Synthesis of Sound and Precise Storage Cost Bounds via Unsound Resource Analysis and Max-SMT

Elvira Albert
elvira@sip.ucm.es
Complutense University of Madrid
Spain

Jesús Correas
jcorreas@ucm.es
Complutense University of Madrid
Spain

Pablo Gordillo
pabgordi@ucm.es
Complutense University of Madrid
Spain

Guillermo Román-Díez
guillermo.roman@upm.es
Universidad Politécnica de Madrid
Spain

Albert Rubio
alberu04@ucm.es
Complutense University of Madrid
Spain

Abstract

A storage is a persistent memory whose contents are kept across different program executions. In the *blockchain* technology, storage contents are replicated and incur the largest costs of a program's execution (a.k.a. *gas fees*). Storage costs are *dynamically* calculated using a rather complex model which assigns a much larger cost to the first access made in an execution to a storage key, and besides assigns different costs to *write* accesses depending on whether they change the values *wrt.* the initial and previous contents. Safely assuming the largest cost for all situations, as done in existing gas analyzers, is an overly-pessimistic approach that might render useless bounds because of being too loose. The challenge is to soundly, and yet accurately, synthesize storage bounds which take into account the dynamicity implicit to the cost model. Our solution consists in using an off-the-shelf static resource analysis —but do not always assuming a worst-case cost— and hence yielding *unsound* bounds; and then, in a posterior stage, computing corrections to recover soundness in the bounds by using a new Max-SMT based approach. We have implemented our approach and used it to improve the precision of two gas analyzers for Ethereum, `gastap` and `asparagus`. Experimental results on more than 400,000 functions show that we achieve great accuracy gains, up to 75%, on the storage bounds, being the most frequent gains between 10-20%.

CCS Concepts

• **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Automated static analysis**.

Keywords

Smart contracts, Ethereum blockchain, Static analysis, Resource analysis, SMT solver.

ACM Reference Format:
Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2024. Synthesis of Sound and Precise Storage Cost Bounds via Unsound Resource Analysis and Max-SMT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3680352

This work is licensed under a Creative Commons Attribution 4.0 International License.

ISSTA '24, September 16–20, 2024, Vienna, Austria

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680352

1 Introduction

The problem. The notion of *storage* is used to refer to a persistent memory whose contents are maintained across different executions of the program. This notion, which is inherent to (distributed) databases, is also used today in the *blockchain* ecosystem, in which storage accesses often incur the largest costs of a program's execution (a.k.a. *gas fee* of the transaction). Naturally, a cost model that determines the cost of the different instructions in the programming language will assign larger cost to *persistent* storage access than to *volatile* memory access. In the context of the blockchain, this (much) larger storage cost is clearly justified by the replication of contents of the storage required to implement the consensus protocols. Besides, as in distributed databases [39], not all accesses to the storage cost the same: on one hand, the first access to a *storage location* (denominated *key* in what follows) requires retrieving the value from the global storage and costs notably more, regardless of whether it is a read or write access; on the other hand, when writing a value, the cost might depend also on whether it is a fresh new allocation (costs a lot), if we are not modifying the initial value (costs little), or if the initial value is modified but it was already allocated (a cost in-between). This intuition is the rationale behind the storage cost model used in Ethereum-based blockchains [1–3]. Let v_0 be the initial value, v the current value and v' the value to be assigned to a key. The cost model for storage costs in Ethereum [37] precisely distinguishes the following cases:

(i) *c(cold)=2,100*: cost of the first access (load or store);
(ii) *c(warm-load)=100*: cost of next (non-first) read accesses;
(iii) *c(set-store)=20,000*: if $v \neq v' \wedge v_0 = v \wedge v_0 = 0$, i.e., a store in which the key contained initially a zero when the execution began (which means in Ethereum that it was not allocated), the current value before the store is (again) a zero (as initially) and it is set to a value different from zero (intended to capture high price for new allocations);
(iv) *c(reset1-store)=2,900*: if $v \neq v' \wedge v_0 = v \wedge v_0 \neq 0$, i.e., when the key contained initially a non-zero value (i.e. it is already allocated) and the current value before the store is (again) the initial one and we are changing this value.

(v) *c(reset2-store)=100*: if $v = v' \vee v_0 \neq v$, i.e., when the initial value has already been changed or we are reassigning the same value that is currently in the key (intended to capture that a *set* or *reset1* for the change has already been paid before);

As seen above, the differences in the costs of each type (i)-(v) are enormous and safely assuming the largest cost (*c(cold)*+*c(set-store)* = 22,100 for stores and *c(cold)* = 2,100 for loads), as done in existing gas analyzers [7, 12] and also in the gas estimator of the Solidity compiler solc, is an overly-pessimistic approach that might render useless bounds, as they are often too far from the actual cost. The problem investigated in the paper is the synthesis of storage bounds for the above dynamic cost model, statically obtained from the code, using a sound and yet precise approach. In particular, we aim at synthesizing a storage bound of the form:

$$\boxed{UB = ub_i{*}2{,}100{+}ub_{ii}{*}100{+}ub_{iii}{*}20{,}000{+}ub_{iv}{*}2{,}900{+}ub_v{*}100}$$

