The Journal of Systems and Software 221 (2025) 112284

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Harnessing heap analysis for the synthesis of superoptimized bytecode ☆

Elvira Albert [a], Jesús Correas [a], Pablo Gordillo [a], Guillermo Román-Díez [b,*], Albert Rubio [a]

[a] Complutense University of Madrid, Spain
[b] Universidad Politécnica de Madrid, Spain

ARTICLE INFO

Keywords:
Superoptimization
Synthesis
Heap-analysis
Smart-contracts
Ethereum

ABSTRACT

Superoptimization is a type of program synthesis technique that, given an original loop-free sequence of instructions, synthesizes an alternative semantically-equivalent sequence that is optimal wrt the considered objective function. Working on loop-free sequences restricts the kind of achievable optimizations to the *local* scope of the considered sequences. This article harnesses a *global* heap analysis for the synthesis of superoptimized loop-free sequences of bytecode. The global heap analysis will allow us to infer useless write heap accesses, aliasing and non-aliasing properties, and calling-contexts for the sequences. Exploiting this information on an existing superoptimizer for Ethereum bytecode has required novel extensions: (1) developing a finer-grained heap analysis able to infer heap properties that can boost superoptimization, (2) adapting several components of the superoptimizer to leverage the heap properties, and (3) extending the superoptimization algorithm to work in a context-sensitive way. Our experimental results on more than 200,000 sequences show that harnessing heap analysis for superoptimization not only improves the quality of the optimization but it can even reduce the optimization time.

1. Introduction

Superoptimization (Massalin, 1987) is a class of program synthesis techniques that, given an original loop-free sequence of instructions (acting as specification), searches for an equivalent sequence that is *optimal* wrt a given objective function. Optimality is achieved by searching in the full space of alternative sequences using an automated constraint-solver, e.g., an SMT solver. Superoptimization was conceived to be applied in a second layer of optimization within a compiler, i.e., traditional optimizations such as inlining, dead-code elimination, etc. are typically applied by the compiler before. Within a second pass, superoptimization can achieve additional types of optimizations that are unachievable by means of pattern-based transformations. For instance, in a bytecode sequence, reordering two consecutive heap accesses to different locations can lead to more efficient code if the address of the later access is found at the top position of the stack; namely reordering would avoid swapping instructions to place the needed arguments at the top. Due to these additional optimization achievements, superoptimization is gaining much attention in contexts in which efficiency is crucial (e.g., in the blockchain context as clients pay a fee according to the executed opcodes). Superoptimization tools have been developed for LLVM (Jangda and Yorsh, 2017; Sasnauskas et al., 2017; Mukherjee et al., 2020), Ethereum VM (EVM) (Nagele and Schett, 2019; Albert et al., 2022b), and WebAssembly (Cabrera-Arteaga et al., 2020). While our approach is defined and implemented in the context of EVM, the ideas are applicable to other languages simply by using a heap analysis for the corresponding heap model.

The main limitation of superoptimization is the intrinsic *locality* of the method because (by definition) it is applied on a per-sequence basis by ignoring *global* information that could be only learned by a whole-program analysis. This limitation is particularly noticeable for heap-related instructions as by reasoning on a local sequence is very unlikely to find out, e.g., aliasing and/or non-aliasing properties. Therefore, optimizations – such as the reordering of accesses mentioned above – are missed by superoptimization. While recent work (Albert et al., 2022a) on the GASOL system (a state-of-the-art superoptimization tool for EVM smart contracts) has pointed out the relevance of optimizing heap accesses, GASOL's reasoning so far has been limited to local sequences. This article harnesses a global heap analysis for the local synthesis of superoptimized bytecode in GASOL as follows:

(i) we leverage information on useless write heap accesses, inferred using a heap analysis, to the superoptimizer in such a way that it not only eliminates the useless MSTORE opcode, but also other opcodes that are only required by the MSTORE;

☆ This work is partially funded by the Ethereum Foundation under grant 435-2023.
* Corresponding author.
E-mail addresses: elvira@sip.ucm.es (E. Albert), jcorreas@ucm.es (J. Correas), pabgordi@ucm.es (P. Gordillo), guillermo.roman@upm.es (G. Román-Díez), alberu04@ucm.es (A. Rubio).

https://doi.org/10.1016/j.jss.2024.112284
Received 6 March 2024; Received in revised form 30 September 2024; Accepted 12 November 2024
Available online 26 November 2024

(ii) we annotate heap-related opcodes with aliasing and non-aliasing properties which enable a series of optimizations within the superoptimizer such as instruction reordering, detection of redundant MLOAD's, etc.;
(iii) in order to infer the (global) heap properties, the heap analysis propagates information on the *calling contexts* to each of the sequences which can be exploited by the superoptimizer.

Contributions. To intertwine the heap analysis and superoptimization as described in the three points above, this article makes the following technical contributions:

(1) *Heap analysis with offsets*. We increase the accuracy of the *heap analysis* for EVM bytecode of Albert et al. (2023), which only inferred the heap *slots*[1] created by the code, in order to infer also the offsets for each of the accesses within the slots. Our finer-grained extension of the heap analysis is required to infer the aliasing properties in point (ii) above, it is able to detect more useless accesses in point (i) and a more accurate context in (iii) than Albert et al. (2023) would do.
(2) *Superoptimization with heap properties*. We identify different types of global heap-related properties that a heap analysis can provide to a superoptimization tool (points 1–2 above) and enhance an existing superoptimizer that was operating at a purely local level on the sequences to exploit the inferred global heap properties in the most advantageous way.
(3) *Context-sensitive superoptimization.* We leverage a context-insensitive superoptimizer by providing a calling context that provides global information on the input to the sequences.
(4) *Implementation and evaluation.* We integrate our approach within the GASOL tool and perform a thorough experimental evaluation on sequences taken from 14,034 real smart contracts. Our results show that, when heap properties can be exploited, optimization gains wrt the previous version of GASOL (Albert et al., 2022a) improve from 10.7% length-reduction to 16.3% while the median of the optimization time per sequence is reduced by 0.4 s

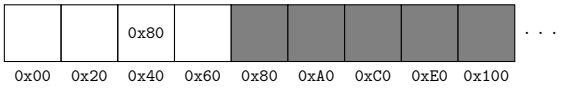
Our previous work in this area has focused either: (1) on defining the basic technique of superoptimization in the context of the EVM (Albert et al., 2022b,a), and scalability improvements (Albert et al., 2024), or (2) on developing a heap analysis (Albert et al., 2023) for the EVM bytecode generated by the Solidity compiler, solc (Solc, 2024). Such heap analysis has been applied in the context of verification and code optimization. As regards the latter application to optimization, its use consisted in finding write memory accesses that can be removed because they are not read afterwards. This optimization is captured by our combination of superoptimization and heap analysis, but beyond it, we can achieve new kinds of optimizations that are novel and original to this work as we will see throughout the paper.

2. Heap analysis of EVM bytecode with offsets

The EVM has several memory regions to store data: a stack of 256-bit words with a maximum size of 1024 words, a byte- and word-addressable volatile local memory, and a persistent memory called *storage*. The EVM bytecode is a stack-based virtual machine in which most opcodes operate with the topmost elements of the stack to perform computations. The stack is handled using standard stack-manipulating opcodes (e.g., DUP, SWAP, POP, ...), and the EVM bytecode language includes the usual arithmetic and bit operations (e.g., ADD retrieves the two topmost elements of the stack and leaves its addition at the top) and control-flow bytecodes (e.g., JUMP to continue execution at the opcode located at the top of the stack). The EVM opcode set contains two basic opcodes to access memory: MLOAD and MSTORE, which load and store a 32-byte word from memory, respectively.[2] The Solidity compiler solc uses the EVM local memory to allocate reference-type data (*e.g.*, arrays or structs). Also, some specific EVM bytecode opcodes require lengthy operands to be stored in local memory, *e.g.* the opcodes for computing cryptographic hashes, or for communicating data to and from external function calls. The bytecode generated by the solc compiler handles local memory using a cumulative model that never releases allocated memory. As there is no specific opcode in the EVM for allocating memory, bytecode generated by solc uses the value stored at a particular address (0x40) as a *free memory pointer*, i.e., a pointer to the first memory address available.

Memory slots. The bytecode generated by the solc compiler accesses the memory by *slots*, which are sequences of consecutive memory locations that are always accessed by using the address of the initial location in the slot, that we call its *base reference*, plus an offset used to access a specific location within the slot. When a Solidity program has to allocate t memory locations, the bytecode generated by solc reads the free memory pointer at address 0x40. The value read from the free memory pointer is the *base reference* to the slot, and any subsequent access to the slot is performed using this base reference plus an optional offset. Finally, the reference value stored in 0x40 is incremented by t positions and will be used in the next memory allocation.

Example 1 (*Memory Slot Reservation*). The Solidity code shown in Fig. 1 represents a usual programming pattern for struct creation (the private function fn), found in many real contracts.[3] The solc compiler, with the most advanced optimizations enabled, produces the bytecode showed below the Solidity code. The bytecode generated by solc initializes the free memory pointer as follows:



Locations 0x00, 0x20 and 0x60 are reserved by the compiler, and the free memory pointer at location 0x40 points to the first available location. The left column of the EVM code in Fig. 1 corresponds to the creation and (default) initialization of the struct o that is declared as the result of function fn: a fresh heap slot is created, and both fields are assigned an initial value of zero. The instructions for the creation of the struct are at *program points* (*pp* for short) 1–7, first reading the free memory pointer (*pp* 2), then adding the size of the struct (*pp* 3–4), and finally pushing the new free memory pointer (*pp* 7). The dots (*pp* 5) omit opcodes that perform a maximum length check, which are not relevant to this work. The struct contents are initialized to zero at *pp* 13 and 15, and the assignments of its fields in o are performed at *pps* 21 and 25. The execution of the EVM bytecode shown in the left column leaves the memory as follows:

The Solidity code also includes an external function extfn that returns the struct just created by fn. Thus, the execution of extfn creates two slots: the one created in fn (named m_1 and shown in light blue) and another slot needed to return the result of fn and assign it to variable o2 of extfn (named m_2 and shown in light red in next figure). The

[1] A slot in EVM is a region of the heap accessed from the same base reference.

[2] The EVM also includes the opcode MSTORE8 to access the local memory at a byte level, although we only consider the more general word-addressable MSTORE to keep the description simpler.

