Information and Software Technology 186 (2025) 107800

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Neural-guided superoptimization in ethereum

Matheus Araújo Aguiar [a], Elvira Albert [b], Samir Genaim [b], Pablo Gordillo [b,*], Alejandro Hernández-Cerezo [b], Daniel Kirchner [a,c], Albert Rubio [b]

[a] Ethereum Foundation, Zug, Switzerland
[b] Complutense University of Madrid, Madrid, Spain
[c] University of Bamberg, Bamberg, Germany

ARTICLE INFO

Dataset link: https://github.com/costa-group/gasol-optimizer

Keywords:
Smart contracts
Ethereum
Optimization
Machine learning

ABSTRACT

Context: Superoptimization is a synthesis technique that, given a *loop-free sequence* of instructions, searches for an equivalent sequence that is *optimal wrt.* an objective function. Superoptimization of Ethereum smart contracts aims at minimizing the *size of their bytecode* and the *gas consumption* of executing the contract's functions. The search for the optimal solution poses huge computational demands – as the search space to find the optimal sequence is exponential on the given *size-bound* – being the main challenge for superoptimization today to scale up to real, industrial software. Even if the underlying problem for finding the optimal solution is decidable, practical tools often prioritize efficiency over completeness. This means they might be implemented to find a sub-optimal solution or even time out.
Objective: This work aims at leveraging superoptimization to a real setting: Ethereum blockchain. This paper proposes a *neural-guided superoptimization* (NGS) approach which incorporates deep neural networks using (supervised) learning into superoptimization to improve scalability by predicting: (1) if a sequence is already optimal and hence the search can be skipped; (2) the size-bound for the optimal solution in order to reduce the search space.
Method: We have downloaded over 13,000 smart contracts deployed on the blockchain for training and testing the machine learning models, and a disjoint set with 100 of the smart contracts with more transactions to prove our scalability gains and impact for the Ethereum community.
Results: Incorporating DNNs resulted in a 16x overall speedup (12x for gas) with only 12% optimization loss (14% for gas), or a 3-4x speedup with no optimization loss. For the 100 analyzed contracts, this approach reduced the average compilation time to 3 min per contract and achieved monetary savings of $1.24M.
Conclusions: The integration of machine learning models mitigates several limitations of traditional superoptimization by drastically reducing execution times while maintaining most of the original optimization gains.

1. Introduction

This section introduces: the Ethereum blockchain, that constitutes the industrial setting to which our work is applied; the technique of superoptimization and its practical applications; the challenges of applying it on an industrial setting; the decidability of the problem; the machine learning (ML) technology we use to define NGS; and a brief summary of the main contributions of our work.

The Ethereum blockchain. The industrial context of this work is the Ethereum blockchain, a technology booming digital economy by means of thousands of decentralized applications: its current market capitalization is around $220B [1], the funds in decentralized finance (DeFi) applications are around $51.86B, and the number of transactions processed on the network is huge (e.g., on average, there were 1.05M transactions per day in 2023[1]). The Ethereum Blockchain distinguishes itself from other chains, such as Bitcoin, by aiming to be a platform to build decentralized applications that range from social networks and identity systems to supply chain management and finances. The stored data contains not only accounts and transactions, but also a machine state which changes from block to block according to predefined rules

* Corresponding author.
E-mail addresses: matheus.pit@gmail.com (M.A. Aguiar), elvira@sip.ucm.es (E. Albert), samir.genaim@fdi.ucm.es (S. Genaim), pabgordi@ucm.es (P. Gordillo), aleher06@ucm.es (A. Hernández-Cerezo), daniel.kirchner@ethereum.org (D. Kirchner), alberu04@ucm.es (A. Rubio).

[1] Data gathered from https://etherscan.io/txs on Jan 14, 2024.

https://doi.org/10.1016/j.infsof.2025.107800
Received 13 June 2024; Received in revised form 22 May 2025; Accepted 23 May 2025
Available online 16 June 2025

and arbitrary programs, called *smart contracts*, consisting of machine instructions or *opcodes*. The Ethereum Virtual Machine (EVM) defines the specific rules of state changing from block to block. Execution cost in Ethereum is tightly metered to, among reasons, prevent Denial of Service attacks on the network and is measured per opcode with a metric called *gas*. Such costs can be significant, e.g., one of the largest exchange platforms Uniswap (which reached more than US$ 1.5 trillion trade volume in 2023 [2,3]), through one contract, Uniswap V3 [4], has consumed more than 1.8 trillion gas [5] since 2022. Because the amount of gas consumed is a determinant factor in the fee paid for a transaction, which is usually handed down to the end-user, it is important that smart contracts use the most efficient sequences of instructions to perform their actions. Optimization is critical to the scalability and composability of contracts and the services they provide, since it reduces the consumed gas and thus results in lower transaction fees. This benefits not only the user of the contract, who pays a lower fee, but also the entire network which has its load work reduced. Hence the industry spends a lot of effort in hand-optimizing contract code to reduce the execution costs. However, manually optimized code (often written in assembly) is harder to read and audit and thus more error-prone than less-optimized high-level code. This can cause significant problems for example the security of contracts that move large sums of funds, as Uniswap which reached more than US$ 1.5 trillion trade volume in 2023 [2,3]. Therefore, automatic optimization techniques are highly sought after and can provide significant benefit in this context.

Superoptimization and its applications. Superoptimization is a synthesis technique, pioneered by Massalin [6], that aims at finding the optimal translation of an initial *loop-free* sequence of instructions by searching *all* possible alternative sequences that are semantically equivalent to the initial one *automatically*, e.g., by using a SAT or SMT solver. Hence, superoptimized code has two appealing features: *correctness*, as semantic equivalence must be guaranteed, and *optimality*, as the final goal is to find the optimal code for the considered objective function.

Superoptimization has very interesting applications. (1) Our main target is its use as an extra layer of optimization within the compiler. Superoptimization can typically find optimizations which are hard (or impossible) to be achieved by relying on predefined rule transformations, as traditional optimizing compilers do. For instance, a superoptimizer applied on stack-based bytecode can achieve optimizations by reordering the elements of the stack so that the number of `SWAP` opcodes is reduced; (2) Also, superoptimization has been used to learn new peephole optimizations [7], which are local rewrites to improve the efficiency and the code size, which can then be incorporated as rule optimizations within a compiler. (3) Finally, it achieves that advances in hardware design can be easily and safely transferred to software: rather than having to rewrite the compiler to generate efficient code for new hardware, one can provide a superoptimizer with the instruction set and corresponding cost model for the new hardware and use it to generate efficient code.

In spite of these relevant applications and its clear potential to produce greener software, the use of superoptimization is not as widespread as one could imagine. The main limitation for its use in industry has been its huge computation demands: searching for optimal solutions is very hard to scale.

Challenges in scaling superoptimization. The problem of finding the optimal code is a type of program synthesis problem (a problem which is known to be hard to scale): from unoptimized code acting as a specification, the synthesis procedure attempts to create a more efficient implementation. An important class of synthesis algorithms work by enumerating candidates and checking if each candidate meets the specification. This approach has been applied to superoptimization as well in [7,8]. A major step for enhancing scalability was given by [9] with the idea of *unbounded* superoptimization. Basically, instead of searching through the full space of candidate instruction sequences and then calling the solver, unbounded superoptimization consists in lifting the *search* for the target program to the solver. The advantage is that SMT solvers are able natively to prune incorrect solutions and search through a much smaller space of correct solutions towards finding an optimal one. Such SMT-based approach is being used by modern superoptimizers such as Souper [10], ebso [11] and GASOL [12]. In spite of unprecedented advances in SMT solvers, searching for optimal solutions is very hard to scale when the sequences to be optimized are of considerable size, as it often happens. This is because the search space grows exponentially with the *length of the sequence* assumed when searching for the optimal solution (named *size-bound* in what follows). This article identifies two challenges for scaling SMT-based approaches to superoptimization that, in our case, have been key to handle real smart contracts *efficiently* (i.e., greatly reducing optimization times) and *effectively* (i.e., losing none or little optimization, and even sometimes gaining):

Challenge 1: Avoid unnecessary search. It can be the case that the original sequence produced by the compiler is already the optimal one (e.g., our experiments reveal that this happens quite often for sequences of stack-manipulating bytecode). The challenge is to be able to know it beforehand and avoid unnecessary exploration for sequences that are already optimal.

Challenge 2: Use accurate size-bounds. When invoking the SMT solver, a *bound* for the size of the solution needs to be given to bound the search. The more accurate the bound is, the more efficient (and more scalable) the search will be. However, giving a bound smaller than the required size of the solution might lead to an unsatisfiable problem for the SMT solver (and hence all possible optimization lost). The size of the original code is often a *sound* upper bound (i.e., there exists a optimal solution whose size $s \leq$ than the bound) but not necessarily sound for all objective functions (e.g., when optimizing the size in bytes having less opcodes but large constants is more expensive than having more opcodes and smaller constants) and, importantly, it is often larger than needed (hence making the overall cost of superoptimization exponentially more expensive). The challenge is to infer accurate size-bounds for the optimal solution.

Decidability of superoptimization. Given a jump-free block of code, the problem of synthesizing a more efficient equivalent block (when considered in isolation from the rest of the program) is decidable (though computationally intractable). Note that the size of the optimal block can be at most equal to the cost of the original block, since each instruction costs at least 1 unit, and the cost of the original block can be statically bounded. This means that we can enumerate all possible blocks up to this size, execute them on all possible inputs, and compare the results with those of executing the original block (this is possible because stack, memory, and storage are finite).

In the context of GASOL [12], this process is even simpler. Because instead of searching the whole space of possible blocks, we only consider blocks that (1) use a subset of the original instructions, possibly with the addition of instructions like `PUSHX`, `SWAPX` and `DUPX`; and (2) respect some order dependencies between the memory/storage access instructions. GASOL [12] formulates this optimization problem as an SMT formula with linear arithmetic, which makes the problem (of finding a block of this particular type) decidable. This will be explained in Section 3 in detail.

It is important to note that even if the problem is decidable, actual implementations may not find an optimal solution given enough time. This is on one hand due to design choices that sacrifice completeness for efficiency in some of their components. In other words, they may sometimes find a solution that is not optimal but only better than the original code. Besides, completeness is also lost due to the use of time-outs in practice.

Machine learning. Given the challenges mentioned above, and the nature of our setting, we will use supervised learning [13] to build models that predict the corresponding answers. In such approach the model is trained using existing data that is labeled with the correct answers, which in our case consist of sequences of opcodes labeled with answers that correspond to the above challenges. We will rely on deep neural networks to build our model, which also allows learning an adequate representation of the input data. In particular, we will use recurrent neural networks (such as LSTM [14]) that are used to learn from sequences. This kind of network is ideal for solving problems where a sequence of items carries more information than individual items. This is the case of our setting since optimization does not focus on a single instruction, but rather on their interaction and possibly the order in which the instructions are applied.

Summary of contributions. This paper introduces *neural-guided superoptimization* (NGS for short): a superoptimization framework which incorporates advanced ML technology to approach *Challenges 1&2* in Section 1. The framework is realized, and experimentally evaluated, in the real context of the Ethereum blockchain. Briefly, the main contributions of the paper are summarized as follows:

1. We present a classifier model that predicts whether a sequence of opcodes is already optimal based on the list of opcodes in the initial sequence (*Challenge 1* above).
2. We introduce a regression model that infers an accurate bound for the size of the optimal solution (*Challenge 2* above).
3. We leverage a superoptimization algorithm for EVM smart contracts to rely on the previous two models to greatly speed up the process while little optimization gains are lost.
4. We implement our approach within the GASOL superoptimizer [12] and assess its performance and impact on 100 of most-used smart contracts (arguably the most relevant ones) deployed on Ethereum.

