of software engineering can be applied to Solidity smart contracts in order to reduce their gas consumption. For this, we have analyzed 25 optimization strategies with respect to potential gas savings when being

applied to Solidity smart contracts. We utilized a defensive white-listing approach when analyzing the rules, i.e., only known patterns are recognized as rule violations.

We have identified 21 of these strategies as generally applicable to smart contracts: Two of these rules are already implemented in `solc`, ten rules highly depend on a specific use case (i.e., the context), and nine rules have been identified as candidates for automatic optimization. We have implemented these nine rules in order to find rule violations in smart contracts. Out of these nine rules, we have also implemented six rules in order to automatically optimize smart contracts. We then evaluated our implemented rules for a test dataset from etherscan.io, finding violations in 6.76% of the smart contracts – despite the supposed high quality of the 3,018 verified open source contracts which we applied in our experiments. The saved gas cost differ significantly between different rules, but are overall substantial.

While these results are already very promising, we nevertheless see them primarily as a first step towards smart contract optimization. In our future work, we want to extend especially the rule detection and the automatic optimization, i.e., we want to increase the amount of detected violations and further automate rule application in these detected cases. For instance, we are currently working on extending the exploitation of algebraic identities by further cases.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *White Paper*, 2008.

[2] A. Zohar, "Bitcoin: Under the Hood," *Comm. of the ACM*, vol. 58, no. 9, pp. 104–113, 2015.

[3] F. Tschorsch and B. Scheuermann, "Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies," *IEEE Comm. Surveys and Tutorials*, vol. 18, no. 3, pp. 2084–2123, 2016.

[4] C. Dannen, *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, 2017.

[5] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2019.

[6] J. Nagele and M. Schett, "Blockchain Superoptimizer," in *29th International Symp. on Logic-based Program Synthesis and Transformation*, Springer, 2019.

[7] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th Int. Conf. on Software Analysis, Evolution and Reengineering*, pp. 442–446, IEEE, 2017.

[8] M. di Angelo and G. Salzer, "A Survey of Tools for Analyzing Ethereum Smart Contracts," in *2019 IEEE Int. Conf. on Decentralized Applications and Infrastructures*, pp. 69–78, IEEE, 2019.

[9] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang, "Towards saving money in using smart contracts," in *2018 IEEE/ACM 40th Int. Conf. on Software Engineering: New Ideas and Emerging Technologies Results*, pp. 81–84, IEEE, 2018.

[10] 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 T. on Emerging Topics in Computing*, 2020.

[11] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "EthIR: A Framework for High-Level Analysis of Ethereum Bytecode," in *International Symp. on Automated Technology for Verification and Analysis*, pp. 513–520, Springer, 2018.

[12] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts," in *26th Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems*, pp. 118–125, Springer, 2018.

[13] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2nd Int. Works. on Emerging Trends in Software Engineering for Blockchain*, pp. 8–15, IEEE / ACM, 2019.

[14] E. S. Lowry and C. W. Medlock, "Object code optimization," *Comm. of the ACM*, vol. 12, no. 1, pp. 13–22, 1969.

[15] J. L. Bentley, *Writing Efficient Programs*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1982.

[16] R. Leupers, *Code Optimization Techniques for Embedded Processors*. Springer-Science+Business Media, B.V., 2000.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[18] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and Tools for the Static Analysis of Ethereum Smart Contracts," in *30th Int. Conf. on Computer Aided Verification*, pp. 51–78, Springer, 2018.

[19] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He, Y. Tang, X. Lin, and X. Zhang, "SODA: A Generic Online Detection Framework for Smart Contracts," in *27th Ann. Network and Distributed Systems Security Symp.*, The Internet Society, 2020.

[20] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *White Paper*, 2014.

[21] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "Caramel: Detecting and fixing performance problems that have non-intrusive fixes," in *2015 IEEE/ACM 37th IEEE Int. Conf. on Software Engineering*, pp. 902–912, IEEE, 2015.

[22] I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, and J. Mendling, "Untrusted Business Process Monitoring and Execution Using Blockchain," in *14th Int. Conf. on Business Process Management*, pp. 329–347, Springer, 2016.

[23] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *J. of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, 2011.

[24] J. Hughes, "Why Functional Programming Matters," *The Computer J.*, vol. 32, no. 2, pp. 98–107, 1989.

[25] A. L. D. Moura and R. Ierusalimschy, "Revisiting coroutines," *ACM T. on Programming Languages and Systems*, vol. 31, no. 2, pp. 1–31, 2009.

[26] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding Concurrency to Smart Contracts," in *ACM Symp. on Principles of Distributed Computing*, pp. 303–312, ACM, 2017.

[27] I. Sergey and A. Hobor, "A Concurrent Perspective on Smart Contracts," in *Int. Conf. on Financial Cryptography and Data Security*, pp. 478–493, Springer, 2017.

[28] W. Chen, Z. Zheng, J. Cui, E. Ngai, P. Zheng, and Y. Zhou, "Detecting Ponzi Schemes on Ethereum: Towards Healthier Blockchain Technology," in *2018 World Wide Web Conf.*, pp. 1409–1418, ACM, 2018.

[29] R. Norvill, B. B. F. Pontiveros, R. State, I. Awan, and A. Cullen, "Automated labeling of unknown contracts in Ethereum," in *2017 26th Int. Conf. on Computer Communication and Networks*, pp. 1–6, IEEE, 2017.

[30] S. Linoy, N. Stakhanova, and A. Matyukhina, "Exploring Ethereum's Blockchain Anonymity Using Smart Contract Code Attribution," in *15th Int. Conf. on Network and Service Management*, pp. 1–9, IEEE, 2019.