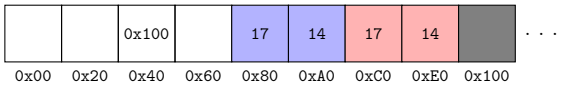
[3] Indeed, our example is a simplification of code in a real smart contract (ERC721A, 2024).

```
struct StrType {
  uint a;
  uint b;
}

function fn() private pure returns(StrType memory o) {
  o.a = 17;
  o.b = 14;
}

function extfn() external pure returns (StrType memory o2) {
  o2 = fn();
}
```

```
1  PUSH 0x40   // freeptr address
2  LOAD        // freeptr read
3  PUSH 0x40   // size of slot m1
4  ADD         // freeptr computation
5  ...
6  PUSH 0x40   // freeptr address
7  MSTORE      // freeptr update
8  PUSH 0x0    // zero initial value
9  PUSH 0x20   // offset of m1.b
10 DUP4        // address of slot m1
11 DUP3        // zero initial value
12 DUP2        // address of m1.a
13 MSTORE      // m1.a initialized to 0
14 ADD         // address of m1.b
15 MSTORE      // m1.b initialized to 0
16 JUMP
17 ...
18 JUMPDEST
19 PUSH 0x11   // 17
20 DUP2        // address of m1.a
21 MSTORE      // m1.a = 17
22 ADD         // address of m1.b
23 PUSH 0x0e   // 14
24 DUP2        // address of m1.b
25 MSTORE      // m1.b = 14
26 DUP2        // 0x40(freeptr address)
27 MLOAD       // freeptr read (addr.m2)
28 SWAP1       // address of m1.b
29 PUSH 0x11   // 17
30 DUP3        // address of m2.a
31 MSTORE      // m2.a = 17
32 MLOAD       // load of m1.b
33 PUSH 0x20   // offset of m2.b
34 DUP3        // address of m2
35 ADD         // address of m2.b
36 MSTORE      // m2.b = m1.b
37 RETURN      // return of extfn()
```

Fig. 1. Solidity code and an excerpt of its corresponding Bytecode.

right column of the EVM bytecode in Fig. 1 shows the bytecode that allocates the slot of extfn for storing the struct o2 (*pp* 27), and copies the contents of o to o2 (*pp* 31–36). The final layout of the memory after executing extfn is as follows:



2.1. Basic heap analysis for EVM (Albert et al., 2023)

The final goal of the heap analysis is to infer, for all heap access instructions (MLOAD and MSTORE) in the bytecode program, an (over-)approximation of the heap location(s) that might be accessed by the corresponding instructions. The definition of the analysis includes an analysis domain which represents all possible accessed heap locations "abstractly" (as usual, an abstract value might represent multiple concrete values and $\top$ means any location). Note that the same instruction might access more than one location depending on the actual execution trace, and the analysis must compute a sound over-approximation that includes all of them. There exists a wide variety of heap analyses formalized using different underlying techniques, and with different accuracy levels, for multiple programming languages (see, e.g., Itzhaky et al. (2014), Smaragdakis et al. (2011), Berdine et al. (2007) and Milanova et al. (2005) and their references). For the case of the EVM bytecode, we rely on the heap analysis in Albert et al. (2023), briefly overviewed in this section and, afterwards, we propose an accuracy extension in Section 2.2.

The analysis of Albert et al. (2023) aims at inferring the set of slots to which each heap access opcode refers. By analyzing the bytecode and searching for the patterns used to create slots, it first defines the set of *abstract slots* (named $\mathcal{L}$) which are accessed in the bytecode. This set $\mathcal{L}$ is defined as $\mathcal{L} = \{l_1, \ldots, l_n\}$ where each l_i is a symbolic unique identifier used to refer to a distinct abstract slot. The slots are *abstract* as several runtime slots might be represented by a single abstract slot, e.g., when multiple slots are created within a loop. When this happens in the analysis, we annotate it by adding a * to the slot name (e.g. $l_i{}^*$) as it is needed by the aliasing definition below. We denote by $\mathcal{L}^*$ the subset of $\mathcal{L}$ containing the annotated slots. As notation, we use $\mathcal{P}$ to refer to the set of all instructions in the program, which have the form $pp{:}I \in \mathcal{P}$ where I is the instruction at program point pp. The following definition summarizes the output of the heap analysis of Albert et al. (2023) which is our starting point.

Definition 1 (basicHeapAnalysis)**.** Function basicHeapAnalysis returns a mapping $\pi : \mathcal{P} \times \mathcal{S} \mapsto \wp(\mathcal{L})$ that, for each program point $pp{:}I \in \mathcal{P}$ and stack position $s \in \mathcal{S}$, returns a set of abstract slots L that either is empty, or s is a reference to one of the slots in L at pp in any execution trace.

Example 2. The basic heap analysis of Albert et al. (2023) identifies two abstract slots: l_1 and l_2 that correspond to the concrete slots m_1 and m_2 in Example 1, and yields the following function π that we show at some relevant program points: $\pi(13, s_0){=}\{l_1\}$, $\pi(15, s_0){=}\{l_1\}$, $\pi(21, s_0){=}\{l_1\}$, $\pi(25, s_0){=}\{l_1\}$, $\pi(31, s_0){=}\{l_2\}$, $\pi(32, s_0){=}\{l_1\}$, $\pi(36, s_0){=}\{l_2\}$. We use s_0 to refer to the top and s_n to the $top - n$ of the stack. We can observe that, though *pps* 21 and 25 access different heap locations (fields a and b respectively), the basic heap analysis cannot distinguish them: using π we only know that slot l_1 is accessed at *pps* 21 and 25.

2.2. Extension with offsets

Our contribution is the extension of the heap analysis described above in order to consider not only the abstract slots referenced by stack locations, but also the particular word being accessed (i.e., the *offset*). This extension is fundamental to infer the aliasing properties in Section 3.3 and to have a more precise detection of useless write accesses in Section 3.2 and calling contexts in Section 3.4. Developing this extension amounts to keeping track during the analysis of the offsets that are added to the slot base references as constants. Our extension with offsets has two fundamental differences with the basic heap analysis above: (1) It computes an additional analysis function, named σ, that takes care of the offset component only. This analysis is quite standard as the offsets are in the set $\mathbb{N}^0 \cup \{\top\}$ which constitute the domain of the analysis and whose values are unsigned constants (see, e.g., *pp* 9) incremented using addition (see, e.g., *pp* 14) and the special symbol $\top$. Constant values and increments need to be propagated within the stack elements at each program point during the analysis. In order to guarantee termination of the fixed-point analysis in the presence of loops, after a finite number of iterations, we return $\top$ as offset for those accesses that are not converging to a finite set of values. (2) The second difference is that function ϕ which takes care of the heap locations now returns pairs $\langle l, o\rangle$ made up of the abstract slot $l \in \mathcal{L}$ and the offset $o \in \mathbb{N}^0 \cup \{\top\}$. Note that a pair with $\top$ in the offset component means that we only have information about the abstract slot being referenced. We denote by $\mathcal{L}^+$ the set of all such pairs. While function σ is auxiliary to compute ϕ, the heap analysis with offsets now returns, besides ϕ, also function σ because it will be needed to compute the calling contexts in Section 3.4.

Definition 2 (`heapAnalysis`)**.** The heap analysis with offsets, invoked by means of function `heapAnalysis`, returns the following functions:

(1) $\sigma : \mathcal{P} \times \mathcal{S} \mapsto \wp(\mathbb{N}^0 \cup \{\top\})$ is a mapping that, for each program point $pp{:}I \in \mathcal{P}$ and stack position $s \in \mathcal{S}$, returns a finite set C such that if $C \neq \emptyset$ and $C \subset \mathbb{N}^0$ then $v \in C$, where v either is $\top$ or v is one possible value stored in the stack at position s at pp in any execution trace.

(2) $\phi : \mathcal{P} \times \mathcal{S} \mapsto \wp(\mathcal{L}^+)$ is a mapping that, for each program point $pp{:}I \in \mathcal{P}$ and stack position $s \in \mathcal{S}$, returns a set L of pairs $\langle m, o\rangle$ that is either empty or s is a reference to one of the pairs in L at pp in any execution trace.

We assume that `heapAnalysis`(blocks) returns the results of (ϕ, σ) only for the program points in the set of sequences blocks.

Example 3. As an example, function σ at *pp* 22 returns $\sigma(22, s_1)$={32}, which is used to compute the location $\langle l_1, 32\rangle$, accessed by the `MSTORE` at *pp* 25. Our heap analysis extended with offsets produces the following result for ϕ at some selected *pps*: $\phi(13, s_0)$=$\{\langle l_1, 0\rangle\}$, $\phi(15, s_0)$=$\{\langle l_1, 32\rangle\}$, $\phi(21, s_0)$=$\{\langle l_1, 0\rangle\}$, $\phi(25, s_0)$=$\{\langle l_1, 32\rangle\}$, $\phi(31, s_0)$=$\{\langle l_2, 0\rangle\}$, $\phi(32, s_0)$=$\{\langle l_1, 32\rangle\}$, $\phi(36, s_0)$=$\{\langle l_2, 32\rangle\}$, where we can observe that the heap locations accessed at *pps* 13 and 15 are identified now as distinct (and similarly at other program points).

Ensuring aliasing at two heap access instructions can only be done using our analysis when the inferred result refers to a single concrete slot. This can be guaranteed if there has been no abstraction due to loops (i.e., the slot has not been annotated with * and hence does not belong to $\mathcal{L}^*$, see Section 2.1) and there is only one possible slot accessed (i.e., the size of the set $\phi(pp_1, s_0)$ is equals to one). The following definition of "safe alias" ensures these conditions.

Definition 3 (*safeAlias*)**.** Given two program points pp_i and pp_j and stack positions s_k and s_m, `safeAlias`(pp_i, s_k, pp_j, s_m) returns true iff $\phi(pp_i, s_k)$ is of the form $\{\langle x, o\rangle\}$ and the following holds: $x \notin \mathcal{L}^* \wedge o \neq \top \wedge \phi(pp_i, s_k) = \phi(pp_j, s_m)$.

Algorithm 1: Superoptimization Algorithm (black) with Heap (blue)

```
1:  procedure SuperoptimizationWithHeapProperties(p,f)
2:  Input: bytecode program p, objective function f
3:  Output: optimized bytecode o
4:  Ensures: f(o) = min{f(o') | o' ≡ p}
5:  blocks ← buildCFG(p)
6:  (φ,σ) ← heapAnalysis(blocks)
7:  useless ← computeUseless(blocks,φ)
8:  result ←[ ]
9:  for (b, istack) ∈ blocks do
10:   (ostack,mem) ← symbolicExecution(b,istack)
11:   (alias,nonalias) ← computeAliasing(b,φ)
12:   context ← computeContext(b,φ,σ)
13:   mem ← applyUseless(mem,useless)
14:   mem ← applyNonAliasing(mem,nonalias)
15:   (istack,ostack,mem) ←
      applyContext(istack,ostack,mem,context)
16:   (ostack,mem) ←
      ruleBasedSimplification(ostack,mem,alias)
17:   opt ← synthesizeOptimal(f,istack,ostack,mem)
18:   result ← result.append(opt)
19: return o ← rebuild(result)
```

Intuitively, two instructions are guaranteed not to access the same location if the sets inferred by the heap analysis are disjoint. It must be checked explicitly that the offsets, within the possible slots accessed at the instructions, have not been lost by the analysis in just one of the sets (become $\top$), as definition below states.