2. Background

Superoptimization. Superoptimization [6] is a compilation technique that, given a loop-free sequence of instructions, searches for a semantically equivalent sequence of instructions that is optimal *wrt.* an optimization criteria. There exist superoptimization tools for a range of languages e.g., for LLVM [9,10], for EVM [12,15], for Wasm [16], among others. As mentioned before, actual tools may not find an optimal solution given enough time when they use design choices that sacrifice completeness for efficiency in some component. This means that they may sometimes find a solution that is not optimal but only better than the original code. Besides, completeness is also lost due to the use of time-outs in superoptimization tools.

The Ethereum virtual machine. The EVM contains more than 150 opcodes [17] representing instructions that allow for arithmetic, bit-wise, cryptographic and memory/storage operations, as well as opcodes for control flow and stack manipulation. The EVM operates as a stack machine with a depth of 1024 words, each of which has 256 bits in size. There is also a temporary memory, much like RAM, that only stays alive during the execution of a single call to a contract, but does not persist across calls between contracts or between transactions. Finally, there is a persistent key–value storage which is allocated to every account on the chain. Each instruction of the EVM is represented using a unique opcode, which consists of a single byte, which is in some cases followed by an immediate argument for pushing constants to the stack. Each opcode has a gas cost associated with it. For example, opcode 0x01 corresponds to the instruction `ADD`, which consumes two operands from the stack and produces their sum (module 2^{256}) as result on the stack and has a cost of 3 gas. Since opcodes generally operate on the top of the stack, the EVM involves specialized opcodes for duplicating elements deeper on the stack (`DUP1..DUP16`) and swapping the top element of the stack with deeper elements (`SWAP1..SWAP16`). Constants can be put on stack using a set of `PUSH` opcodes with differently sized immediate arguments, and elements can be removed from stack using a `POP` opcode. Control flow is described using unconditional jumps (`JUMP`) and conditional jumps (`JUMPI`) to absolute offsets into the bytecode that have to be marked with the `JUMPDEST` opcode. The most expensive opcodes are the ones manipulating the persistent key–value store, `SLOAD` and `SSTORE`.

Optimization of EVM. The natural and most important optimization criterion for EVM bytecode is the runtime *g*as *c*ost (**gc**) that is incurred upon the execution of a transaction. However, deploying EVM bytecode to the Ethereum blockchain is also associated with a one-time cost of 200 gas for each byte of contract code, which necessitates a tradeoff between optimizing for runtime gas cost (finding the sequence of cheapest opcodes for any given computation) and optimizing for *s*ize in *b*ytes (**sb**) that results in the deploy time cost of the contract (finding the shortest sequence of opcodes for any given computation). This tradeoff depends on the expected number of interactions with the contract over its lifetime. While in most cases (e.g. in the context of a trading platform) a lot of transactions are expected over the lifetime of a contract tilting this tradeoff strongly in favor of optimization for runtime gas costs, some smart contracts like wallets intended for long-term storage of assets may only expect a low number of transactions and favor optimization for deploy time cost instead.

Objective functions. The objective functions used in our experimental evaluation are those available in the GASOL system [12] to measure optimization of EVM code: **gc** assigns to each EVM opcode the amount of gas it consumes as specified in [17]; **sb** assigns size 1 to all opcodes except `PUSHX` that has size X+1 bytes, 1 for the `PUSH` and X for the argument.

3. Neural-guided superoptimization

This section describes the architecture of the proposed NGS framework in a language-agnostic way — which can be applied to practitioners outside of the context of Ethereum (e.g., to the above tools). Fig. 1 provides a general overview of the whole process (the new ML components appear in dashed boxes): the process starts from a program **P** to be superoptimized ① and a selection of an objective function **Obj** ⑤ (among those available in the system), and yields an optimized code ⑩ (optionally with a report of the gains **G**). According to the original formulation in [6], superoptimization must guarantee:

(i) **(correct)** $\mathbf{P}'$ is semantically equivalent to **P**, written $\mathbf{P}' \equiv \mathbf{P}$;
(ii) **(optimal)** the cost of $\mathbf{P}'$ is minimal *wrt.* **Obj**, i.e., $\mathbf{Obj}(\mathbf{P}') = \mathbf{min}\{\mathbf{Obj}(\mathbf{P}'') \mid \mathbf{P}'' \equiv \mathbf{P}\}$;

Loop-free sequences. As mentioned in Sections 1 and 2, superoptimization must be applied on loop-free sequences of instructions. Component ② takes the input program **P** to be optimized and generates all loop-free sequences **S** from **P** on which the search for the optimal sequences will be applied. This step requires the generation of the CFG for **P** from which loops can be detected and the loop-free sequences extracted. A common approach is to define **S** as the sequences of instructions within each block of the CFG (as done, e.g., in [10,11,18]). There are other tools though that consider larger or smaller units. For instance, unbounded superoptimization as presented in [9] treats as many blocks together as possible and even propagates the guards in conditional statements to the branches of the conditional. However, only small-sized programs have been considered in the experiments in [9], and the approach might become too expensive for larger code. This is why other superoptimization tools [12] split the blocks of the CFG into sub-blocks [12] when the size of the full block exceeds a fixed maximal size. Component ② generalizes any of the above instantiations.

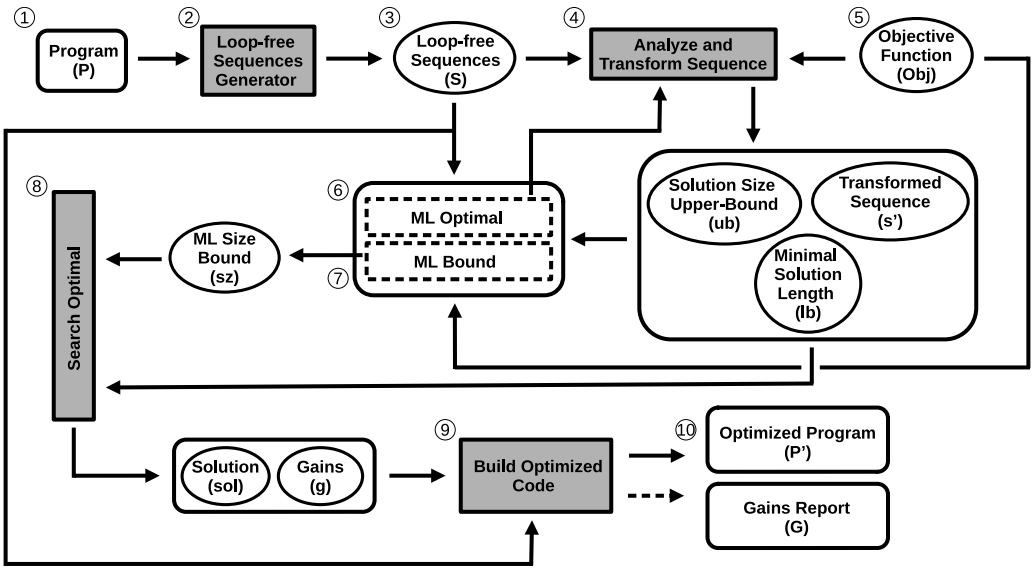

Fig. 1. Architecture of a Neural-Guided Superoptimizer. First, loop-free sequences are extracted from the source program (component ②). From these sequences, a higher-level representation is synthesized to guide the search (component ④). Machine learning models (components ⑥ and ⑦) enhance the search for improved sequences (component ⑧). Finally, the optimized program is reconstructed from the resulting sequences.

ML optimal. The aim of component ⑥ is to classify the loop-free sequences according to whether they can be optimized or not (addressing *Challenge 1* in Section 1). It can be realized as a binary classification problem that generalizes to existing techniques in supervised learning such as decision trees, support vector machines (SVM), or recurrent neural networks (RRN). For the particular programming language used, an appropriate representation has to be chosen to encode the original sequence in order to be able to apply the ML technology. If the sequence **s** should not be optimized, all the optimization process is skipped and the original sequence is passed directly to the last component ⑨ (see description below). Otherwise, the process continues with the analysis and transformation of the sequence. This component can greatly improve the performance of the approach as there are sequences that are no longer analyzed, saving the time spent in computing the optimization and in proving the optimality of the solution.

Analysis and transformation of sequences. Component ④ takes a sequence of instructions $\mathbf{s} \in \mathbf{S}$ and analyzes it in order to gather different types of information, which in turn may lead to transforming the sequence. Most tools perform some form of symbolic execution (which usually requires a transformation into SSA form) to synthesize a higher-level intermediate representation ($\mathbf{s}'$) from which the search-query can be more directly generated. For instance, [10] targets LLVM and C++ programs by using their own intermediate representation, while [9] uses the existing LLVM intermediate representation directly to build the constraints passed to the corresponding solver. Besides, the intermediate representation $\mathbf{s}'$ can be simplified at this stage before proceeding to the next one ⑧, e.g., by carrying out arithmetic/bit-wise operations that can be fully executed (like addition by zero, etc.) and obtain a simplified transformed form [12,19]. Besides transforming and simplifying **s**, the analysis can gather useful information from it. One type of analysis that not every optimizer uses is a size-bound predictor that provides an upper-bound **ub** for the size of the optimal solution and the minimum size that it can have **lb**. The bounds could be obtained from **s**, but also from $\mathbf{s}'$, and may vary according to the selected **Obj** (see *Challenge 2*). We generalize within this component ④ a range of analysis and transformation techniques that may be applied to the original sequence and simply assume that we obtain here a transformed sequence ($\mathbf{s}'$) and the minimal length that the solution can have (**lb**) and an upper-bound for it (**ub**). In the absence of this component, we assume that ④ simply returns **s** and $\perp$ for the size-bounds and thus the original instructions are used as result.

ML bound. Component ⑦ aims to use advanced ML technology to predict, given a loop-free sequence **s**, the size **sz** of the corresponding optimal one (addressing *Challenge 2* in Section 1). This component generalizes to a regression problem, where the input is a sequence of instructions and the output is a prediction **sz** for the size of the optimal solution. The results inferred by this learning might enhance both the performance and the savings obtained. Clearly, a better bound reduces (exponentially) the search space for the optimal solution, what might avoid timeouts in the search (and in turn might enable finding a solution and increase optimization gains).

Search optimal. Component ⑧ aims at searching all possible solutions that are semantically equivalent to the original sequence **s**, or to its transformed form $\mathbf{s}'$, and it is carried out by a constraint solver. A constraint encoding of the problem is automatically generated and **sz** is used to constrain the size of the solution. The semantics of the instructions of the programming language being used must have been encoded (with their corresponding costs for the objective functions available). The search does not consider solutions that are larger than the given bound **sz** (hence optimality results are ensured for solutions up to this bound). The procedure returns the solution **sol** and the gain **g** if a better solution has been found and otherwise it returns false. State-of-the-art superoptimizers use SMT solvers [9–12] or SAT solvers [20,21].

BuildOptimizedCode. There is a final engineering step ⑨ in which, from the solution, the sequence of instructions for the optimized code is generated and the final optimized program $\mathbf{P}'$ is rebuilt (e.g., jump addresses have to be recomputed).

4. Neural-guided superoptimization in Ethereum

This section describes the realization of NGS and its associated ML technology in the context of Ethereum. Section 4.1 first presents the basic features of the algorithm used for superoptimizing EVM smart contracts (without ML) and the following sections focus on the ML components: Section 4.2 describes how EVM blocks are encoded; Section 4.3 describes the set of blocks we use for training/validation; Section 4.4 describes the model we build for predicting if a given EVM block is already optimal; and Section 4.5 describes the model we build for predicting the size of the optimal block.

4.1. EVM superoptimization (without ML)

Input: Smart contract **P**, Objective function **Obj**, Timeout **Tout**
Output: Optimized EVM bytecode **P′**, Gain **G**
Ensures: $\mathbf{P}' \equiv \mathbf{P} \wedge \mathbf{Obj}(\mathbf{P}') = min\{\mathbf{Obj}(\mathbf{P}'') \mid \mathbf{P}'' \equiv \mathbf{P}\}$ (if **Tout** not reached) $\vee$ $\mathbf{Obj}(\mathbf{P}') \leq \mathbf{Obj}(\mathbf{P})$ (if **Tout** reached)

```
1  Superoptimization(P,Obj,Tout)
2    S ← GenerateBlocks(P)
3    NewSq ←[ ]
4    for s ∈ S do
5      optimize← PredictOptimizable(s,Obj)
6      if optimize then
7        ini, fin ← Symbolic(s)
8        ub, lb ← ComputeBounds(s, ini, fin)
9        sz ← PredictBlockSizeBound(s,lb,ub,Obj)
10       sol, g ← SearchOptimal(ini, fin,Obj,Tout,sz)
11       if g > 0 then
12         NewSeq ← NewSeq.append((sol,g))
13       else
14         NewSeq ← NewSeq.append((s,0))
15     else
16       NewSeq←NewSeq.append((s,0))
17   P′,G ← BuildOptimizedCode(NewSeq)
```

Algorithm 1: GASOL+ML: Superoptimization using Deep Neural Networks. The framed instructions are our contributions to the superoptimization algorithm in GASOL. First, the smart contract is divided into sequences in `GenerateBlocks`. Next, a model determines whether a block can be further optimized. If so, `Symbolic` performs symbolic execution to produce a representation of the initial and final states, and `ComputeBounds` generates safe lower and upper bounds on the number of instructions for the search. A second model, used in `PredictBlockSizeBound`, infers a more precise upper bound for the search, which is then carried out in `SearchOptimal` within a specified timeout. Once the search concludes or the timeout is reached, the resulting sequence is incorporated into the optimized program if it improves **Obj**. Finally, in `BuildOptimizedCode`, a smart contract is constructed from the optimized sequences.

Algorithm 1 outlines the implementation of the Ethereum superoptimization algorithm in GASOL [12]: the framed instructions were not part of the existing algorithm but are rather novel components of our neural-guided extension (named GASOL+ML in what follows).

Input–output. The input to the basic superoptimization algorithm (Algorithm 1) is a smart contract **P** (either in Solidity source code or compiled EVM bytecode, component ① in Fig. 1) to be optimized according to the selected objective function **Obj** (that can be **gc** or **sb**, see Section 2). This function is represented by component ⑤ in Fig. 1. The output will be an optimized bytecode **P′** together with the gains achieved by superoptimization **G** (component ⑩ in Fig. 1). In order to use superoptimization in a real setting, an additional parameter **Tout** is used to end up the search process when such timeout is reached. As state-of-the-art SMT solvers (e.g., Z3 [22], OMS [23], Barcelogic [24]) are able to return the best solution found within the timeout, the algorithm ensures the **better** property ($\mathbf{Obj}(\mathbf{P}') \leq \mathbf{Obj}(\mathbf{P})$) if there are sequences that reach the timeout and **optimality** for those that complete the search before the timeout is reached. This means that the algorithm is incomplete for optimality (ii) but keeps the correctness guarantee in (i).

Case study. The smart contract *Seaport*[2] [25], which is part of our evaluation testbed (Section 5), will be used in this section to explain the techniques and results. It is written in Solidity and models a marketplace protocol to manage NFTs. Its public functions have been executed in 13,124,146 transactions.[3]

`GenerateBlocks`. This procedure, invoked at Line 2, generates loop-free sequences of instructions from the blocks of the CFG obtained from the EVM bytecode as defined in Definition 1. We refer to [26–29] for EVM CFG generation. As it can be seen in the definition below, the blocks are obtained from the partitions induced by jump-related EVM instructions (`JUMP`, `JUMPI`, `JUMPDEST`). These jump instructions are not kept in the sequence but rather recalculated and integrated later by `BuildOptimizedCode`. Each block may be split into sub-blocks (and one sequence is produced for each sub-block) due to instructions that cannot be relocated to ensure the original semantics of the block (e.g., `GAS`, `CALL`, `LOGX`, `CREATE`, `CODECOPY`, among others). This procedure corresponds to component ② in Fig. 1.

Definition 1 (*Block-Generation*)**.** Given a sequence of instructions $I = [i_0, i_1, \ldots, i_n]$, we define its block-partitioning as follows:

$$\text{blocks}(I) = \left\{ I_x \equiv i_x, \ldots, i_y \;\middle|\; \begin{array}{l} (\forall k.x < k < y, i_k \notin \mathit{Jump} \cup \mathit{Terminal} \cup \mathit{Split} \cup \\ \{\texttt{JUMPDEST}\}) \wedge (\; x = 0 \vee i_{x-1} \in \text{Split} \cup \{\texttt{JUMPDEST}\}\;) \wedge \\ (\; y = n \vee i_{y+1} \in \text{Jump} \cup \text{Split} \cup \text{Terminal}\;) \end{array} \right\}$$

where

$$\begin{aligned} \text{Jump} &= \{\texttt{JUMP}, \texttt{JUMPI}\} \\ \text{Terminal} &= \{\texttt{RETURN}, \texttt{REVERT}, \texttt{STOP}, \texttt{INVALID}\} \\ \text{Split} &= \{\texttt{LOGX}, \texttt{CALL}, \texttt{GAS}, \texttt{DELEGATECALL}, \texttt{STATICCALL}, \\ &\quad \texttt{CALLDATACOPY}, \\ &\quad \texttt{CODECOPY}, \texttt{EXTCODECOPY}, \texttt{RETURNDATACOPY}\} \end{aligned}$$

Given a smart contract **P**, and let I be the sequence of instructions of its EVM bytecode, `GenerateBlocks`(**P**) returns the set of blocks $\mathbf{S} = \mathit{blocks}(I)$, as described in Definition 1.

Example 1 (*Generation of Sequences*)**.** The EVM bytecode of *Seaport*, compiled from the main contract (the root one), has 17,030 EVM opcodes contained in 1420 blocks. The next fragment of opcodes leads to three basic blocks according to Definition 1:
$[\texttt{PUSH2 0x2dc4 PUSH2 0x4ea0}]_{\text{Block1}}$ `JUMP`
`JUMPDEST` $[\texttt{SWAP2 ISZERO ISZERO PUSH1 0x20 SWAP3 DUP4 MUL SWAP2 SWAP1 SWAP2 ADD SWAP1 SWAP2 ADD MSTORE DUP1 MLOAD PUSH1 0x60 ADD MLOAD DUP1 MLOAD PUSH1 0x00}]_{\text{Block2}}$
`JUMPDEST` $[\texttt{DUP2 DUP2 LT ISZERO PUSH2 0x2e44}]_{\text{Block3}}$ `JUMPI`
beginning and end of blocks are marked between brackets.

`Symbolic`. The next step is to generate symbolic structures that capture the effect of executing a sequence of instructions. These structures can be later used in the search for optimal solutions. For this purpose, we introduce a symbolic structure to represent how the stack is affected and which memory accesses occur after executing certain instructions as follows. A symbolic state, `state := (stack, mem)`, consists of two components: a symbolic stack (`stack`) and symbolic memory (`mem`), which includes store operations both in the memory and storage. The symbolic execution step begins with an initial symbolic state, `ini := (istack, [])`, where `[]` indicates that no memory access occurs initially. We statically compute the minimum number of stack elements required to execute all the instructions, and assign a distinct stack variable s_i to each element in `istack`. Using this state, we compute `fin := (fstack, fmem)`, which represents the final stack elements and memory accesses after the symbolic execution over `ini`.

Example 2 (*Symbolic*)**.** For Block 2, we determine that at least four elements are required initially in the stack. Thus, we construct the

[2] Deployed on 2022-06-11 21:19:20 on block 14,946,474.

[3] Statistics on 2023-05-10.

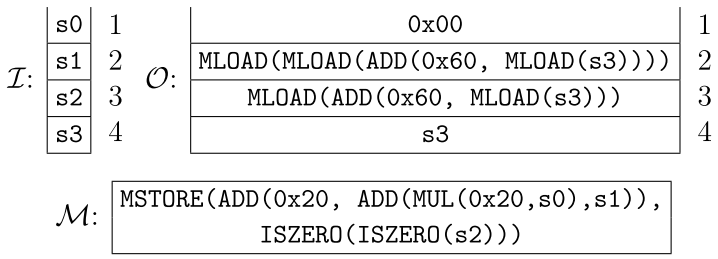

Fig. 2. Resulting state after performing symbolic execution of the sequence *Block2* in Example 1. The state comprises three components: the initial symbolic stack $\mathcal{I}$, the resulting output stack $\mathcal{O}$, and the symbolic memory $\mathcal{M}$.

symbolic stack before executing the sequence $\mathcal{I}$. Through symbolic execution of $\mathcal{I}$, we obtain the output stack $\mathcal{O}$ and symbolic memory $\mathcal{M}$ in Fig. 2.

At this stage, we apply simplification rules to simplify the expressions. Additionally, we perform a *flattening* step, ensuring that every computation in our symbolic structures is associated with a unique stack element s_i. A complete formalization of this process can be found in [30].

Example 3 (*Flattening*)**.** In the previous example, no simplification rules apply. After flattening, the resulting state is ([s4, s9, s7, s3], [MSTORE(s13,s15)]), where different terms and subterms to variables have been mapped through the following transformation:

{s4 ↦ 0x00, s5 ↦ 0x60, s6 ↦ MLOAD(s3),
s7 ↦ ADD(s5,s6)s8 ↦ MLOAD(s7),
s9 ↦ MLOAD(s8), s10 ↦ 0x20,
s11 ↦ MUL(s10,s0), s12 ↦ ADD(s11,s1),
s13 ↦ ADD(s10,s12), s14 ↦ ISZERO(s2), s15 ↦ ISZERO(s14), }

Note that *flattening* allows us identifying elements that must be duplicated. For instance, element s7 appears both as part of the term s8 and in the final stack.

ComputeBounds. As described in Definition 2, for the size-bound (Line 8 in Algorithm 1), the length of the original solution may be used as upper bound **ub**. Note that more refined bounds can be obtained if some simplifications [12,31] are applied over the original sequence of instructions. These simplification rules capture the semantics of the instructions taking advantage of algebraic identities. Regarding the lower bound **lb**, it can be inferred from the flattened symbolic structures.

Definition 2 (*Upper-Bound*)**.** Given a sequence of instructions $I = [i_0, i_1, \ldots, i_n]$, we define its upper bound as:

$$ub(I) = min(n+1, n+1-o)$$

where o = len(optimize_sequence(I)). Hence, o is the length of the sequence of instructions obtained by applying the simplification rules defined in [12,31] to I.

Definition 3 (*Lower-Bound*)**.** Given the initial state (istack, []) and the final *flattened* state (fstack, fmem) with the mapping map obtained from a sequence of instructions I, we define the number of times an element is used as:

$$used_I(s_i) = \#(s_i, \texttt{fstack}) + \#(s_i, \texttt{fmem}) + \#(s_i, \texttt{map}) - \#(s_i, \texttt{istack})$$

The function # counts the occurrences of an element in each structure. For map, it counts the appearances of s_i in right-hand side terms.

If S_I represents the set of all stack variables from the initial and final flattened states, the lower bound can be computed as:

$$lb(I) = \sum_{s_i \in S_I} |used_I(s_i)| + len(\texttt{fmem}).$$

Note that we take the absolute value of *used* because some initial elements may not be used elsewhere, resulting in negative values. As we want to remove these elements with POP, our formula considers these instructions as well. Moreover, store accesses are not included in the mapping and must be counted separately. Informally, our lower bound accounts for the operations required, the duplication of stack elements, and the POP operations needed to discard unused elements. The following example illustrates this computation.