Definition 4 (*nonAlias*)**.** Given two program points pp_i and pp_j and stack positions s_k and s_m, and function ϕ. The function `nonAlias`(pp_i, s_k, pp_j, s_m) returns true iff $\forall\langle l_1, o_1\rangle \in \phi(pp_i, s_k), \langle l_2, o_2\rangle \in \phi(pp_j, s_m)$: $l_1 \neq l_2 \vee (o_1 \neq \top \wedge o_2 \neq \top \wedge o_1 \neq o_2)$.

3. Harnessing heap analysis for superoptimization

This section presents the properties and relations that we compute from the heap analysis output and how they can be exploited by a superoptimization tool. The rest of the section is structured as follows: Section 3.1 first overviews a basic superoptimization algorithm; Section 3.2 presents our approximation of useless write accesses and its integration within the superoptimization algorithm; Section 3.3 defines the aliasing and non-aliasing properties that we compute from the heap analysis and then incorporates them within the superoptimizer; finally, Section 3.4 provides the notion of calling context and our context-sensitive extension.

3.1. Synthesis of superoptimized bytecode

This section describes the basic components of the state-of-the-art algorithm for superoptimization, which correspond to lines 2–5, 8–10 and 16–19 displayed in black text in Algorithm 1. The remaining lines displayed in blue color are the novelties that will be explained in the coming sections. The algorithm receives as input (line 2) a bytecode p to be superoptimized and an objective function f among those available in the system. The only assumption for the objective function is that the cost for each instruction in p is statically known. The original objective function used in superoptimization is the program's *length* (Bansal and Aiken, 2006; Jangda and Yorsh, 2017; Sasnauskas et al., 2017) measured by its number of operations. However, other objectives have been introduced by new programming environments. In particular, `ebso` (Nagele and Schett, 2019), `SYRUP` (Albert et al.,

$$
\begin{array}{lll}
(\mathsf{stack}, \mathsf{mem}) & \rightarrow_{pp:\mathtt{POP}} & (\mathsf{stack}[1:n], \mathsf{mem}) \\
(\mathsf{stack}, \mathsf{mem}) & \rightarrow_{pp:\mathtt{DUPx}} & ([\mathsf{stack}[\mathsf{x}-1] \mid \mathsf{stack}], \mathsf{mem}) \\
(\mathsf{stack}, \mathsf{mem}) & \rightarrow_{pp:\mathtt{SWAPx}} & (\mathsf{stack}', \mathsf{mem}), \mathsf{stack}'[0] = \mathsf{stack}[\mathsf{x}], \mathsf{stack}'[\mathsf{x}] = \mathsf{stack}[0], \\
\ldots & & \quad \mathsf{stack}'[i] = \mathsf{stack}[i]\ \forall i > 0, i \neq \mathsf{x} \\
(\mathsf{stack}, \mathsf{mem}) & \rightarrow_{pp:\mathtt{UF}} & ([\mathtt{UF}(\mathsf{stack}[0], ..., \mathsf{stack}[\delta-1]) \mid \mathsf{stack}[\delta:n]\ ], \mathsf{mem}), \delta = ar(\mathtt{UF}) \\
(\mathsf{stack}, \mathsf{mem}) & \rightarrow_{pp:\mathtt{MLOAD}} & ([\mathtt{MLOAD}_{pp}(\mathsf{stack}[0]) \mid \mathsf{stack}[1:n]], add(\mathtt{MLOAD}_{pp}(\mathsf{stack}[0]), \mathsf{mem})) \\
(\mathsf{stack}, \mathsf{mem}) & \rightarrow_{pp:\mathtt{MSTORE}} & (\mathsf{stack}[2:n], add(\mathtt{MSTORE}_{pp}(\mathsf{stack}[0], \mathsf{stack}[1]), \mathsf{mem}))
\end{array}
$$

Fig. 2. Symbolic execution of selected opcodes (pp current program point).

2020) and GASOL (Albert et al., 2022a) minimize the program's *gas* (a precise definition for the gas objective function appears in Gavin Wood (2019)). Gas is used to measure the price to pay for executing the instructions, thus assigning larger gas cost to opcodes that require more computation or storage (e.g., stack-manipulating opcodes like DUP, SWAP and heap accesses cost 3 gas units, but accessing the global storage can cost up to 22 100 units). Our examples in the article report the gains for both objective functions: gas and length, and the experimental evaluation uses the length. The algorithm returns as output (line 3) an optimized bytecode o that is ensured to be optimal (line 4), i.e., it has minimal cost wrt f (for the considered partitioning into sequences) and is semantically equivalent (denoted ≡) to p.

The first step for all superoptimizers is to build a control flow graph (CFG) from the bytecode program that allows detection of its loops and branching. A common approach is to generate then one sequence per block of the CFG; hence, the sequence is not only loop-free but also *jump-free*. This is done, for example, by the superoptimizers ebso (Nagele and Schett, 2019), SYRUP (Albert et al., 2020), and Souper (Sasnauskas et al., 2017), and assumed (without loss of generality) in what follows. In this first step, one can also statically compute the number k of elements the operand stack must have right before reaching each sequence. This k is used to create an initial stack, denoted istack, with k distinct variables istack = $[s_0, \ldots, s_{k-1}]$. We assume that function buildCFG in line 5 returns a set blocks containing all the blocks in the CFG of the program p, where each element of blocks is a pair (b, istack) with b being the sequence of opcodes within the block and istack its initial stack.

Example 4. The left column in Fig. 1 results in two blocks partitioned by the JUMP* opcodes. The right column is a single block and the size of its input stack is known to be three, i.e., istack = $[s_0, s_1, s_2]$ where s_0 is the top of the stack.

Each of the sequences is superoptimized within a loop iteration of line 9. First, function symbolicExecution in line 10 symbolically executes the opcodes in b from the initial stack istack. Fig. 2 shows the symbolic execution of some representative opcodes. Here, a stack of n elements is denoted as stack[0 : $n-1$] and stack[k] denotes the $k+1$th stack element, with stack[0] being the topmost. We use UF (Uninterpreted Function) whenever the actual opcode cannot be executed and ar(UF) denotes its arity. During symbolic execution, heap access opcodes are annotated in the symbolic state with their program points as subindex to uniquely identify them (e.g., $\mathtt{MLOAD}_{pp}$). We may omit the subscript pp when it is not relevant. Symbolic execution of a block b starts from the state (istack,[ ]), where [ ] denotes the lack of information on previous heap accesses, to obtain the final state (ostack,mem), where ostack is the resulting stack (represented as a list of elements) and mem the list of heap accesses within b.

Example 5. Let us show the symbolic execution of the right sequence in Fig. 1 from (istack = $[s_0, s_1, s_2]$,[ ]). Underlined terms are used for later reference in Example 12.

$([s_0, s_1, s_2], [\,]) \rightarrow_{\mathtt{PUSH\ 0x11}} ([\mathtt{0x11}, s_0, s_1, s_2], [\,])$
$\rightarrow_{\mathtt{DUP2}} ([s_0, \mathtt{0x11}, s_0, s_1, s_2], [\,])$
$\rightarrow_{\mathtt{MSTORE}} ([s_0, s_1, s_2], [\mathtt{MSTORE}_{21}(s_0, \mathtt{0x11})]$
$\rightarrow_{\mathtt{ADD}} [\mathtt{ADD}(s_0, s_1), s_2], [\mathtt{MSTORE}_{21}(s_0, \mathtt{0x11})]$
$\rightarrow_{\mathtt{PUSH\ 0x0E}} ([\mathtt{0x0E}, \mathtt{ADD}(s_0, s_1), s_2], [\mathtt{MSTORE}_{21}(s_0, \mathtt{0x11})]$
$\rightarrow_{\mathtt{DUP2}} [\mathtt{ADD}(s_0, s_1), \mathtt{0x0E}, \mathtt{ADD}(s_0, s_1), s_2], [\mathtt{MSTORE}_{21}(s_0, \mathtt{0x11})]$
$\rightarrow_{\mathtt{MSTORE}} ([\mathtt{ADD}(s_0, s_1), s_2], [\mathtt{MSTORE}_{21}(s_0, \mathtt{0x11}), \mathtt{MSTORE}_{25}$
$(\mathtt{ADD}(s_0, s_1), \mathtt{0x0E})])$
$\rightarrow_{\mathtt{DUP2}} ([s_2, \mathtt{ADD}(s_0, s_1), s_2], [\mathtt{MSTORE}_{21}(s_0, \mathtt{0x11}), \mathtt{MSTORE}_{25}$
$(\mathtt{ADD}(s_0, s_1), \mathtt{0x0E})])$
$\rightarrow_{\mathtt{MLOAD}} ([\mathtt{MLOAD}_{27}(s_2), \mathtt{ADD}(s_0, s_1), s_2],$
$[\mathtt{MSTORE}_{21}(s_0, \mathtt{0x11}), \mathtt{MSTORE}_{25}(\mathtt{ADD}(s_0, s_1), \mathtt{0x0E}), \mathtt{MLOAD}_{27}(s_2)])$
$\rightarrow_{\mathtt{SWAP1}} ([\mathtt{ADD}(s_0, s_1), \mathtt{MLOAD}_{27}(s_2), s_2],$
$[\mathtt{MSTORE}_{21}(s_0, \mathtt{0x11}), \mathtt{MSTORE}_{25}(\mathtt{ADD}(s_0, s_1), \mathtt{0x0E}), \mathtt{MLOAD}_{27}(s_2)])$
$\rightarrow_{\mathtt{PUSH\ 0x11}} ([\mathtt{0x11}, \mathtt{ADD}(s_0, s_1), \mathtt{MLOAD}_{27}(s_2), s_2],$
$[\mathtt{MSTORE}_{21}(s_0, \mathtt{0x11}), \mathtt{MSTORE}_{25}(\mathtt{ADD}(s_0, s_1), \mathtt{0x0E}), \mathtt{MLOAD}_{27}(s_2)])$
$\rightarrow_{\mathtt{DUP3}} ([\mathtt{MLOAD}_{27}(s_2), \mathtt{0x11}, \mathtt{ADD}(s_0, s_1), \mathtt{MLOAD}_{27}(s_2), s_2],$
$[\mathtt{MSTORE}_{21}(s_0, \mathtt{0x11}), \mathtt{MSTORE}_{25}(\mathtt{ADD}(s_0, s_1), \mathtt{0x0E}), \mathtt{MLOAD}_{27}(s_2)])$
$\rightarrow_{\mathtt{MSTORE}} ([\mathtt{ADD}(s_0, s_1), \mathtt{MLOAD}_{27}(s_2), s_2],$
$[\mathtt{MSTORE}_{21}(s_0, \mathtt{0x11}), \mathtt{MSTORE}_{25}(\mathtt{ADD}(s_0, s_1), \mathtt{0x0E}), \mathtt{MLOAD}_{27}(s_2),$
$\mathtt{MSTORE}_{31}(\mathtt{MLOAD}_{27}(s_2), \mathtt{0x11})])$
$\rightarrow_{\mathtt{MLOAD}} ([\mathtt{MLOAD}_{32}(\mathtt{ADD}(s_0, s_1)), \mathtt{MLOAD}_{27}(s_2), s_2],$
$[\mathtt{MSTORE}_{21}(s_0, \mathtt{0x11}), \mathtt{MSTORE}_{25}(\mathtt{ADD}(s_0, s_1), \mathtt{0x0E}), \mathtt{MLOAD}_{27}(s_2),$
$\mathtt{MSTORE}_{31}(\mathtt{MLOAD}_{27}(s_2), \mathtt{0x11}), \mathtt{MLOAD}_{32}(\mathtt{ADD}(s_0, s_1))])$