Example 4 (*Size-Bound*)**.** The initial length of the sequence of instructions for Block 2 is 23 and we cannot apply any simplification rule based on the semantics of the instructions. Therefore, according to Definition 2, we use 23 as **ub** for Block 2. The value for **lb** inferred by GASOL is 16, derived from the following values of *used*:

$$\text{used}(s_i) = \begin{cases} 2 & \text{if } i \in [3,7] \\ 0 & \text{if } i \in [0,1,2] \\ 1 & \text{otherwise} \end{cases}$$

and the formula in Definition 3:

$$\sum_{s_i \in S_I} |\text{used}_I(s_i)| + \text{len}(\texttt{fmem}) = |2*2| + |0*3| + |1*11| + 1 = 16$$

The informal reasoning to infer **lb** = 16 from Example 2 is as follows: The element at 1 in $\mathcal{O}$ only needs 1 opcode to be computed. Element 2 needs 6 opcodes (those 4 that appear explicitly plus a PUSH for the constant and a duplication for s3). Element 3 only needs one opcode to be duplicated, as it has been computed previously at position 2 of $\mathcal{O}$. s3 at position 4 does not add any opcode as it is already in $\mathcal{I}$. For the symbolic memory, we need 1 opcode for MSTORE, 5 opcodes to build the first argument (elements s0 and s1 are already in $\mathcal{I}$ and do not appear in $\mathcal{O}$) and the second argument needs 2 more.

Therefore, procedure ComputeBounds implements component ④ in Fig. 1 and produces **ub** and **lb** which are used by ⑥ and ⑦.

SearchOptimal. Finally, the optimal solution **sol** (within the size-bounds given) and the gains **g** are automatically obtained by means of an SMT solver invoked in Line 10 (corresponds to component ⑧ in Fig. 1). When ML is not used, SearchOptimal uses **ub** and **lb** instead of **sz** to bound the search from above and below. Definition 4 shows a high-level description of the encoding used by a SMT solver to find the optimal solution. The encoding captures the effect of selecting a computation to perform at each possible execution step as *hard constraints* while the objective function is encoded as *soft constraints*. For each sequence of instructions, given its initial stack, its final stack, and the mapping map with the set of possible computations carried out in the block (that is followed from the final symbolic state and its associated mapping), we have to encode the following: (i) S_V contains the representation of the stack and the elements that it contains during the execution of the block. We use existentially quantified variables to express the word stored at each stack position and propositional variables to represent the utilization of the stack, as not all the positions are used at each execution step. (ii) C_I denotes the encoding of the EVM instructions. (iii) B contains the constraints that represent how

the stack at the beginning is and, (iv) E describes how the stack at the end is. Finally, (v) C_{COST} represents the cost of the instructions we select, as the cost of the solution can be expressed in terms of the cost of each individual instruction (both for gas and size criteria). The solution has to satisfy the *hard constraints*, ensuring an equivalent solution, (they correspond to S_V, C_{INS}, B, and E in Definition 4) and minimize the soft constraints, seeking for minimal cost (constraints represented by C_{COST} in Definition 4. For instance, a DUPX instruction at step j introduces in its encoding (among others) the hard constraint: $x_{0,j+1} = x_{X-1,j}$, where $x_{i,j}$ represents the content of the stack position i after executing the opcode at position j in the sequence, indicating that the content of the top of the stack after step j contains the element located at position X-1 of the stack. Its soft constraint is $(t_j = \texttt{DUPX}, w)$ where w is the cost returned by **Obj** if opcode DUPX is selected at step j (3 for **gc** and 1 for **sb**). Arithmetic and bit-wise operations are treated as uninterpreted functions and hence optimality results in GASOL are given for the stack-manipulating opcodes (and within the considered size-bounds). All details of the encoding can be found in [12,19].

Definition 4 (*(High-Level) Description of SMT Encoding*)**.** Given the initial stack ini and the final stack fin for a sequence of instructions I, the mapping map obtained from the symbolic execution, and the bound over the length of the solution, the encoding has this form:

$$O = S_V \wedge C_I \wedge B \wedge E \wedge C_{COST}$$

Example 5 (*Optimal Solution*)**.** For Block 2 in Example 1 and with the previous **ub** and **lb** bounds (using objective **sb**), procedure SearchOptimal timeouts in 20 s when searching for the optimal solution. However, it will be able to find a solution when a smaller size-bound is predicted by procedure PredictBlockSizeBound.

BuildOptimizedCode. The last step is reconstructing the bytecode after superoptimization. We replace each block where the optimized version achieves further gains. Since the superoptimization process preserves block equivalence, this translation is straightforward. Nevertheless, there are certain EVM instructions which depend directly on bytecode positions and can be affected by the optimization. For example, JUMP instructions require the specific bytecode of the target address, meaning that changes in bytecode size can misalign jump destinations. To address this, we optimize not directly the EVM bytecode but rather the "EVM Assembly Format" used by the Solidity compiler solc. In Section 6, we further discuss how using this format, together with solc, enables accurate reconstruction of the optimized bytecode.

4.2. Encoding EVM blocks for learning

The first problem one encounters when applying ML is how to encode the input data. In our case, the input is an EVM block as described in Section 4.1, and thus it is natural to encode it as a sequence of tokens where each token corresponds to an opcode. The choice of this representation is also influenced by the use of recurrent neural networks (RNNs) as we will see later.

Tokens are numbers between 1 and 130 (we have 130 different opcodes as the instructions that induce partitioning, see Section 4.1, are excluded from the block), in addition to 0 that is used for padding sequences to make them of the same length during training. An exception are the PUSHX opcodes (for $1 \leq \texttt{X} \leq 32$) that take as operand a value that fits in X bytes. We have considered several approaches to encode these numbers. The first approach (E1) encodes an operand using its corresponding sequence of digits. This, on the one hand, might result in long sequences (numbers in EVM are of 256 bits) which might affect the precision of the learning process, and, on the other hand, is not really needed since optimizations are rarely affected by the actual values, but rather by the fact that several PUSHX instructions use the same value. This leads us to a second approach (E2) that first replaces each operand by a corresponding constant C_i, where different occurrences of the same number are encoded to the same C_i, and then encodes C_i as the sequence of the digits of i (with a special leading token that corresponds to '#'). This requires adding 11 more tokens, and thus in total we have 141 tokens. The third approach (E3) that we have considered discards all operands, which is reasonable since PUSHX already carries information on the number of bytes (X) used to represent the operand.

4.3. The training (and validation) data set

For training we downloaded the last 5,000 verified contracts from Etherscan [32] on three different dates: (July 22, 2022), (January 12, 2023) and (March 14, 2023). From these 15,000, we removed those that were compiled with a version of the Solidity compiler solc lower than 0.8, the ones that raised a compilation error and those that were duplicated, i.e., those whose runtime compiled bytecode generated by the compiler is the same. This resulted in 13,195 smart contracts that contain 9,861,151 blocks. We have compiled and optimized them with the version 0.8.19 of solc[4] and using the flag ––optimize in order to assess the additional gains of applying superoptimization as a second optimization layer, i.e., on code that has been already optimized by the compiler. After removing duplicates (according to E2 representation), we have remained with 95,140 blocks (see Section 5 for details on duplicates). Out of these 95,140 blocks, we only use those for which GASOL either proved that the block is already optimal or succeeded to optimize it: (1) for **gc** we have 57,583 EVM blocks; and (2) for **sb** we have 57,233 EVM blocks. Every input block is labeled with the following data: if the block is already optimal (for the corresponding criteria); the number of opcodes in the corresponding block generated by GASOL; and the corresponding **lb**.

Note that although we call them training sets, they will actually be used for training and validation. Testing is done comparing the performance of GASOL and GASOL+ML on a different set of contracts (see Section 5), which we also use to evaluate the models in Sections 4.4 and 4.5.

4.4. Learning a model for PredictOptimizable

Our aim is to develop a model, based on deep neural networks, that predicts if a given EVM block s is already optimal or not (*Challenge 1* in Section 1). This is a classical binary classification problem, where the input is a sequence of tokens (see Section 4.2) and the output is class 0 (for *already optimal*) and class 1 (for *not optimal*).

The general structure of the neural network that we use to learn such a model is depicted in Fig. 3. The first layer is an embedding layer that maps tokens into vectors in $\mathbb{R}^d$. This layer can also be replaced by a straightforward embedding that maps each token t into 1-hot vector of dimension that is equal to the number of tokens (referred as 1-hot embedding). The next layer is a RNN layer (e.g., LSTM [14] or GRU [33]), followed by a dropout layer which is effective for regularization (it basically zeroes some of the elements of the input tensor with probability p). This is followed by several hidden linear layers (with a corresponding ReLU activation function), which are followed by a final output linear layer with two output channels c_0 and c_1. Given values for c_0 and c_1, the predicted class is 1 if $\frac{e^{c_1}}{e^{c_1}+e^{c_0}} > p$ for a given probability threshold p; otherwise 0. During training we fix p to 0.5, and when using the model we can adjust p to increase the confidence in the answer.

We have tried with several settings that use different encodings for the input and different parameters for the model above. The one that achieved the best performance is as follows: the blocks are encoded using E2, the embedding layer is a 1-hot vector of dimensions 141,

[4] The latest version released up to April 2023.

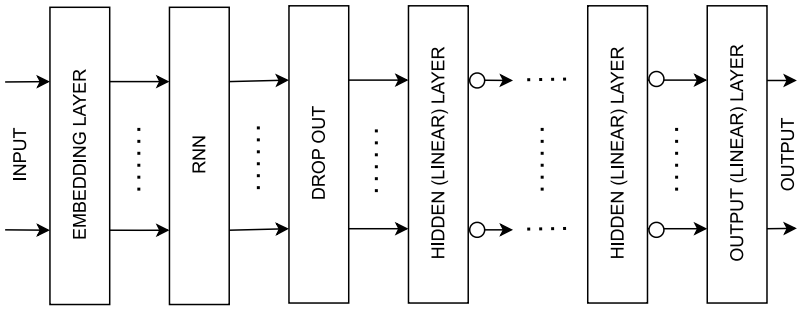

Fig. 3. The neural network architecture.

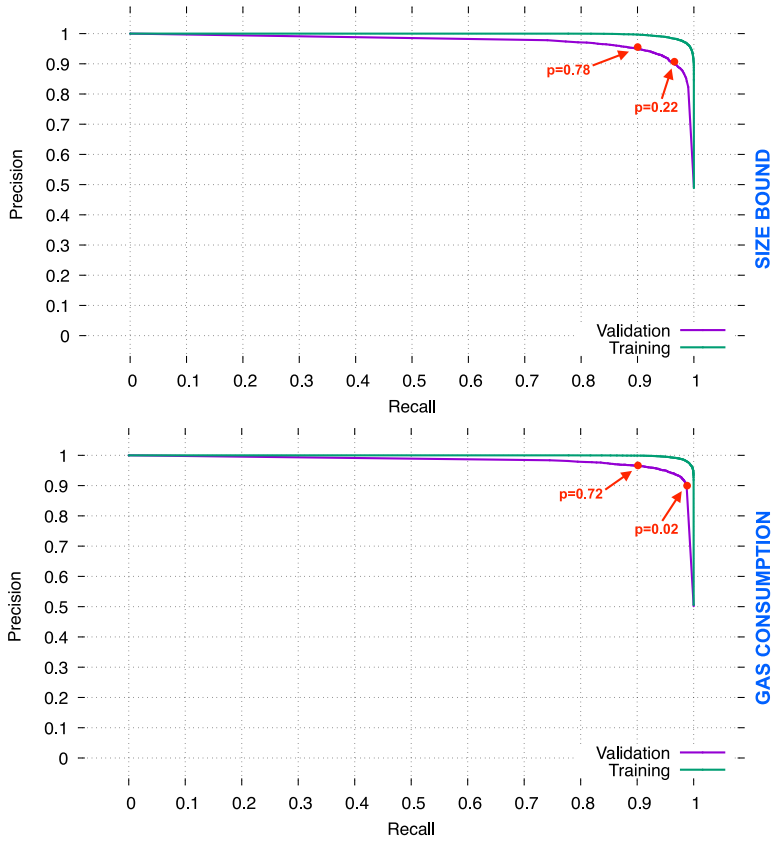

Fig. 4. The precision–recall curves for size and gas consumption. The Y-axis corresponds to *precision*, which is the number of 1 answers that are correct out of all 1 answers. The X-axis is the *recall*, which is the probability that a block labeled as 1 will be answered as 1.

the RNN used is LSTM, the dropout probability is 0.5, and we have 2 hidden layers with 128 channels (for input and output). Based on this setting, we have learned two models to predict if a block is optimal: one *wrt.* **sb**, and one *wrt.* **gc**. The learning loop uses the *Cross Entropy* loss function, an optimizer based on the *Adam algorithm*, and a learning rate of 0.001. We have used 80% of the data sets for training, and 20% for validation (the data sets that we have are quite balanced, with about 50% in each class).

To evaluate the performance of the models, we provide their corresponding Precision–Recall (PR) Curve for class 1 in Fig. 4. This graph is based on 101 samples for different values of the probability threshold p ($p_i = i/100$ for $0 \leq i \leq 100$). A good model (and probability threshold) is one that has a high *recall* and high *precision* at the same time.

For the case of **sb**, the training PR curve is almost ideal, and the validation PR curve is quite good since the recall and precision are both above 0.9 for all probability thresholds between $p = 0.22$ and $p = 0.78$, and for $p = 0.5$ we have 0.92 precision and 0.94 recall. For the case of **gc**, we have a similar situation. The training PR curve is almost ideal, and the validation curve is quite good since the recall and precision are both above 0.9 for all probability thresholds between $p = 0.02$ and $p = 0.72$, where for $p = 0.5$ we have 0.95 precision and 0.93 recall. We have also applied the model to the testset of Section 5 (only to those that can be handled by GASOL without ML, i.e., can be labeled), after eliminating repeated blocks and blocks that already appeared in the training/validation set. We got a precision and recall very similar to that of the validation set.

To summarize, procedure `PredictOptimizable`($\mathbf{s}$, **Obj**) implements component ⑥ in Fig. 1 and is as follows: encode the block $\mathbf{s}$ using E2, apply the corresponding model to obtain c_0 and c_1, and then return 1 if $\frac{e^{c1}}{e^{c1}+e^{c0}} > p$ for the given probability threshold p (controllable by the user); otherwise 0.

Example 6. Given the blocks of Example 1, `PredictOptimizable` classifies Block 1 and Block 3 as already optimal and Block 2 as optimizable. When considering the 1420 blocks of Seaport, 1098 of them are classified as optimal while the remaining 322 are considered

Table 1
Precision of the bound model for **sb** and **gc**. Each table has two blocks, one for training and one for validation. Each block has 4 columns: (#) the number of blocks with the corresponding d_s; (=) the number of cases that were predicted exactly; (>) the number of larger predictions; and (<) the number of smaller predictions.

d_s	Training				Validation				Training				Validation			
	#	=	>	<	#	=	>	<	#	=	>	<	#	=	>	<
0	23,800	23,017	783	0	5915	5581	334	0	27,816	27,413	403	0	7002	6760	242	0
1	10,861	9,673	463	725	2719	2190	256	273	9,577	8,641	449	487	2342	1939	234	169
2	4,095	3,027	358	710	991	542	161	288	2,984	2,343	282	359	770	452	148	170
3	2,412	1,728	279	405	610	278	119	213	1,695	1,229	223	243	442	194	93	155
4	1,720	1,230	257	233	420	192	84	144	1,187	857	151	179	292	122	53	117
5	1,187	882	228	77	349	169	79	101	835	599	90	146	210	74	26	110
6	842	625	173	44	201	86	55	60	690	517	56	117	161	61	27	73
7	475	373	76	26	120	54	23	43	425	300	33	92	102	34	14	54
8	295	206	71	18	79	39	15	25	249	125	44	80	58	8	10	40
9	150	85	49	16	48	16	10	22	144	83	31	30	26	11	0	15
10	84	52	25	7	33	12	8	13	81	48	21	12	18	7	2	9
11	81	57	14	10	17	7	2	8	59	38	5	16	16	3	1	12
12	49	28	0	21	12	5	0	7	33	17	0	16	5	2	0	3
13	11	0	0	11	2	0	0	2	8	0	0	8	3	0	0	3
14	3	0	0	3	1	0	0	1	1	0	0	1	0	0	0	0
15	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0
	sb								**gc**							

optimizable with **sb**. From the 1098 blocks, GASOL is able to optimize 16 of them (1.45%), saving 24 additional bytes. Hence, GASOL+ML loses 5.71% of the savings. However, thanks to the learning GASOL+ML greatly improves performance (saving 3962.56 s out of the 6568.90 s needed by GASOL). If we select **gc**, we obtain that 1183 blocks are classified as optimal, and 237 optimizable. 26 blocks out of the 1183 optimal (2.19%) are optimized by GASOL, losing 125 gas in GASOL+ML from the 1022 saved gas. Importantly, GASOL+ML reduces the time to almost one third (from 6207.75 s to 2353.84 s).

4.5. *Learning a model for* `PredictBlockSizeBound`

Our aim is to develop a model, based on deep neural networks, that for a given EVM block **s** predicts the size n_s of a corresponding optimal block (*Challenge 2* in Section 1). This is a regression problem, where the input is a sequence of tokens (see Section 4.2) that correspond to **s**, and the output is the desired size n_s. Instead of building a model that directly predicts n_s, we build one to predict $d_s = n_s - \texttt{lb}(\mathbf{s})$ where $\texttt{lb}(\mathbf{s})$ is a lower bound **lb** at Line 8 of Algorithm 1. Then, if the model answers d_s for a block **s**, we can use $d_s + \texttt{lb}(\mathbf{s})$ as a prediction for the size of the optimal block. The advantage of predicting d_s instead of n_s is that its set of possible values is much smaller than that of n_s, e.g., in our training set it reduces the possible values from [1..54] to [0..18], where 97% of the blocks have $0 \le d_s \le 5$. This improves the precision dramatically in practice. Another important problem that we faced is that the prediction for d_s is not necessarily an integer, and thus we need to convert it to an integer value using $round(d_s)$, $\lfloor d_s \rfloor$, or $\lceil d_s \rceil$. In what follows we use $round(d_s)$, which achieves better predictions when compared to the others.

The general structure of the neural network that we use to learn such a model is as the one used in Section 4.4 (see Fig. 3), but the output layer has only one output channel d_s. We have tried with several settings that use different encodings for the input and different parameters for the model above. The one that achieved the best performance is as follows: the blocks are encoded using E2, the embedding layer maps tokens into vectors in $\mathbb{R}^{64}$, the RNN used is GRU, the dropout probability is 0.5, and we have 2 hidden layers with 128 channels (for input and output). Based on this setting, we have learned two models to predict d_s: one *wrt.* **sb**, and one *wrt.* **gc**. The learning loop uses the *Mean Square Error* loss function, an optimizer based on the *Adam algorithm*, and learning rate 0.001. We have used 80% of the data sets for training, and 20% for validation.

The performance of the models on the training and validation sets is as follows. For the case of **sb**, the model succeeded to correctly predict d_s for 80% of the blocks in the validation set (the average loss is 0.4) and 89% for the training set (the average loss is 0.1). For **gc**, the model succeeded to predict d_s for 84% of the blocks in the validation set (the average loss is 0.5) and 92% for the training set (the average loss is 0.08). We have also applied the model to the testset of Section 5 as described in Section 4.4. We got similar results, in particular it correctly predicted d_s for 85% of the blocks for **sb** (the average loss is 0.3), and 85% for **gc** (the average loss is 0.5).

In Table 1 we give detailed information on the predictions mentioned above, for each value of $d_s \in [0..18]$. Recall that we round the prediction to the nearest integer. We can see that the precision is very high for small d_s values, and decreases for larger ones. This is explained by the fact that the training set does not have enough cases for large values of d_s. The "problematic" predictions are those in the column <, since they lead to a bound smaller than the optimal size and thus GASOL+ML would spend time looking for an optimal block but it will not find one. Those in column > will improve the performance of GASOL+ML when they are smaller than the bound that GASOL uses by default, and they will never diminish performance of GASOL+ML because they are simply discarded in such case (Line 9 of Algorithm 1).

To summarize, procedure `PredictBlockSizeBound`(**s**, **lb**, **ub**, **Obj**) implements component ⑦ in Fig. 1 and is as follows: encode the block **s** using E2, apply the corresponding model to obtain d_s, and return $min(\mathbf{ub}, \mathbf{lb} + round(d_s))$.

Example 7. The bound **sz** returned by `PredictBlockSizeBound`(**s**, 16, 23, **sb**) for Block 2 of Example 1 is 22. This small difference between **ub** and **sz** avoids reaching the timeout of 20 s, as the search takes 12.01 s. The solution returned is: `PUSH1 0x20 MUL ADD PUSH1 0x20 DUP4 SWAP3 ISZERO ISZERO SWAP2 ADD MSTORE MLOAD PUSH1 0x60 ADD MLOAD DUP1 MLOAD PUSH1 0x00` that saves 4 bytes in size (as it removes 5 `SWAPX` instructions from the initial block and transforms a `DUP` instruction into a `PUSH1`) and 15 gas.

Considering all blocks of Seaport using **sb**, there are 15 blocks for which GASOL reaches the timeout limit. In addition, in 9 of them GASOL is not able to find a solution. However, GASOL+ML is able to handle them, finding a solution for the 9 blocks, and proving the optimality for the remaining 6. For all of them, the predicted bound **sz** is smaller than **ub**. Indeed, it holds for 187 blocks. For 1209 blocks, **sz** and **ub** have the same value. In 13 blocks, the predicted bound **sz** is too

small making the SMT encoding unsatisfiable, and losing 10 bytes that were saved by GASOL. Despite this fact, GASOL+ML is able to save 96 bytes more than GASOL.

5. Experimental evaluation

This section reports on GASOL+ML, our implementation of NGS within the GASOL system [12]. GASOL[5] is implemented in Python and uses OptiMathSAT [23] version 1.7.3 as Max-SMT solver. The new ML-layout has been built on top of GASOL and it allows executing the two learned models described in Section 4. Our ML-layout is also implemented in Python using `PyTorch` [34] and its source code for the ML modules as well as the smart contracts analyzed are available in a separate repository.[6] To experimentally assess the efficiency/effectiveness and impact of NGS, we downloaded (using BigQuery [35] and Etherscan [32]) two different sets of contracts:

1. The 100 most-called contracts deployed on Ethereum (hence the most relevant ones to be optimized) whose source code was available and were compiled with version 0.8 of `solc`
2. 1000 contracts randomly selected from the contracts deployed throughout 2023, whose source code meets the same requirements as for the previous set.

We compiled them again using the same setting as for the training set of Section 4.3. Then, we optimized these contracts combining GASOL with the different ML models for both objective functions. All experiments have been performed on an AMD Ryzen Threadripper PRO 3995WX 64-cores and 512 GB of memory, running Debian 5.10.70. In what follows we refer to the model of Section 4.4 as the *classifier*, and the model of Section 4.5 as the *bound predictor*.

Our experimental evaluation addresses four research questions:

RQ1: Which gains does NGS bring over plain superoptimization?
RQ2: How should GASOL+ML be leveraged to an industrial setting using the introduced techniques?
RQ3: Which type of blocks are filtered by the classifier? How is the outcome affected by the bound predictor?
RQ4: What is the impact of NGS in the Ethereum blockchain?

To address **RQ1**, Figs. 5 and 6 show the overall results when optimizing *wrt.* **sb** and **gc** resp. for different combinations of models, timeouts and datasets. The results are similar for both datasets, thus we are focusing on the results for dataset (1). These figures include the time spent in minutes (left) and the gains in number of bytes and units of gas (right) resp. that correspond to **g** in Line 10 of Algorithm 1. In these figures, we analyze different model configurations (enabling the bound predictor or the classifier, both or neither), alongside different timeout settings. These settings follow the formula $N \cdot (1 + \#\texttt{STORE})$ seconds, which is the default timeout formula in GASOL (since in many cases, `STORE` opcodes define subsequences that turn out to be independent *wrt.* the optimization). Moreover, the top x-axis and in-bar percentages compare the gains and overhead relative to the default configuration used in GASOL, $(t = 10\text{ s}, \emptyset)$; and with the configuration with the same timeout and no model enabled $(t = N s, \emptyset)$, resp. For instance, Fig. 5 (upper-left) shows that configuration $(t = 2\text{ s}, \texttt{b+opt})$ for dataset (1) reduces the overall time to 8.04% compared to the configuration $(t = 10\text{ s}, \emptyset)$ for **sb**, which can be followed from the corresponding bar being between $5 - 10\%$ *wrt.* the top axis; while reducing it to 26.4% compared to $(t = 2\text{ s}, \emptyset)$.