$\rightarrow_{\mathtt{PUSH\ 0x20}} ([\mathtt{0x20}, \mathtt{MLOAD}_{32}(\mathtt{ADD}(s_0, s_1)), \mathtt{MLOAD}_{27}(s_2), s_2],$
$[\mathtt{MSTORE}_{21}(s_0, \mathtt{0x11}), \mathtt{MSTORE}_{25}(\mathtt{ADD}(s_0, s_1), \mathtt{0x0E}), \mathtt{MLOAD}_{27}(s_2),$
$\mathtt{MSTORE}_{31}(\mathtt{MLOAD}_{27}(s_2), \mathtt{0x11}), \mathtt{MLOAD}_{32}(\mathtt{ADD}(s_0, s_1))])$
$\rightarrow_{\mathtt{DUP3}} ([\mathtt{MLOAD}_{27}(s_2), \mathtt{0x20}, \mathtt{MLOAD}_{32}(\mathtt{ADD}(s_0, s_1)), \mathtt{MLOAD}_{27}(s_2), s_2],$
$[\mathtt{MSTORE}_{21}(s_0, \mathtt{0x11}), \underline{\mathtt{MSTORE}_{25}(\mathtt{ADD}(s_0, s_1), \mathtt{0x0E})}, \mathtt{MLOAD}_{27}(s_2),$
$\mathtt{MSTORE}_{31}(\mathtt{MLOAD}_{27}(s_2), \mathtt{0x11}), \mathtt{MLOAD}_{32}(\mathtt{ADD}(s_0, s_1))])$
$\rightarrow_{\mathtt{ADD}} ([\mathtt{ADD}(\mathtt{MLOAD}_{27}(s_2), \mathtt{0x20}), \mathtt{MLOAD}_{32}(\mathtt{ADD}(s_0, s_1)), \mathtt{MLOAD}_{27}(s_2), s_2],$
$[\mathtt{MSTORE}_{21}(s_0, \mathtt{0x11}), \mathtt{MSTORE}_{25}(\mathtt{ADD}(s_0, s_1), \mathtt{0x0E}), \mathtt{MLOAD}_{27}(s_2),$
$\mathtt{MSTORE}_{31}(\mathtt{MLOAD}_{27}(s_2), \mathtt{0x11}), \mathtt{MLOAD}_{32}(\mathtt{ADD}(s_0, s_1))])$
$\rightarrow_{\mathtt{MSTORE}} ([\mathtt{MLOAD}_{27}(s_2), s_2],$
$[\mathtt{MSTORE}_{21}(s_0, \mathtt{0x11}), \mathtt{MSTORE}_{25}(\mathtt{ADD}(s_0, s_1), \mathtt{0x0E}), \mathtt{MLOAD}_{27}(s_2),$
$\mathtt{MSTORE}_{31}(\mathtt{MLOAD}_{27}(s_2), \mathtt{0x11}), \underline{\mathtt{MLOAD}_{32}(\mathtt{ADD}(s_0, s_1))},$
$\mathtt{MSTORE}_{36}(\mathtt{ADD}(\mathtt{MLOAD}_{27}(s_2), \mathtt{0x20}), \mathtt{MLOAD}_{32}(\mathtt{ADD}(s_0, s_1)))])$

Importantly, in the heap opcodes, procedure add not only adds the heap accesses performed during the symbolic execution to mem, but it also creates a dependency list, referred to as $dep\text{-}list$(mem), which provides dependencies that are partial orders indicating the order in which accesses must happen. A dependency of the form $pp_1 < pp_2$ indicates that heap access at position pp_1 must happen before that at position pp_2. One must keep the order in which heap accesses appear in the code unless it can be proven that they are independent. The approach in Albert et al. (2022a) infers dependency information by reasoning locally (at an intra-block level) in a purely syntactical way, and it can only remove dependencies if the heap locations accessed by the opcodes in mem are known to be distinct constant values. Instead,

we will be able to refine the information in *dep-list*(mem) with global information inferred by the inter-block heap analysis with offsets in Section 2.2.

Example 6. Using the syntactic dependency analysis presented in Albert et al. (2022a) (and even the results of the heap analysis described in Section 2.1 without offsets), *dep-list* results in all heap accesses in the order of appearance in the bytecode as we do not know if the heap accesses refer to the same or distinct heap locations: *dep-list*(mem)= [(21<25), (25<27), (27<31), (31<32), (32<36)]. These dependencies restrict the possibilities of the superoptimizer for reordering the opcodes and, consequently, optimization opportunities might be lost.

The symbolic state can be optimized by `ruleBasedSimplification` before starting the search for an optimal solution. This enables optimizing the arithmetic and bit-wise opcodes and also optimizations on the heap opcodes. All simplification rules used in EVM bytecode can be seen in Albert et al. (2022b). In Section 3.3, the heap simplification rules are revised and improved with heap properties.

Example 7. No simplification can be applied to (ostack,mem) in our running example. However, if we had got ostack = [MLOAD(ADD(s_2, 0x0)), s_2], it would have been simplified to [MLOAD(s_2), s_2] using the rule ADD(x, 0) → x.

Finally, function `synthesizeOptimal` invokes a constraint solver (GASOL uses a Max-SMT solver and leaves out of the SMT encoding arithmetic and bitwise opcodes as described in Albert et al. (2020)) to search for an alternative sequence $sfopt$ of opcodes that when executed from the same input stack istack results in the same symbolic state ostack and mem and minimizes the objective function f.

Example 8. The invocation of `synthesizeOptimal` with istack, ostack and mem, obtained through the previous examples, results in the original sequence for both objective functions: gas and length, i.e., it is not possible to further optimize it since, with the current information, it is already optimal.

The returned sequence is added to the algorithm result and there is a final step that invokes `rebuild` (line 19), a standard procedure to reconstruct executable code from optimized sequences (e.g., jump addresses might need to be recomputed).

3.2. Superoptimization with useless write accesses

The sufficient condition of Albert et al. (2023) to detect a *useless write access to a memory location* made by an instruction at program point *pp* is to prove that there is no execution of the program in which such memory location is read after executing *pp*. In Albert et al. (2023), as the granularity of the heap analysis is the slot, this definition is abstracted as: a *write access to a memory location* made at *pp* is useless if there is no execution of the program in which the slot written at *pp* is read after executing *pp*. Our definition of function `computeUseless` below is an accuracy improvement of Definition 6 in Albert et al. (2023) in two aspects: its condition (i) increases the accuracy thanks to the use of offsets and condition (ii) detects also as needless write accesses those that are overwritten before being read. The latter condition requires having the certainty that the subsequent write accesses refer to the same location (Definition 3). Given a program p, we denote by $t \equiv t_0 \mapsto^* t_n$ an execution trace of p that starts from an empty memory and stack in t_0, and executing n opcodes until reaching t_n. The opcode executed at step t_i of t is referred to as op_i. We use pp:$I \in$ blocks to refer to an opcode I at pp in a sequence b, with b ∈ blocks.

Definition 5 (`computeUseless`)**.** Function `computeUseless` receives the set blocks for a program p and the result ϕ of the heap analysis (Definition 2), and returns a set of program points pp:MSTORE ∈ blocks that, in any execution trace t of p in which $op_i \equiv pp$:MSTORE, satisfy one of these conditions:
(i) for all $op_j \equiv pp_1$:MLOAD with $j>i$ in t, nonAlias(pp, s_0, pp_1, s_0) holds;
(ii) there is a subsequent $op_k \equiv pp_2$:MSTORE in t with $k>i$ s.t. safeAlias (pp, s_0, pp_2, s_0) holds, and for all $op_j \equiv pp_1$:MLOAD with $i<j<k$, nonAlias (pp, s_0, pp_1, s_0) holds.

To ensure that conditions (i) and (ii) hold in *any* execution of the program, our implementation of function `computeUseless` (invoked at line 7 in Alg. 1) relies on a reachability analysis across the blocks of the CFG of the program. Basically, given a write access, the reachability analysis checks the conditions on the read and write memory accesses of the subsequent reachable blocks in the CFG using the results of the heap analysis to ensure the `safeAlias` and `nonAlias` properties.

Example 9. The application of function `computeUseless` to the code in Fig. 1 returns useless = {13, 15, 21}, as (i) the memory location $\langle l_1, 0\rangle$ written at the two MSTORE opcodes at *pps* 13 and 21 is never read after executing such opcodes, hence they are useless; and (ii) the memory location $\langle l_1, 32\rangle$ written at the two MSTORE opcodes at *pps* 15 and 25 is read at *pp* 32 only, hence *pp* 15 was useless.

The removal of useless write accesses at the bytecode level, by an optimization tool, is far from being trivial. This is because just eliminating the MSTORE instruction may not be optimal, as the instructions used to build the memory location being accessed and the value being stored could be eliminated as well, and indeed their cost will be greater than just the MSTORE. A rule-based transformation approach that searches syntactically for patterns would not work well at the bytecode level as too many patterns would have to be added (e.g., in the leftmost column of Fig. 1, the removal of the MSTORE at *pp* 15 makes *pps* 14 and 9 unnecessary, and *pps* 8–12 will be replaced by a sequence of just two instructions, "PUSH 0x0 DUP3" that places the data on top of the stack in the right order for the MSTORE instruction at *pp* 13, saving two additional instructions). In general, one would have to infer semantically the dependencies among the instructions involved to make the MSTORE instruction. All this process comes for free using superoptimization, simply by removing the useless accesses from the symbolic memory description.

Definition 6 (*Function* `applyUseless`)**.** Function `applyUseless` receives a list of memory accesses mem and a set of program points useless, and deletes from mem the opcodes MSTORE_{pp} if $pp \in$ useless and from *dep-list*(mem) all relations $pp_1 < pp_2$ with $pp_1 \equiv pp$ or $pp_2 \equiv pp$.