By looking at the percentages inside the bars, we see that the configurations that include the classifier reduce the overall time to at least to 40% (for size) and a half (for gas) while preserving more than 95% of the gains. The impact the bound predictor has is more subtle, but yet very prominent. The gains in categories that only include `b` (green bars) are always over 100% compared to the configuration with no model and same timeout. This happens because, in general, with the original upper-bound **ub**, the solver reaches the time out with a worse solution (if any) than when using a smaller ML-bound **sz** (since the search space is larger). For smaller timeouts, this is even more relevant because the timeout is reached earlier. Thus, to answer **RQ2**, our approach is to decrease the default timeout in GASOL and include the new ML techniques, similar to setting $(t = 2\text{ s}, \texttt{b+opt})$. The median time GASOL spends optimizing a contract in this setting for both datasets is ∼2 min and the maximum is ∼19 min. In the Ethereum ecosystem, we consider them to be reasonable times because of the reasons listed in Section 1.

As already mentioned, we found out that a significant amount of blocks from the evaluation sets already appeared in the dataset of Section 4.3, when in principle they are completely unrelated sets of smart contracts (downloaded in different dates and using different forms of selection criteria, namely the dataset of Section 4.3 are from the three different days and the others are either the most used or contracts throughout 2023). Therefore, in order to check the validity of our evaluation, we reproduced the tables in this section just for the subset of non-repeated blocks and obtained results which are proportionally similar to the ones with repetitions, which confirms the relevance and validity of our experiments. Block repetitions are inherent to the Ethereum setting, mainly due to two reasons. There exist popular standards (e.g., ERC20 [36], ERC721 [37] or OpenZeppelin contracts [38]) that are widely adopted in smart contracts. Besides, `solc` tends to repeat similar block patterns in all contracts to handle specific situations. For instance, similar blocks (according to the encoding of Section 4.2) are used to select which function has been called in a transaction. Our findings show that there are few blocks that are repeated a huge amount of times but lead to no savings and very little overhead, while most gains and overhead comes from the blocks that are repeated fewer times or none. Hence, ML techniques are important to deal with challenging blocks.

Fig. 7 dives further into how the optimization outcomes are affected by the different configurations in order to address **RQ3**. The classifier filters out ∼80% blocks for both criteria. Most filtered blocks are from **Alr**, which decreases from nearly ∼70% to less than 1%. It also reduces the blocks in **Tout** nearly by a third in all configurations. The bound predictor only leads to unsatisfiable SMT problems in 0,4% for **sb** and 0.7% for **gc** configurations, which means very few blocks cannot be optimized due to an unsound bound. For categories $(t = 1\text{ s}, \emptyset)$ and $(t = 1\text{ s}, \texttt{b})$, it also increases significantly the amount of blocks in **Optim**, explaining why savings are more significant in these configurations.

Finally, Table 2 shows the percentage that savings in Figs. 5 and 6 represent *wrt.* the original sizes and an estimation on the gas consumption of all smart contracts resp. for datasets (1) and (2). For instance, for $(t = 10\text{ s}, \texttt{b+opt})$ we have reduced the size of the bytecode by 1.91% in dataset (1) (this is computed as 17,810/930,677*100, where 930,677 is the total size of the bytecode of the 100 contracts). To estimate the overall gas consumption, we have assumed that each instruction is executed once and that relevant instructions whose gas cost is dynamic (e.g., `STORE`) have the most usual gas cost. To assess the impact of the approach, let us observe the savings for **gc** in dataset (1) (0.1–0.2%), as they comprise a large number of transactions. They are as one would expect when optimizing stack-manipulating bytecode as their gas cost is small when compared to other non-stack operations. However, in order to understand their great impact in the overall blockchain, we need to consider that such popular contracts manage millions of transactions. To answer **RQ4**, we have downloaded all transactions from the 100 analyzed contracts as of block 17,226,487[7] using Etherscan's

[5] https://github.com/costa-group/gasol-optimizer.
[6] https://github.com/costa-group/gasol_ml.
[7] Produced on 2023-05-10 12:36:23 AM +UTC.

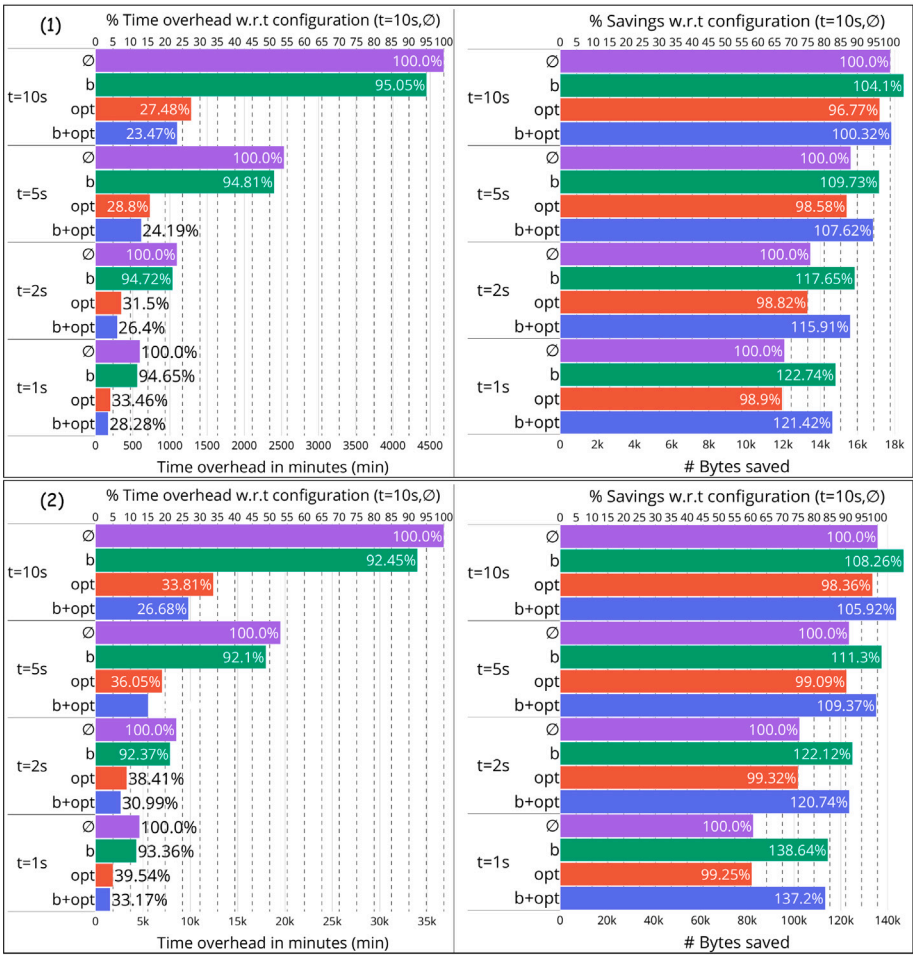

Fig. 5. Time overhead (left) and size savings (right) for optimization *wrt.* **sb** for combinations of timeouts and models (Y-axis). The top row displays the results for dataset (1), while the bottom row shows the results for dataset (2). $t = Ns$ expresses that the superoptimization halts after $N \cdot (1 + \#\texttt{STORE})$ seconds. ∅ (purple) represents no model has been used; in `b` (green) and `opt` (red) the bound predictor and the classifier is used resp. and `b+opt` (blue) denotes the combination of both models. The bottom X-axis represents the absolute value of the corresponding measure and the top X-axis represents the percentage of gains and overhead compared to $(t = 10\ \text{s}, \emptyset)$. Each bar contains an additional percentage that compares the corresponding configuration $(t = Ns, m)$ with $(t = Ns, \emptyset)$.

Table 2
Overall savings for **sb** and **gc** for combinations of timeouts and models with datasets (1) and (2).

	$t = 10$ s				$t = 5$ s				$t = 2$ s				$t = 1$ s			
	∅	b	opt	b+opt	∅	b	opt	b+opt	∅	b	opt	b+opt	∅	b	opt	b+opt
(1) **sb**	1.91	1.99	1.85	1.91	1.68	1.84	1.66	1.81	1.45	1.7	1.43	1.68	1.3	1.59	1.28	1.57
(1) **gc**	0.2	0.21	0.2	0.2	0.18	0.19	0.18	0.19	0.16	0.18	0.15	0.18	0.13	0.17	0.13	0.16
(2) **sb**	1.95	2.11	1.92	2.07	1.78	1.98	1.76	1.94	1.47	1.8	1.46	1.78	1.19	1.65	1.18	1.63
(2) **gc**	0.27	0.28	0.27	0.28	0.25	0.27	0.25	0.27	0.2	0.25	0.2	0.25	0.17	0.22	0.17	0.22

Python API [39], consisting of 41,106,276 transactions. Then we have combined the transaction fee information with the price per Eth in dollars in the corresponding day (downloaded from [40]), resulting in $656.25M spent in transaction fees. Finally, we have removed the costs inherently tied to the transactions (default 21,000 units of gas fee plus the fee for sending the transaction data), which results in a total of $509.1M of execution costs. Assuming these contracts were deployed using setting $(t = 2\ \text{s}, \texttt{b+opt})$, whose estimated gas savings are 0.18%; and considering these savings affect all transactions uniformly, this translates to savings of $1.24M just on these 100 contracts, while the average compilation time is around 3 min. Such a compilation time is not far from the times of state of the art compilers when the higher optimization options are activated on large code. Overall, we argue that our experimental results prove the impact of NGS in the Ethereum industry and its scalability *wrt.* plain superoptimization: it achieves a speedup of 16x (12x for gas) by only losing ∼12% of optimization (∼14% for gas), or a speedup of 3-4x while no optimization is lost.