Example 10. Using useless of Example 9, function `applyUseless` removes the MSTORE_{21} from mem of Example 5 and the dependency (21<25) from *dep-list* of Example 6. Given such new symbolic state, procedure `synthesizeOptimal` obtains the superoptimized sequence: "ADD PUSH 0x0E DUP2 MSTORE DUP2 MLOAD SWAP1 PUSH 0x11 DUP3 MSTORE MLOAD PUSH 0x20 DUP3 ADD MSTORE" where we can see that not only the MSTORE_{21} has been eliminated but also the instructions needed to compute the data used by MSTORE, i.e., opcodes at *pps* 19 and 20, have become unnecessary to obtain the final state. While the original sequence had 20 opcodes and 50 units of gas, the new one has respectively 17 and 46.

3.3. Superoptimization with aliasing and non-aliasing properties

Our heap analysis with offsets allows us to detect memory accesses made at different *pps* that are not aliases or that are (safe) aliases. When the heap analysis works at a slot granularity level, as in Albert et al. (2023), the fact that two instructions access the same slot (even if unique and "safe", see Definition 3) does not ensure that it is the same location within the slot. Non-aliasing can be guaranteed when the instructions access a different slot, however, the precision level is inferior to ours. Function `computeAliasing` (line 11 Alg. 1), returns the aliasing and non-aliasing pairs.

Definition 7 (computeAliasing)**.** Function computeAliasing receives a block b and the heap analysis results ϕ, and returns a pair of sets (alias, nonalias) that contain pairs (pp_i, pp_j) of program points in b defined as follows:

(1) alias=$\{(pp_i, pp_j) \mid pp_i : I_i, pp_j : I_j \in \mathsf{b} \wedge I_i, I_j \in \{\texttt{MLOAD}, \texttt{MSTORE}\} \wedge \texttt{safeAlias}(pp_i, s_0, pp_j, s_0)\}$

(2) nonalias=$\{(pp_i, pp_j) \mid pp_i : I_i, pp_j : I_j \in \mathsf{b} \wedge I_i, I_j \in \{\texttt{MLOAD}, \texttt{MSTORE}\} \wedge \texttt{nonAlias}(pp_i, s_0, pp_j, s_0)\}$

Example 11. The information inferred by our heap analysis (see Example 10) returns the following sets of pairs: alias $=\{(25, 32)\}$ and nonalias $=\{(21, 25), (21, 27), (21, 31), (21, 32), (21, 36), (25, 27), (25, 31), (25, 36), (31, 27), (31, 32), (31, 36), (36, 27), (36, 32), (27, 32)\}$. Set alias ensures that *pps* 25 and 32 are accessing the same location (l_1.b), while nonalias ensures that locations accessed at *pps* 21 and 25 are different.

The nonalias set is used to eliminate dependencies within *dep-list*(mem), similarly to what we have done with the useless set as follows.

Definition 8 (applyNonAliasing)**.** Function applyNonAliasing receives a list of memory accesses mem and the non-aliasing list nonalias and removes from *dep-list*(mem) all $(pp_1 < pp_2)$ s.t. $(pp_1, pp_2) \in$ nonalias or $(pp_2, pp_1) \in$ nonalias.

Besides this, when triggering the simplification rules for memory accesses, we can take advantage of the safe aliasing properties to leverage them at this stage.

Definition 9 (ruleBasedSimplification)**.** The ruleBasedSimplification function now receives an additional parameter alias that is used by its *memory* simplification rules (which previously used a syntactic equality check, see Albert et al. (2022a) for details):

(1) if $\texttt{MSTORE}_{pp_1}(l_1, v), \texttt{MLOAD}_{pp_2}(l_2) \in$ mem with $pp_1 < pp_2$, $(pp_1, pp_2) \in$alias and $\nexists \texttt{MSTORE}_{pp_3} \in$ mem s.t. $pp_1 < pp_3 < pp_2$ and $(pp_1 < pp_3) \in$dep-list(mem), then remove $\texttt{MLOAD}_{pp_2}$ from mem and replace all its occurrences by v in ostack and mem;

(2) if $\texttt{MLOAD}_{pp_1}, \texttt{MLOAD}_{pp_2} \in$ mem with $pp_1 < pp_2$, $(pp_1, pp_2) \in$alias and $\nexists \texttt{MSTORE}_{pp_3} \in$ mem s.t. $pp_1 < pp_3 < pp_2$ and $(pp_3 < pp_2) \in$dep-list(mem), then remove $\texttt{MLOAD}_{pp_2}$ from mem and replace all its occurrences by $\texttt{MLOAD}_{pp_1}$ in ostack and mem;

(3) if $\texttt{MLOAD}_{pp_1}(l_1), \texttt{MSTORE}_{pp_2}(l_2, \texttt{MLOAD}_{pp_1}(l_1)) \in$ mem with $pp_1 < pp_2$, $(pp_1, pp_2) \in$alias, and $\nexists \texttt{MSTORE}_{pp_3} \in$ mem s.t. $pp_1 < pp_3 < pp_2$, $(pp_1 < pp_3) \in$ dep-list(mem), then remove $\texttt{MSTORE}_{pp_2}$ from mem.

Case 1 and 2 correspond, respectively, to loading a location that has been written or read before and has not changed since then, which is ensured by the fact that there is no dependent MSTORE in between. Hence the MLOAD can be replaced by the written value (case 1) or the previous MLOAD (case 2) both at the stack and memory. Case 3 eliminates a MSTORE that is going to write the same value that was already at this memory location. Importantly, applying such aliasing and non-aliasing properties enables new bytecode optimizations, not only of the involved heap accesses, but also of stack bytecodes that can now operate more efficiently.

Example 12. Using nonalias of Example 11, applyNonAliasing removes all dependencies from dep-list(mem) of Example 6 as all pairs appear in nonalias. Then, when we apply the function ruleBasedSimplification on the state computed in Example 5 using the alias set from Example 11, we obtain as resulting state the same ostack and a new mem $=$ $[\texttt{MSTORE}_{21}(s_0, \texttt{0x11}), \texttt{MSTORE}_{25}(\texttt{ADD}(s_0, s_1), \texttt{0x0E}), \texttt{MLOAD}_{27}(s_2), \texttt{MSTORE}_{31}(\texttt{MLOAD}_{27}(s_2), \texttt{0x11}), \texttt{MSTORE}_{36}(\texttt{ADD}(\texttt{MLOAD}_{27}(s_2), \texttt{0x20}), \texttt{0x0E})]$, in which the elements that were underlined in mem in Example 5 have triggered rule 1 from Definition 9, and we have replaced all the occurrences of $\texttt{MLOAD}_{32}(\texttt{ADD}(s_0, s_1))$ by 0x0E, the value stored previously by $\texttt{MSTORE}_{25}$ (as there is no MSTORE that can write a value on location s_0+s_1). From this symbolic state, the superoptimizer returns the sequence: "PUSH 0X0E PUSH 0x11 DUP3 MSTORE SWAP2 ADD MSTORE DUP1 MLOAD PUSH 0x0E DUP2 PUSH 0x20 ADD MSTORE PUSH 0x11 DUP2 MSTORE" that removes the unnecessary MLOAD and saves 3 gas units.

3.4. Context-sensitive superoptimization

The concept of *context-sensitive superoptimization* is used to refer to superoptimizing a loop-free sequence from a given initial *calling-context*. In our case, the calling-context will be given by a heap analysis, but in general could be provided by any other type of analysis or even by a user. Our context is made up of the *aliasing context* which includes equalities among stack variables before entering the block (available in ϕ), and the *constancy context*, which includes constant information corresponding to offsets (available in σ). Function computeContext(invoked at line 12 of Alg. 1) gives us such context information on the initial state when reaching each block of the CFG of the program.

Definition 10 (computeContext)**.** Function computeContext receives a block b and the heap analysis results (ϕ, σ), and returns as *calling* context the sets: aliasing_context $= \{(s_i{=}s_j) \mid s_i, s_j \in \mathcal{S} \wedge pp_0 : I_0 = \mathsf{b}_0 \wedge \texttt{safeAlias}(pp_0, s_i, pp_0, s_j)\}$ constancy_context $= \{(s_i{=}c) \mid s_i \in \mathcal{S} \wedge pp_0 : I_0 = \mathsf{b}_0 \wedge \sigma(pp_0, s_i) = \{c\}\}$ where b_0 is the first program point in block b.

The application of the calling context is performed in the call to applyContext in line 15 of Alg. 1. Function applyContext simply applies the unifications in the aliasing_context and constancy_context to the initial and final stacks istack and ostack, and the memory mem. As usual, this is done by using a representative for each group of equal elements and replacing all of them by the representative.

Example 13. Let us show the effect of the context on sequences found in a real smart contract (FlipItBurgerIngredient, 2024). The first sequence is "PUSH 0x20 SWAP1 DUP2 MUL SWAP2 SWAP1 SWAP2 ADD ADD MSTORE SWAP3 SWAP2 POP POP" and istack with 7 elements. Symbolic execution returns (ostack $= [s_6, s_3]$, mem $= [\texttt{MSTORE}_{10}(\texttt{ADD}(\texttt{ADD}(\texttt{MUL}(\texttt{0x20}, s_0), s_1), \texttt{0x20}), s_2)]$). Function computeContext gives aliasing_context $= \emptyset$ and constancy_context $= \{(s_0 = 0)\}$, and applyContext results in istack$= [0, s_1, s_2, s_3, s_4, s_5, s_6]$, ostack as before and mem$= [\texttt{MSTORE}_{10}(\texttt{ADD}(\texttt{ADD}(\texttt{MUL}(\texttt{0x20}, 0), s_1), \texttt{0x20}), s_2)]$. This is optimized to "POP PUSH 0x20 ADD MSTORE SWAP3 SWAP2 POP POP", that has 6 fewer instructions (and saves 21 gas units). The gain has been possible because knowing that s_0 is 0 allows applying $\texttt{MUL}(x, 0) \rightarrow 0$ and $\texttt{ADD}(x, 0) \rightarrow x$. As for the aliasing context, we found this sequence from the same contract "SWAP1 PUSH 0x1F NOT DUP4 AND SWAP4 PUSH 0x03B6 PUSH 0x02 PUSH 0x0 MSTORE PUSH Z SWAP1" with istack $= [s_0, s_1, s_2, s_3]$, for which symbolic execution obtains the state ostack $= [\texttt{0x03B6}, \texttt{Z}, s_3, s_1, s_0, s_2, \texttt{AND}(s_2, \texttt{NOT}(\texttt{0x1F}))]$ and mem $= [\texttt{STORE}_{10}(0, 2)]$. Then, computeContext gives constancy_context $= \emptyset$ and aliasing_context $= \{(s_0 = s_1)\}$, and when it is applied to the previous symbolic state, we have istack $= [s_0, s_0, s_1, s_2]$, ostack$= [\texttt{0x03b6}, \texttt{Z}, s_3, s_0, s_0, s_2, \texttt{AND}(s_2, \texttt{NOT}(\texttt{0x1F}))]$ and mem $= [\texttt{MSTORE}_{10}(0, 2)]$. The optimized sequence can avoid the first SWAP and save 3 units of gas and 1 instruction.