6. Threats to validity

Concerning *external* threats to validity, there are two questions: (1) whether our results for superoptimizing EVM code using supervised learning generalize to other code unrelated to Ethereum, and (2) whether there are limitations on the EVM code we are able to analyze. Let us start by discussing (1). We distinguish here between the superoptimization framework and the learning add-ons. As regards the former, its constituent components (for generating the loop-free sequences, analyzing and transforming them and searching for the optimal) can be developed for any other languages without requiring conceptual changes. As a matter of fact, [41] applies this framework to superoptimize Wasm code. Regarding the ML add-ons, the main changes would

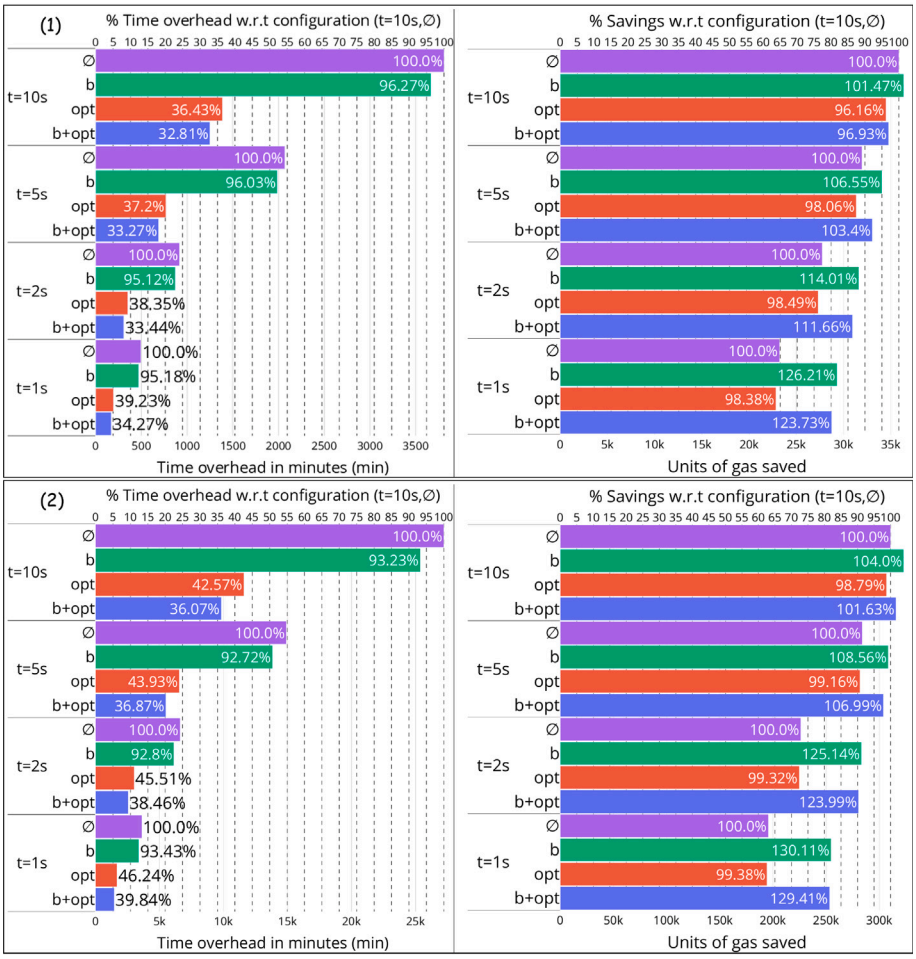

Fig. 6. Time overhead (left) and gas savings (right) for optimization *wrt.* **gc** for combinations of timeouts and models. It follows the same notation as Fig. 5.

be on the encoding of the blocks for the corresponding code being optimized. For other bytecode languages, our approach can be directly used while higher-level languages would require a proper encoding. As regards the optimization objective, both the `PredictOptimizable` and `PredictBlockSizeBound` use the gas consumption or the bytecode-size to construct the models. However, the definition of the models is generic and other cost models could be used without requiring any change to them.

As regards (2), one limitation on the code we can handle could be related to optimizing instructions whose arguments depend on the code itself. However, GASOL accepts "EVM Assembly Format", an intermediate representation used by solc to represent EVM code without instantiating concrete values for instructions whose arguments depend on the code itself (such as jumps or `CODECOPY`). This representation includes all EVM instructions, along with new pseudo-instructions that act as placeholders for these values — similar to the use of labels in low-level languages. We have used this format in the experimental section, as it allows direct comparison with EVM code.[8] For `CODECOPY`, two different instructions are used to express certain parameters: `PUSH [$]`, which is replaced by the offset in the code to copy; `PUSH #[$]`, which is replaced by the size of the run-time code. Additionally, for `JUMP` instructions, the format introduces the concept of tags with numerical identifiers. These tags are unique and can be referenced using the instruction `PUSH [tag] tag_id`. The solc compiler allows reimporting this assembly format without further optimizations by setting the flag "–import-asm-json" in the Command Line Interface; or by providing the assembly code in the Standard JSON format. In addition, the "EVM Assembly Format" stores the metadata code separated from the deployment and runtime code. Relying on this pipeline provides an additional layer of correctness, as we are effectively using the same mechanisms as solc to instantiate these instructions. While this format is specific to the Solidity language, code from other sources can be easily adapted to it and further optimized.

The main *internal* threat concerns possible biases in the selection of contracts, that could influence the quality of models and also the relevance of precision/accuracy in the evaluation phase. However, this threat is mitigated for the following reasons:

- **Non-random selection of data.** The contracts used both for testing and validation (Section 4.3) and those used in the experimental evaluation (Section 5) correspond to contracts deployed in different dates. The contracts used in the training set and validation set have been deployed in three different dates which are 8 months apart. In order to study the impact of our approach in a real setting, we downloaded the 100 most-called contracts deployed on Ethereum whose source code was available and were compiled with version 0.8 of solc. These contracts were deployed between March 2021, 25 and Feb 2023, 14 in 92 different dates. In addition, we have executed our approach on a second data set (see Section 5). We have downloaded from Ethereum all the contracts deployed in 2023 (from Jan 2023, 1 to Dec 2023, 31) whose source code was available and that were compiled with the version 0.8 of solc, and we select 1000 contracts randomly.

[8] The complete list of assembly pseudo-instructions can be found at the following link: https://github.com/ethereum/solidity/blob/develop/libevmasm/Assembly.cpp#L218.

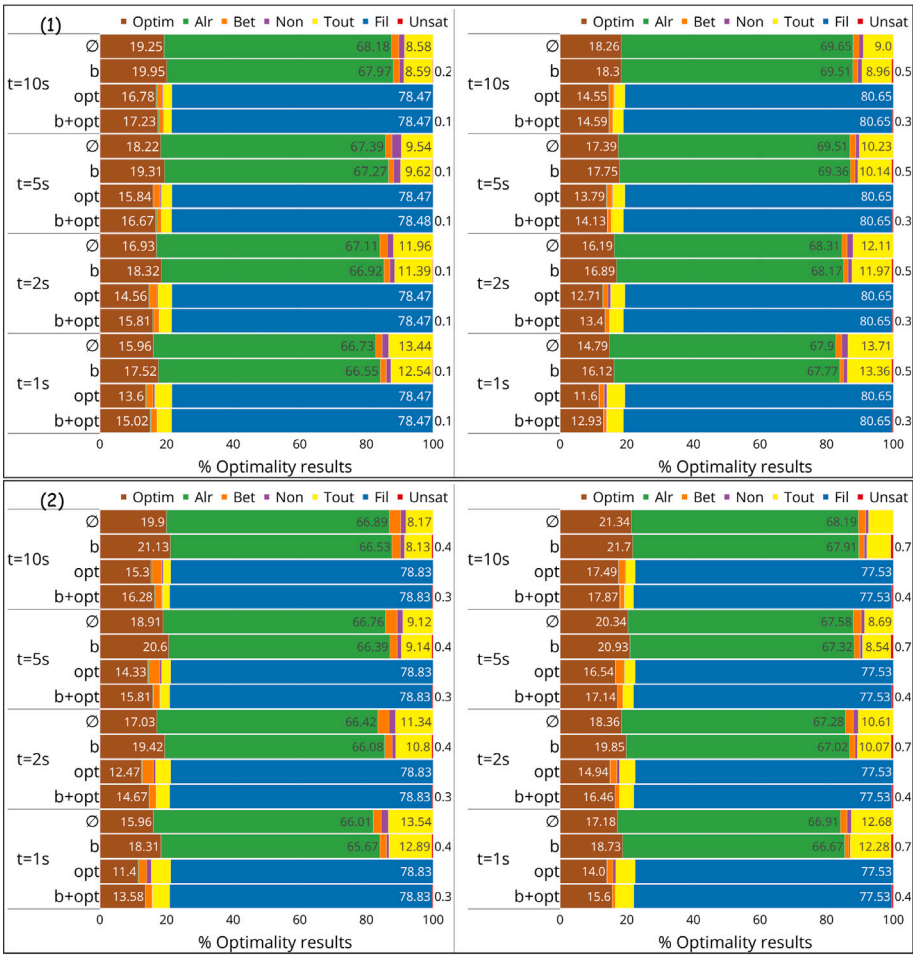

Fig. 7. Optimality results for **sb** (left) and **gc** (right) for combinations of timeouts and models. The results for dataset (1) are shown in the top row, while those for dataset (2) are in the bottom row. $t = Ns$ refers to the timeout formula $N \cdot (1 + \#\texttt{STORE})$. `b`, `opt`, `∅`, `b+opt` denote whether the bound predictor, classifier, neither of them or both are enabled, resp. **Optim** (brown) represents the percentage of blocks that found the optimal solution before reaching the timeout and improved the original block. **Alr** (green) is similar to **Optim** but the optimal solution found did not improve the original block. **Bet** (orange) and **Non** (purple) are similar to **Optim** and **Alr** resp. but the timeout was reached and an intermediate solution was found. **Tout** (yellow) corresponds to blocks that reached the timeout and no solution was found. **Fil** (blue) groups blocks filtered out by the classifier. **Unsat** (red) represents blocks whose bound was too low, leading to an unsatisfiable SMT problem. The concrete percentage is shown inside the bar for those categories with a higher number of blocks, as well as for **Unsat** category just right the bar (as long as it is > 0).

- **Duplicated code.** In order to avoid redundancies, we remove duplicates in two different steps when considering the contracts in the training set (see Section 4.3): (i) we remove those contracts whose runtime compiled bytecode generated by solc is the same and, (ii) we remove those EVM blocks that have the same sequence of instructions (abstracting the corresponding constants of the PUSHX instructions).
- **Compiler version.** The different versions of solc compiler used to compiled the source code of the smart contracts and deployed them on the blockchain may affect to the accuracy of our approach. However, despite the compiler version used to deploy the contracts, we have recompiled them using the same version of solc. Hence, we have described the results obtained by analyzing the runtime bytecode generated with the version 0.8.19 of solc for all the smart contracts.

7. Related work

Massalin [6] coined the term "superoptimization" emphasizing the idea that the technique is applied as a second layer after compilation. It originally assumed that the input program was written in machine language and the optimization function is the length of the code. The technique in [6] was defined as follows: *The search finds the shortest program that computes the same function as the source program by doing an exhaustive search over all possible programs. The search space is defined by choosing a subset of the machine's instruction set, and the op-codes of these instructions are stored in a table. Superoptimizer consults this table and generates all combinations of these instructions, first of length 1, then of length 2, and so on. Each of these generated programs is tested, and if found to match the function of the source program, superoptimizer prints the program and halts.* By that time, there was no soundness guarantee since testing was used to check program equivalence. Denali [42] generalized the original technique to more complex objective functions and proposed a goal-directed approach that encoded the semantics of the original code to guide the search.

The use of SMT solvers in superoptimization was first proposed in [9]. Since then, with the enormous advances on SMT, superoptimization has become a more tractable problem, and numerous tools are emerging, including LLVM superoptimizers [10,18], Ethereum bytecode superoptimizers [11,12], WebAssembly superoptimizer [16], and recently superoptimization for Quantum circuits [43]. The main distinguishing features of the various tools are:

- Souper [18] has investigated the use of dataflow-based techniques to speed up the process;

- ebso [11] has proposed the gas model of Ethereum smart contracts as objective function; and a full SMT encoding of the EVM bytecode semantics that is very hard to scale;
- GASOL [12] has proposed a pre-phase for which the bytecode is first symbolically executed and optimized by applying peephole optimizations so that the SMT search only focuses on the stack operations and performs much more efficiently; item GreenThumb [8] applies different search techniques in parallel.

Another effort to scale superoptimization is [21] which consists in formulating the loop-free superoptimization task as a stochastic search problem. All such previous proposals for speeding up the process are orthogonal and complementary to ours. The use of ML to speed up superoptimization has been also explored in [44]. In their case, learning is used to decide which instructions are the most likely to appear in the optimal solution so that the search is guided to use them. This is fundamentally different from our proposal since we do not use ML in the search but rather to predict external parameters. In the context of bytecode optimization though, the most used instructions are typically stack operations and we do not see much interest in the type of guided search of [44] in this context. ML has been used in combination with program synthesis [45] and static analysis [46]. As these methods differ from supercompilation both in their goals and components, the ML extension differs substantially as well.

8. Conclusions

This article leverages synergies of superoptimization and ML. On the one hand, using ML directly to optimize code (i.e. replacing the exhaustive search that superoptimization does by an ML-predicted optimized code) is unlikely to achieve as good results as using the SMT-search due to the huge variety of code patterns that can appear in real programs. On the other hand, trying to manually predict heuristic components used within a superoptimizer (i.e., size bounds for the optimal solution) will clearly lead to worse results than learning them from the results gathered from thousands of examples. Our synergistic approach highly contributes to transfer our tool to a real industrial context: it achieves a speedup of 12–16x by only losing ∼12 − 14% of optimization (the range varies according to the objective function), or a speedup of 3–4x while no optimization is lost.