4. Experimental evaluation

Our implementation, on one hand, extends the basic heap analysis in Albert et al. (2023) to compute the useless, alias, nonalias and context sets and, on the other hand, it also extends the GASOL superoptimizer in Albert et al. (2022a) in order to apply them, as described along the article. The implementation is in Python, open-source, and

Table 1
Experimental evaluation.

data.tex	solc+opt		gasol			gasol+heap			gains		
	#seq	*#op/s*	∞	*%len*	*time*	∞	*%len*	*time*	Δ_{∞}	$\Delta_{\%len}$	Δ_{time}
					set1: all sequences						
useless 3.2	5636	22.7	537	8.2%	0.1 s	426	23.3%	0.1 s	−111	15.1%	0.0 s
aliasing 3.3	201 354	26.5	25 232	10.4%	8.2 s	14 446	15.4%	7.3 s	−10 786	5.0%	−0.9 s
context 3.4	27 715	32.9	11 644	8.4%	30.5 s	7148	12.0%	61.6 s	−4496	3.6%	31.1 s
all 3	216 030	26.4	29 258	10.7%	7.0 s	17 342	16.3%	6.6 s	−11 916	5.6%	−0.4 s
					set2: improve of gasol+heap over gasol						
useless 3.2	3632	15.8	97	13.3%	0.1 s	0	47.0%	0.1 s	−97	33.7%	0.0 s
aliasing 3.3	33 477	34.7	10 006	12.8%	50.5 s	0	37.2%	35.8 s	−10 006	24.4%	−14.6 s
context 3.4	8661	28.3	4137	8.6%	20.4 s	0	22.9%	70.3 s	−4137	14.3%	50.0 s
all 3	43 681	32.0	13 444	12.2%	40.6 s	0	36.3%	28.1 s	−13 444	24.1%	−12.5 s

can be downloaded from https://github.com/costa-group/green where detailed instructions for its installation and usage are available. The system accepts smart contracts written in versions of Solidity up to 0.8.20 and bytecode for the Ethereum Virtual Machine v1.10.25. Experiments have been performed on an AMD Ryzen Threadripper PRO 3995WX 64-cores processor and 512 GB of memory, running Debian 5.10.70. To experimentally evaluate our approach and compare it to Albert et al. (2022a), we pulled from etherscan.io (Etherscan, 2018) the Ethereum contracts bound to the last 5000 open-source verified addresses whose source code was available on May 31, 2023. They lead to 2,013,181 sequences from 14,034 smart contracts, as each address may correspond to several Solidity files that in turn may contain several contracts in them. All contracts have been compiled enabling the `optimize` flag of `solc`. The whole dataset used is at the above github link. As the objective function, rather than measuring the gas reduction, we measure the length reduction of the code. This is because the EVM opcodes accessing the persistent memory (that we are not optimizing, see Section 5) have much larger gas cost than the others. While relying on the standard objective function, the length reduction, we give the same cost to all instructions and the results are not distorted by the particular features of the blockchain environment. Since searching for the optimal solution can be expensive for large sequences, the synthesizeOptimal procedure of GASOL can be given a timeout. The SMT solver can reach the timeout either by finding a solution (but failing to prove optimality) or without finding any solution. In the latter case, the original sequence is returned by the superoptimizer. In our experiments, as in Albert et al. (2022a), we have used a parametric timeout. There, the timeout was 10*(#store+1) sec, and here since we increase the search space with every new independence we have added, we consider a timeout of 10*(#store+#new_indeps+1) seconds.

Table 1 summarizes the experimental results in which we aim at comparing: columns under `solc+opt` (correspond to the outcome after compilation with `solc` and `optimize` flag enabled), columns `gasol` (the outcome of GASOL before our extension as appears in Albert et al. (2022a)) and `gasol+heap` (our approach). Each row corresponds to the isolated optimization presented in the corresponding section, and row `all` applies all optimizations together. Column *#seq* shows the number of sequences optimized in each case. For each type of optimization, we only analyze the sequences for which we have (new) information of the corresponding type (e.g., if the useless set does not contain any *pp* of the considered sequence, this sequence is discarded in row `useless` , while if it has non-empty alias or nonalias sets and they change the final symbolic state, it will be considered in row `aliasing` . Thus, the column *#seq* is different for the three types of optimizations and row `all` includes all previous sequences (without repetitions). Column *#op/s* shows the average number of opcodes per sequence. We show in columns ∞ the sequences for which each corresponding system reached out the time without solution. Columns *%len* display the percentage of improvement of each system (`gasol` or `gasol+heap`) wrt the original sequences (computed as $\#op/\#op_x$ *100, where *#op* is the total number of opcodes for the `solc` sequence, and $\#op_x$ for `gasol` or `gasol+heap`, in each case). Columns *time* contain the statistical median of the time needed to optimize one sequence. The set of columns `gain` provides the gains of `gasol+heap` over `gasol`: columns Δ_{∞}, $\Delta_{\%len}$ and Δ_{time} show the differences in ∞, *%len* and *time*, resp., of `gasol+heap` minus `gasol`. The results are shown for two selected subsets: `set1` includes all evaluated sequences; `set2` shows only the subset of `set1` for which our solution improves over `gasol`. Note that row `all` in `set1` includes the %seqsinsetu of the total number of sequences in the benchmark set (as mentioned above, the total are 2,013,181), which means that we are analyzing and optimizing a relevant number of them.

Fig. 3 gives more details about the distribution of the improvements gained by `gasol` and `gasol+heap` wrt `solc+opt` for the blocks included in `set2`. It shows one plot for each row in Table 1, namely, plot (i) shows the distribution of the improvements gained by adding `useless` information; plot (ii) shows the gains of `aliasing`; plot (iii) the gains of `context`; and plot (iv) the gains of `all` the information together. The plots depicted in Fig. 3 show the number of blocks (axis y) that gain the corresponding percentage wrt the initial size of the block (axis x).

Let us draw the main conclusions from the figures. Comparing *#seq* for `set1` and `set2`, we can see that the number of sequences that reduce their length with `gasol+heap` is very significant in row `useless` (3632 of 5636), while it is smaller in `aliasing` (33,477 of 201,354) and moderate in `context` (8661 of 27,715). In `aliasing` , we have observed that often the order of accesses to the heap made by the compiler is already optimal, and non-aliasing properties cannot hence improve the results. As regards the gains in length reduction, in `set1`, we can see in column *%len* that the overall improvement of `gasol+heap` is important, being the lowest 12.0% and the highest 23.3%. Such gains are very significant for those sequences in `set2`, and the gains of `gasol+heap` wrt `gasol` (column $\Delta_{\%len}$) are between 14.3% and 33.7%. As expected, because we are removing `MSTORE` opcodes and their dependencies, setting useless improves the most but can be applied many less times. As regards the optimization times, comparing columns *time* for `gasol` and `gasol+heap` for both `set1` and `set2`, we can see that, the time needed to superoptimize each sequence is not increased by `gasol+heap` in row `useless` and is even reduced in row `aliasing` . This reduction is low in `set1` since, as mentioned, in many cases there is no real room for improvement, but it shows to be relevant in `set2`, where `gasol+heap` has improved the results. This is due to the fact that the superoptimizer can take advantage of the rule-based simplification and hence achieve better results in less time. On the contrary, the time needed to treat the context information in both sets is significantly higher. There are two reasons to explain that: (1) adding context information increases the search space since new options to generate constants or duplicate stack values appear, (2) the constant information, unlike what happens with aliasing , cannot be often used for the heap rules simplification and just increases the search space. We have checked that 13,194 of the 27,715 sequences that cause the great difference in time in `context`

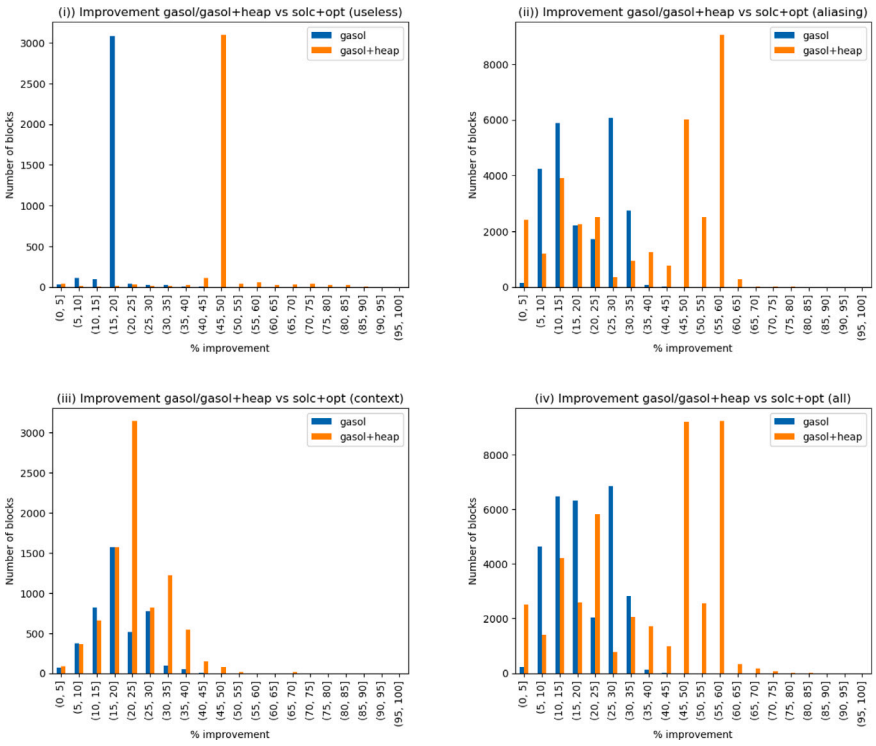


Fig. 3. Distribution of improvements using `gasol` and `gasol+heap` wrt `solc`.

are also in `aliasing` . Importantly, when context and aliasing are applied together, thanks to the additional aliasing information, they do not cause extra time.

In addition, observing Fig. 3, we can see that most of the improvements gained by `gasol` range between 10%–35% in all plots and in any of `gasol`'s plots there are improvements over 40%. The plots show that the behavior is better in all optimizations performed by `gasol+heap`, showing a significant number of blocks with improvements over 40%. In plot (i), which uses the `useless` information, most of the blocks show the same improvements, in the case of `gasol` in range (15,20] and for `gasol+heap` in (45,50]. The reason is that most blocks that contain useless memory instructions contain the same sequence of instructions, namely they correspond to the block created by the compiler for allocating memory and are the same in most of the programs.