CRediT authorship contribution statement

Matheus Araújo Aguiar: Writing – review & editing, Writing – original draft, Validation, Resources, Methodology. **Elvira Albert:** Writing – review & editing, Writing – original draft, Validation, Software, Resources, Methodology, Formal analysis, Conceptualization. **Samir Genaim:** Writing – review & editing, Writing – original draft, Validation, Software, Resources, Methodology, Formal analysis, Conceptualization. **Pablo Gordillo:** Writing – review & editing, Writing – original draft, Validation, Software, Resources, Methodology, Formal analysis, Conceptualization. **Alejandro Hernández-Cerezo:** Writing – review & editing, Writing – original draft, Validation, Software, Resources, Methodology, Formal analysis, Conceptualization. **Daniel Kirchner:** Writing – review & editing, Writing – original draft, Validation, Resources, Methodology. **Albert Rubio:** Writing – review & editing, Writing – original draft, Validation, Software, Resources, Methodology, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research has been funded partially by the Spanish MCI, AEI and FEDER (EU) project PID2021-122830OB-C41, by the CM projects TEC-2024/COM-235 and PR17/24 BOVIR, and by the Ethereum Foundation under grants GASOL (no. FY21-0372) and FORVES (no. FY22-0698).

Data availability

Source code available at https://github.com/costa-group/gasol-optimizer.

References

[1] Live ethereum price, 2022, https://blockworks.co/price/eth.
[2] Uniswap protocol cumulative volume, 2023, https://dune.com/queries/2393272/3926150.
[3] Uniswap trading volume, 2023, https://cryptopotato.com/uniswap-surpasses-1-5t-trading-volume-data/.
[4] UniswapV3 contract, 2022, https://docs.uniswap.org/contracts/v3/overview.
[5] Uniswap V3 gas consumption, 2022, https://dune.com/k2rbpz/df12g3h4j3.
[6] H. Massalin, Superoptimizer - a look at the smallest program, in: Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), 1987, pp. 122–126, http://dx.doi.org/10.1145/36206.36194.
[7] S. Bansal, A. Aiken, Automatic generation of peephole superoptimizers, in: J.P. Shen, M. Martonosi (Eds.), Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006, ACM, 2006, pp. 394–403, http://dx.doi.org/10.1145/1168857.1168906.
[8] P.M. Phothilimthana, A. Thakur, R. Bodík, D. Dhurjati, Scaling up superoptimization, in: T. Conte, Y. Zhou (Eds.), Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016, ACM, 2016, pp. 297–310, http://dx.doi.org/10.1145/2872362.2872387.
[9] A. Jangda, G. Yorsh, Unbounded superoptimization, in: Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2017, Vancouver, BC, Canada, October 23 - 27, 2017, 2017, pp. 78–88, http://dx.doi.org/10.1145/3133850.3133856.
[10] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, J. Taneja, J. Regehr, Souper: A synthesizing superoptimizer, 2017, CoRR abs/1711.04422. arXiv:1711.04422. URL http://arxiv.org/abs/1711.04422.
[11] J. Nagele, M.A. Schett, Blockchain superoptimizer, in: Preproceedings of 29th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2019, 2019, https://arxiv.org/abs/2005.05912.
[12] E. Albert, P. Gordillo, A. Hernández-Cerezo, A. Rubio, A max-SMT superoptimizer for EVM handling memory and storage, in: D. Fisman, G. Rosu (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held As Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I, in: Lecture Notes in Computer Science, vol. 13243, Springer, 2022, pp. 201–219, http://dx.doi.org/10.1007/978-3-030-99524-9_11.
[13] M. Mohri, A. Rostamizadeh, A. Talwalkar, Foundations of machine learning, Adaptive computation and machine learning, MIT Press, ISBN: 978-0-262-01825-8, 2012, URL http://mitpress.mit.edu/books/foundations-machine-learning-0.
[14] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780, http://dx.doi.org/10.1162/neco.1997.9.8.1735.
[15] J. Nagele, M.A. Schett, Blockchain superoptimizer, in: Preproceedings of the 29th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2019, 2019.
[16] J. Cabrera-Arteaga, S. Donde, J. Gu, O. Floros, L. Satabin, B. Baudry, M. Monperrus, Superoptimization of WebAssembly bytecode, in: A. Aguiar, S. Chiba, E.G. Boix (Eds.), Programming'20: 4th International Conference on the Art, Science, and Engineering of Programming, Porto, Portugal, March 23-26, 2020, ACM, 2020, pp. 36–40, http://dx.doi.org/10.1145/3397537.3397567.
[17] G. Wood, Ethereum: A secure decentralised generalised transaction ledger, 2025, https://ethereum.github.io/yellowpaper/paper.pdf.
[18] M. Mukherjee, P. Kant, Z. Liu, J. Regehr, Dataflow-based pruning for speeding up superoptimization, Proc. ACM Program. Lang. 4 (OOPSLA) (2020) 177:1–177:24, http://dx.doi.org/10.1145/3428245.
[19] E. Albert, P. Gordillo, A. Rubio, M.A. Schett, Synthesis of super-optimized smart contracts using max-SMT, in: S.K. Lahiri, C. Wang (Eds.), Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I, in: Lecture Notes in Computer Science, vol. 12224, Springer, 2020, pp. 177–200, http://dx.doi.org/10.1007/978-3-030-53288-8_10.

[20] R. Sharma, E. Schkufza, B.R. Churchill, A. Aiken, Conditionally correct superoptimization, in: J. Aldrich, P. Eugster (Eds.), Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, Part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015, ACM, 2015, pp. 147–162, http://dx.doi.org/10.1145/2814270.2814278.

[21] E. Schkufza, R. Sharma, A. Aiken, Stochastic superoptimization, in: V. Sarkar, R. Bodík (Eds.), Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013, ACM, 2013, pp. 305–316, http://dx.doi.org/10.1145/2451116.2451150.

[22] L.M. de Moura, N. Bjørner, Z3: An efficient SMT solver, in: C.R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, in: Lecture Notes in Computer Science, vol. 4963, Springer, 2008, pp. 337–340, http://dx.doi.org/10.1007/978-3-540-78800-3_24.

[23] A. Cimatti, A. Griggio, B.J. Schaafsma, R. Sebastiani, The MathSAT5 SMT solver, in: Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013. Proceedings, 2013, pp. 93–107, http://dx.doi.org/10.1007/978-3-642-36742-7_7.

[24] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, A. Rubio, The barcelogic SMT solver, in: Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, USA, July 7-14, 2008, Proceedings, 2008, pp. 294–298, http://dx.doi.org/10.1007/978-3-540-70545-1_27.

[25] Seaport contract, 2022, https://etherscan.io/address/0x00000000006c3852cbef3e08e8df289169ede581#code.

[26] N. Grech, S. Lagouvardos, I. Tsatiris, Y. Smaragdakis, Elipmoc: advanced decompilation of ethereum smart contracts, Proc. ACM Program. Lang. 6 (OOPSLA1) (2022) 1–27, http://dx.doi.org/10.1145/3527321.

[27] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, Y. Smaragdakis, MadMax: surviving out-of-gas conditions in ethereum smart contracts, Proc. ACM Program. Lang. 2 (OOPSLA) (2018) 116:1–116:27, http://dx.doi.org/10.1145/3276486.

[28] N. Grech, L. Brent, B. Scholz, Y. Smaragdakis, Gigahorse: thorough, declarative decompilation of smart contracts, in: J.M. Atlee, T. Bultan, J. Whittle (Eds.), Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, IEEE / ACM, 2019, pp. 1176–1186, http://dx.doi.org/10.1109/ICSE.2019.00120.

[29] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, B. Scholz, Vandal: A scalable security analysis framework for smart contracts, 2018, CoRR abs/1809.03981. arXiv:1809.03981. URL http://arxiv.org/abs/1809.03981.

[30] E. Albert, M.G. de la Banda, A. Hernández-Cerezo, A. Ignatiev, A. Rubio, P.J. Stuckey, Superstack: Superoptimization of stack-bytecode via greedy, constraint-based, and SAT techniques, Proc. ACM Program. Lang. 8 (PLDI) (2024) 1437–1462, http://dx.doi.org/10.1145/3656435.

[31] E. Albert, P. Gordillo, A. Hernández-Cerezo, A. Rubio, M.A. Schett, Superoptimization of smart contracts, ACM Trans. Softw. Eng. Methodol. (ISSN: 1049-331X) 31 Issue 4 (70) (2022) 1–29, http://dx.doi.org/10.1145/3506800.

[32] Etherscan, 2018, https://etherscan.io.

[33] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, Learning phrase representations using RNN encoder-decoder for statistical machine translation, in: A. Moschitti, B. Pang, W. Daelemans (Eds.), Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, a Meeting of SIGDAT, a Special Interest Group of the ACL, ACL, 2014, pp. 1724–1734, http://dx.doi.org/10.3115/v1/d14-1179.

[34] T.P. Team, PyTorch, http://pytocrh.org.

[35] BigQuery, 2023, https://cloud.google.com/bigquery.

[36] ERC-20 token standard, 2023, https://ethereum.org/en/developers/docs/standards/tokens/erc-20/.

[37] ERC-721 non-fungible token standard, 2023, https://ethereum.org/en/developers/docs/standards/tokens/erc-721/.

[38] OpenZeppelin contracts repository, 2023, https://github.com/OpenZeppelin/openzeppelin-contracts.

[39] P. Kotsias, pcko1/etherscan-python, 2020, http://dx.doi.org/10.5281/zenodo.4306855, URL https://github.com/pcko1/etherscan-python.

[40] EthereumPrice, 2023, https://ethereumprice.org/history/?start=2019-02-28&end=2023-05-10&currency=USD.

[41] J. Cabrera Arteaga, S. Donde, J. Gu, O. Floros, L. Satabin, B. Baudry, M. Monperrus, Superoptimization of WebAssembly bytecode, in: Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming, Programming '20, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450375078, 2020, pp. 36–40, http://dx.doi.org/10.1145/3397537.3397567.

[42] R. Joshi, G. Nelson, Y. Zhou, Denali: A practical algorithm for generating optimal code, ACM Trans. Program. Lang. Syst. 28 (6) (2006) 967–989, http://dx.doi.org/10.1145/1186633.

[43] M. Xu, Z. Li, O. Padon, S. Lin, J. Pointing, A. Hirth, H. Ma, J. Palsberg, A. Aiken, U.A. Acar, Z. Jia, Quartz: superoptimization of quantum circuits, in: R. Jhala, I. Dillig (Eds.), PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022, ACM, 2022, pp. 625–640, http://dx.doi.org/10.1145/3519939.3523433.

[44] S. Singh, M. Zhang, S. Khurshid, Learning guided enumerative synthesis for superoptimization, in: F. Biondi, T. Given-Wilson, A. Legay (Eds.), Model Checking Software - 26th International Symposium, SPIN 2019, Beijing, China, July 15-16, 2019, Proceedings, in: Lecture Notes in Computer Science, vol. 11636, Springer, 2019, pp. 172–192, http://dx.doi.org/10.1007/978-3-030-30923-7_10.

[45] A. Kalyan, A. Mohta, O. Polozov, D. Batra, P. Jain, S. Gulwani, Neural-guided deductive search for real-time program synthesis from examples, in: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings, OpenReview.net, 2018, URL https://openreview.net/forum?id=rywDjg-RW.

[46] B. Mariano, Y. Chen, Y. Feng, G. Durrett, I. Dillig, Automated transpilation of imperative to functional code using neural-guided program synthesis, Proc. ACM Program. Lang. 6 (OOPSLA) (2022) 1–27, http://dx.doi.org/10.1145/3527315.