Finally, it is worth mentioning that, although there are sequences in ∞ of `set1` using `gasol+heap` that are not in ∞ of `gasol`, the number of sequences in ∞ is reduced in all cases (in this column the only relevant information is given for `set1` since, as described, `set2` does not contain any sequence in ∞ with `gasol+heap`). To sum up, the overall figures in rows `all` experimentally prove that harnessing heap analysis for superoptimization can reach gains in a significant number of sequences (43,681 of 216,030), and for those sequences (`set2`) produce important savings (36.3% wrt the original number of opcodes and 24.1% wrt the basic `gasol`) while the optimization time is reduced (−12.5 s). Moreover, considering that globally `set1` contains the %seqsinsetu of the sequences of code of all contracts in the benchmark set, it is specially relevant that have we reach gains of 16.3% (`all`) on sequences already optimized by compiler.

Gas optimization. The experimental evaluation is performed using the length of the blocks as objective function, in order to avoid having to explain the details of the EVM gas model, which is a non-classical cost model. However, the gas-improvement of GASOL+heap wrt GASOL is proportional to the improvement obtained for the length of the blocks. This can be explained theoretically: the instructions we are optimizing (stack and memory opcodes) have almost the same cost (2–3 gas units) as it happens when minimizing length (cost 1). The only relevant consideration is the fact that there are some instructions whose *gas* cost is significantly larger than common instructions (e.g., `SLOAD` might consume up to 2100 *gas* units, and `SSTORE` might consume up to 22 100 *gas* units). The presence of storage accesses in blocks leads to lower percentages of improvement even when GASOL+Heap is able to remove a significant number of instructions. Nevertheless, even considering that the percentage of gas improvement is lower than the one obtained for length, using the benchmarks results for `all` of `set1` from Table 1, the gains of GASOL+Heap over `solc` and its optimizations are 0.61% in gas-reduction. Hence, using the next data taken from Etherscan (2018) on February 29th, 2024[4]:

1. Eth price: 3342.24 USD/ETH
2. Avg. Gas Price: 0.000000074113151775 ETH/gas
3. Gas used: 108,601,400,000 *gas* units
4. Number of transactions: 1,236,458

We obtain that the 0.61% of gas optimized by GASOL+Heap wrt `solc` would have saved 124,862.1738 USD on that day. Assuming the same data for all days in a year, it would amount to >45M USD/year. Note that we have removed the gas intrinsic cost, which does not depend on the instructions (25,965,618,000). It corresponds to 21,000 *gas* units that are paid for each transaction. Hence, the gas considered in the code operations is 82,635,782,000.

5. Related work

The first static modeling of the EVM memory is Grech et al. (2022), which is formalized as a Datalog analysis. The accuracy of Grech et al. (2022) is later increased in Albert et al. (2023), which presents a flow-sensitive analysis able to accurately model the memory allocated by nested data structures when memory locations contain pointers to other slots, while Grech et al. (2022) does not capture such type of

[4] Date of writing.

structures. Our heap analysis with offsets makes a further increase of precision, as it reasons at the level of locations within the slot, rather than at the level of slots only as in Albert et al. (2023). This further precision is required to infer aliasing and non-aliasing properties that can be exploited by an optimization tool, and also provides further precision for the detection of useless write accesses and for the context information. While we have applied our heap analysis for program optimization, a precise model of the EVM memory is crucial to enhance the accuracy of any posterior analysis (see, e.g., Lagouvardos et al. (2020)).

The relevance of optimizing smart contracts has been pointed out in Brandstätter et al. (2020), Nagele and Schett (2019), Nelaturu et al. (2021) and Zhao et al. (2023) and guidelines for writing more efficient code have been also provided for smart contract developers (Chen et al., 2017, 2020; Kong et al., 2022). It is hence not surprising that optimization of EVM smart contracts is an active research topic: we can also find approaches to optimization that are applied on the (Solidity) source-code level (e.g., Chen et al. (2022)). EVM bytecode optimization is also performed by the standard Solidity compiler `solc` (Optimizer, 2023) which is able to perform certain types of inlining, of dead-code elimination, etc. Superoptimization is complementary to these approaches and it is usually applied after having enabled them as a final code optimization stage. Hence, we do not compare our experimental results to them but rather apply our technique after having applied the available optimizer (note that Chen et al. (2022) is not publicly available).

Comparing our work to existing superoptimizers, we found that most of the existing tools do not include heap optimizations, the only exceptions being Albert et al. (2022a) and Bansal and Aiken (2006). As regards (Albert et al., 2022a), it is only able to perform some memory optimizations at the level of each block of the CFG, while the fact that we use a global heap analysis within the superoptimizer breaks such local limitation, as we have explained in detail through the article. In particular, the detection of useless write accesses would not be possible without a whole-program analysis, and also the heap properties that can be inferred by reasoning at an intra-block level are very scarce. Instead, combining a global heap analysis with an intra-block superoptimizer overcomes this limitation and achieves very powerful heap optimizations as proven in our experimental results. The approach to superoptimize memory operations that we are aware of is that of Bansal and Aiken (2006), which is based on a testing equivalence check and thus differs substantially from SMT-based approaches (following Algorithm 1) like ours. Finally, in addition to dynamic memory, smart contracts also use a persistent memory called storage. For storage, there is no need to develop a static analysis to detect the slots, as they are statically known. However, the inference of the offsets within the slots requires our extension in Section 2.2, which can be done analogously. Condition i) of function `applyUseless` in Section 3.2 is not applicable because, as storage is persistent memory, a write storage access is not removable even if there is no further read access. Aliasing properties and the inference of the context can be done as we have done in Sections 3.3 and 3.4 for the heap. Experienced smart contract developers are aware though of the high cost of using storage and make a very efficient use of it and finding optimizations will be much less frequent than for the heap.

6. Discussion and limitations

Superoptimization (Massalin, 1987; Jangda and Yorsh, 2017) has some limitations that are inherent to the definition of the superoptimization technique itself. First, the technique is applied on loop-free sequences of code. Therefore, it is not able to achieve some of the classical optimizations in loops, such as loop unfolding. Second, the technique searches for the optimal code, what requires exploring the whole space of alternative sequences that result in an equivalent code. This exhaustive search may threaten scalability, and sometimes timeouts are used and rather than returning the optimal code, one obtains the best code found within the timeout.

When compared to the use of design patterns (Chen et al., 2020; Mariano et al., 2022; Marchesi et al., 2020; Nguyen et al., 2022), as well as other forms of source-code optimization techniques (Brandstätter et al., 2020; Chen et al., 2017, 2018; Liu and Song, 2024), we note that their effectiveness should not be compared to that of superoptimization: Superoptimization is intended to be applied as a final layer of optimization on code that has been already optimized at the source level and even optimized by the compiler. Given such (already optimized) code and applied at the level on loop-free sequences, superoptimization will be able to find optimal translations of the code. Therefore, superoptimization is compatible and complementary to other optimization techniques.

As regards the heap analysis, its main limitation is that is tailored specifically for EVM bytecode generated by the Solidity compiler, solc, and its application to code generated by other compilers would require some adaptations. For example, the EVM bytecode compiled from Vyper (2024) has a completely different memory layout compared with the one generated by solc. While the EVM generated by solc uses a specific memory address (`0x40`) to store the free memory pointer (as explained in Section 2), the one generated by Vyper does not implement this free memory pointer. Vyper has a more restrictive approach, where all the local variables are stored in memory (in Solidity are stored in the stack) and the size of the data structures stored in memory has to be known at compilation time. Hence, the notion of slot should be redefined in this context. The Fe language (Fe, 2024) is also compiled into EVM bytecode. In this case, the memory layout is similar to the one generated by solc with the exception that the free memory pointer is stored at position `0x00` instead of `0x40` and, hence, the adaptation is straightforward. The Yul language (Yul, 2024) is an intermediate language that can be compiled to bytecodes for different backends, including EVM bytecode. In this case, Yul compiler is part of solc and, therefore, can generate the same memory layout. It should be noted that solc is by far the most used Solidity compiler. According to Etherscan, 99.61%[5] of the smart contracts have been compiled with some version of solc. Therefore, our heap analysis has wide applicability.

Besides the Ethereum context, our overall approach of combining heap analysis and superoptimization is general and could be applied to other blockchains (Hyperledger, 2024; Solana, 2024; Neo, 2024). Nevertheless, there are two aspects that should be considered when implementing this approach in other contexts. On the one hand, the heap analysis should be adapted to the particular details of the memory model of the smart contract language, as already mentioned even within the context of the EVM language. For instance, Hyperledger Fabric smart contracts can be programmed using Java as well as other languages. In those cases, the particularities of the language used and the bytecode generated should be studied for developing the heap analysis accordingly. On the other hand, the nature of the blockchain determines the cost model to be used for the superoptimizer. In the case of permissioned blockchains such as Hyperledger Fabric, there is no notion of gas consumption or the gas price is set to zero and therefore other metrics should be used instead.

7. Conclusions

This article has proposed a seamless integration of a whole-program heap analysis into a superoptimizer for local sequences of opcodes. Superoptimization tools first extract from the original sequence a higher-level specification which is used to carry out the search for the optimal solution. In our case, such specification includes a symbolic description

[5] Data gathered from https://etherscan.io/dashboards/contract-statistics on Sep 25,2024.

of the operand stack and of the heap contents. The integration of heap analysis into superoptimization is made at the level of the symbolic description and does not affect the search engine: (1) needless write accesses identified using the heap analysis outcome are eliminated from the memory description; (2) aliasing and non-aliasing properties can enable the simplification of the memory description; (3) calling context information can simplify both the operand stack and memory descriptions. Our thorough experimental evaluation on sequences taken from 14,034 real smart contracts shows that, when heap properties can be exploited, optimization gains wrt the previous version of the superoptimization tool improve from 10.7% length-reduction to 16.3% optimization while the median of the optimization time per sequence is reduced by 0.4 s

CRediT authorship contribution statement

Elvira Albert: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Jesús Correas:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Pablo Gordillo:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Guillermo Román-Díez:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Albert Rubio:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Guillermo Roman-Diez reports financial support was provided by Spain Ministry of Science and Innovation. Elvira Albert, Jesus Correas, Pablo Gordillo, Albert Rubio reports financial support was provided by Spain Ministry of Science and Innovation. If there are other authors, they 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 work was partially funded by the Spanish MCI, AEI and FEDER (EU) projects PID2021-122830OB-C41 and PID2021-122830OA-C44, and by the Ethereum Foundation project GREEN.

Data availability

No data was used for the research described in the article.

References

Albert, Elvira, Correas, Jesús, Gordillo, Pablo, Román-Díez, Guillermo, Rubio, Albert, 2023. Inferring needless write memory accesses on ethereum smart contracts. In: 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023. Proceedings. In: Lecture Notes in Computer Science, vol. 13993, Springer, pp. 448–466.

Albert, Elvira, de la Banda, Maria Garcia, Hernández-Cerezo, Alejandro, Ignatiev, Alexey, Rubio, Albert, Stuckey, Peter J., 2024. SuperStack: Superoptimization of stack-bytecode via greedy, constraint-based, and SAT techniques. Proc. ACM Program. Lang. 8 (PLDI), 1437–1462.

Albert, Elvira, Gordillo, Pablo, Hernández-Cerezo, Alejandro, Rubio, Albert, 2022a. A max-SMT superoptimizer for EVM handling memory and storage. In: Fisman, Dana, Rosu, Grigore (Eds.), 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2022. Proceedings. In: Lecture Notes in Computer Science, vol. 13243, Springer, pp. 201–219.

Albert, Elvira, Gordillo, Pablo, Hernández-Cerezo, Alejandro, Rubio, Albert, Schett, Maria A., 2022b. Super-optimization of smart contracts. ACM Trans. Softw. Eng. Methodol. 31 (4), 70:1–29.

Albert, Elvira, Gordillo, Pablo, Rubio, Albert, Schett, Maria A., 2020. Synthesis of super-optimized smart contracts using max-SMT. In: 32nd International Conference on Computer Aided Verification, CAV 2020. Proceedings. In: Lecture Notes in Computer Science, vol. 12224, pp. 177–200.

Bansal, Sorav, Aiken, Alex, 2006. Automatic generation of peephole superoptimizers. In: Shen, John Paul, Martonosi, Margaret (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, pp. 394–403.

Berdine, Josh, Calcagno, Cristiano, Cook, Byron, Distefano, Dino, O'Hearn, Peter W., Wies, Thomas, Yang, Hongseok, 2007. Shape analysis for composite data structures. In: Damm, Werner, Hermanns, Holger (Eds.), Computer Aided Verification. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 178–192.

Brandstätter, Tamara, Schulte, Stefan, Cito, Jürgen, Borkowski, Michael, 2020. Characterizing efficiency optimizations in solidity smart contracts. In: IEEE International Conference on Blockchain, Blockchain 2020, Rhodes, Greece, November 2-6, 2020. IEEE, pp. 281–290.

Cabrera-Arteaga, Javier, Donde, Shrinish, Gu, Jian, Floros, Orestis, Satabin, Lucas, Baudry, Benoit, Monperrus, Martin, 2020. Superoptimization of WebAssembly bytecode. In: Aguiar, Ademar, Chiba, Shigeru, Boix, Elisa Gonzalez (Eds.), Programming'20: 4th International Conference on the Art, Science, and Engineering of Programming, Porto, Portugal, March 23-26, 2020. ACM, pp. 36–40.

Chen, Ting, Feng, Youzheng, Li, Zihao, Zhou, Hao, Luo, Xiapu, Li, Xiaoqi, Xiao, Xiuzhuo, Chen, Jiachi, Zhang, Xiaosong, 2020. GasChecker: Scalable analysis for discovering gas-inefficient smart contracts. IEEE Trans. Emerg. Top. Comput. PP(99), 1–14.

Chen, Ting, Li, Xiaoqi, Luo, Xiapu, Zhang, Xiaosong, 2017. Under-optimized smart contracts devour your money. In: SANER. IEEE Computer Society, pp. 442–446.

Chen, Ting, Li, Zihao, Zhou, Hao, Chen, Jiachi, Luo, Xiapu, Li, Xiaoqi, Zhang, Xiaosong, 2018. Towards saving money in using smart contracts. In: Zisman, Andrea, Apel, Sven (Eds.), Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018. ACM, pp. 81–84.

Chen, Yanju, Wang, Yuepeng, Goyal, Maruth, Dong, James, Feng, Yu, Dillig, Isil, 2022. Synthesis-powered optimization of smart contracts via data type refactoring. Proc. ACM Program. Lang. 6 (OOPSLA2), 560–588.

ERC721A. https://etherscan.io/address/0xfcd5c0ef90715dc052dad6de08efda758aa09f60#code.

2018. Etherscan. https://etherscan.io.

2024. Fe documentation. https://fe-lang.org/docs/index.html.

FlipItBurgerIngredient. https://etherscan.io/address/0x014BDf5237C49fA2B1283AaDE4dB4f78C4C11777#code.

Gavin Wood, 2019. Ethereum: A secure decentralised generalised transaction ledger.

Grech, Neville, Lagouvardos, Sifis, Tsatiris, Ilias, Smaragdakis, Yannis, 2022. Elipmoc: Advanced decompilation of ethereum smart contracts. Proc. ACM Program. Lang. 6 (OOPSLA), 77:1–77:27.

Hyperledger Fabric Documentation. https://hyperledger-fabric.readthedocs.io/en/latest/.

Itzhaky, Shachar, Bjørner, Nikolaj, Reps, Thomas, Sagiv, Mooly, Thakur, Aditya, 2014. Property-directed shape analysis. In: Biere, Armin, Bloem, Roderick (Eds.), Computer Aided Verification. Springer International Publishing, Cham, pp. 35–51.

Jangda, Abhinav, Yorsh, Greta, 2017. 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. pp. 78–88.

Kong, Queping, Wang, Zi-Yan, Huang, Yuan, Chen, Xiangping, Zhou, Xiao-Cong, Zheng, Zibin, Huang, Gang, 2022. Characterizing and detecting gas-inefficient patterns in smart contracts. J. Comput. Sci. Tech. 37 (1), 67–82.

Lagouvardos, Sifis, Grech, Neville, Tsatiris, Ilias, Smaragdakis, Yannis, 2020. Precise static modeling of Ethereum "Memory". Proc. ACM Program. Lang. 4 (OOPSLA), 190:1–190:26.

Liu, Yunqi, Song, Wei, 2024. FunRedisp: A function redispatch tool to reduce invocation gas fees in solidity smart contracts. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2024, Association for Computing Machinery, New York, NY, USA, pp. 1876–1880.

Marchesi, Lodovica, Marchesi, Michele, Destefanis, Giuseppe, Barabino, Giulio, Tigano, Danilo, 2020. Design patterns for gas optimization in ethereum. In: 2020 IEEE International Workshop on Blockchain Oriented Software Engineering. IWBOSE, pp. 9–15.

Mariano, Benjamin, Chen, Yanju, Feng, Yu, Durrett, Greg, Dillig, Isil, 2022. Automated transpilation of imperative to functional code using neural-guided program synthesis. Proc. ACM Program. Lang. 6 (OOPSLA1), 1–27.

Massalin, Henry, 1987. 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, pp. 122–126.

Milanova, Ana, Rountev, Atanas, Ryder, Barbara G., 2005. Parameterized object sensitivity for points-to analysis for java. ACM Trans. Softw. Eng. Methodol. 14, 1–41.

Mukherjee, Manasij, Kant, Pranav, Liu, Zhengyang, Regehr, John, 2020. Dataflow-based pruning for speeding up superoptimization. Proc. ACM Program. Lang. 4 (OOPSLA), 177:1–177:24.

Nagele, Julian, Schett, Maria A., 2019. Blockchain superoptimizer. In: Preproceedings of 29th International Symposium on Logic-Based Program Synthesis and Transformation. LOPSTR 2019.

Nelaturu, Keerthi, Beillahi, Sidi Mohamed, Long, Fan, Veneris, Andreas G., 2021. Smart contracts refinement for gas optimization. In: 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services, BRAINS 2021, Paris, France, September 27-30, 2021. IEEE, pp. 229–236.

Neo Blockchain Documentation. https://docs.neo.org/docs/index.html.

Nguyen, Quang-Thang, Son, Do Bao, Nguyen, Thi Tam, Do, Ba-Lam, 2022. GasSaver: A tool for solidity smart contract optimization. In: Gai, Keke, Choo, Kim-Kwang Raymond (Eds.), BSCI 2022: Proceedings of the 4th ACM International Symposium on Blockchain and Secure Critical Infrastructure, Nagasaki, Japan, May 30, 2022. ACM, pp. 125–134.

2023. Optimizer of solidity compiler. https://docs.soliditylang.org/en/latest/internals/optimizer.html#optimizer.

Sasnauskas, Raimondas, Chen, Yang, Collingbourne, Peter, Ketema, Jeroen, Taneja, Jubi, Regehr, John, 2017. Souper: A synthesizing superoptimizer. CoRR, abs/1711.04422.

Smaragdakis, Yannis, Bravenboer, Martin, Lhoták, Ondrej, 2011. Pick your contexts well: Understanding object-sensitivity. In: Ball, Thomas, Sagiv, Mooly (Eds.), Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 17–30.

Solana Blockchain Documentation. https://solana.com/es/docs.

2024. Solc compiler. https://docs.soliditylang.org/en/latest/using-the-compiler.html.

Vyper. https://github.com/ethereum/vyper.

2024. Yul documentation. https://docs.soliditylang.org/en/latest/yul.html.

Zhao, Ziyi, Li, Jiliang, Su, Zhou, Wang, Yuyi, 2023. GaSaver: A static analysis tool for saving gas. IEEE Trans. Sustain. Comput. 8 (2), 257–267.

Elvira Albert (http://costa.fdi.ucm.es/~elvira) is a professor with the Complutense University of Madrid, Spain, where she leads a research group, the COSTA team, currently made up of seven senior researchers and four postdoctoral researchers. She is author of around 150 publications in international journals and volumes related to the topics of validation and verification of sequential and concurrent programs. She has been invited speaker with major conferences such as the ICST, ICLP, LPAR, and FM.

Jesús Correas (http://costa.fdi.ucm.es/~jcorreas) is an Associate Professor at Complutense University of Madrid since 2009. He was previously employed as an Assistant Professor at the same university and at Universidad Politécnica de Madrid, and he has also worked in private software development companies. His main research interests are static program analysis, constraint declarative programming and object-oriented concurrent and distributed systems. He has been involved in several national and European research projects.

Pablo Gordillo (http://costa.fdi.ucm.es/~pabgordi) is an Assistant Professor at the Complutense University of Madrid (UCM) since 2021. He was previously employed as Postdoc Researcher at the same university. He is a research member of the COSTA Group (http://costa.fdi.ucm.es since 2014. His research interests include static and dynamic analyses, formal methods, testing and verification of concurrent programs and distributed systems. He is currently working on analysis and verification of Ethereum smart contracts. He has been involved in several national and European research projects.

Guillermo Román-Díez (http://costa.ls.fi.upm.es/groman) is Associate Professor at Universidad Politécnica de Madrid since 2020. He was previously employed as Assistant Professor and Postdoc Researcher at the same university and have also worked in private companies developing software projects. His main research interests are program analysis, namely, static resource analysis, object-oriented, concurrent/distributed programs, and blockchain systems. He has been involved in several national and European research projects.

Albert Rubio (http://costa.fdi.ucm.es/~albert) received the Ph.D. degree in computer science from the Technical University of Catalonia, in 1994. He is professor with the Complutense University of Madrid since 2019. He was previously full professor with the Technical University of Catalonia since 2008. He is a author of around 70 papers published in the most prestigious conferences and journals, including LICS, CAV or the Journal of ACM. He is a coauthor of a chapter of the Handbook of Automated Reasoning.