which is larger or equal than the actual cost of any execution of the program, where ub_x denotes an upper bound on the number of each case $x \in \{i, .., v\}$. Each ub_x can be constant (including zero) but in the general case can be a symbolic expression (e.g., when storage instructions appear inside loops). The main difficulty of the problem is to accurately associate and assign each storage access to (safe) costs $(i)-(v)$, in particular, because as storage opcodes can appear within conditionals and loops, their type might not be the same across execution. A simple case is a load to a single key within a loop that makes n iterations that would cost *c(cold)*+$(n-1)$∗*c(warm-load)*, i.e., $ub_i = 1, ub_{ii} = n-1, ub_{iii} = ub_{iv} = ub_v = 0$. A more contrived case would be a store in the branch of a conditional statement of a loop, as in `for(i=0;i<n;i++){if (*) key=*;}`, where we use `*` to ignore the code and `key` is a storage location. In this case, as we will justify through the paper, we aim at inferring that the worst case cost of this loop is bound by *c(cold)*+$\lceil n/2 \rceil$∗*c(set-store)*+$\lfloor n/2 \rfloor$∗*c(reset2-store)*, i.e., $ub_i = 1$, $ub_{ii} = n-1$, $ub_{iii} = \lceil n/2 \rceil$ and $ub_v = \lfloor n/2 \rfloor$ (and $ub_{iv} = 0$). This is a safe and accurate sound bound because – statically, without prior knowledge on the initial value of the key nor the previous value– the worst case occurs when `key` initially contains zero and in the loop `key` alternates between a zero and any non-zero value.

The solution. The verification problem has two components: (a) a storage analysis which is able to detect the keys that each store/load instruction is accessing and (b) the resource analysis problem that given the inferred keys computes a sound upper bound for the storage costs. As regards (a), it is a type of memory analysis, a problem that has been well-studied in the literature for different languages (e.g., pointer and shape analysis for C [9], object-oriented languages [27, 33], or for Ethereum volatile memory [8, 18, 22, 26]). The particularities of the Ethereum's storage analysis are related to inferring the keys when dynamic arrays and mappings are accessed, as such keys are computed by applying hash functions. We present the basis of a storage analysis to statically infer such more contrived storage accesses. After this, as in any memory analysis, the output of the storage analysis is a function that, for each storage instruction in the code, associates to it either a unique key, or a *disjunction* of keys that this instruction might access, or $\top$ denoting the lack of information about the key being accessed.

The important challenge and our main contribution is related to component (b). Our solution is to rely on an off-the-shelf resource analyzer that works for static cost models but, rather than assigning the largest cost as in `gastap` and `asparagus`, we assign *c(warm-load)* for load and *c(reset1-store)* for store, when the keys are not $\top$. Using such a cost model, the analysis will obviously yield an *unsound* bound, named UB_u, which will be "corrected" in a posterior stage. For a $\top$ key, it is unavoidable to assume a worst-case cost *c(cold)* for load and *c(cold)*+*c(set-store)* for store since we have lost track of the accessed key and we will not be able to correct these cases. Essentially, the correction problem has two dimensions: (1) inferring the worst-case number of (non $\top$) cold accesses and obtaining a corresponding correction C_{cold} for the unsound assumption on them, and (2) detecting the worst-case number of (non $\top$) set stores and obtaining a corresponding correction C_{set} for the corresponding unsound gap. This is a challenging problem as it requires finding the trace that has the worst case in number of cold accesses and extra cost in store operations. We solve it by means of a new Max-SMT encoding that requires finding the optimal solution. Once we have obtained C_{cold} and C_{set} the sound upper bound is obtained as $UB = UB_u + C_{cold} + C_{set}$.

Contributions. The main contributions of this paper are: (1) We develop a storage analysis for the storage layout generated by the most-used Solidity compiler, `solc`. Our analysis is able to infer the keys accessed by the storage opcodes, even for nested data structures which are accessed by means of hash functions. (2) We introduce a new approach to accurately infer storage bounds based on using an off-the-shelf resource analyzer with a cost model that, unless there is no information on the accessed key, does not assume the worst-case cost –and hence yields unsound results– which are then corrected in a posterior stage. (3) We present a novel Max-SMT encoding to compute the soundness corrections to the bounds obtained by using resource analysis in an unsound way. The problem amounts to finding the worst case cost trace in number of cold accesses and also in worst cost of the store operations of all possible executions. (4) We undertake our experimental evaluation by integrating our extensions within two existing resource analyzers: `gastap` [7] and `asparagus` [12]. Our experiments on more than 400,000 functions obtained from more than 37,000 smart contracts show that we greatly improve the storage bounds obtained by these systems. The improvements range from 2.5% to 80% (achieving on average between 13-17% of further precision).

2 Language and Storage Costs

Ethereum's cost model [37] is given at the level of its bytecode language, called EVM. Hence, gas analyzers for smart contracts are defined on the bytecode language that we briefly describe in Sec. 2.1 and which is also used to define our analysis. While our work focuses on inferring the storage costs, in Sec. 2.2, we overview the main features of the full EVM gas model, as it is part of the `gastap` and `asparagus` systems on which we perform our experimental evaluation. Finally, Sec. 2.3 formalizes the storage costs that we aim at statically approximating later in Sec. 3.

2.1 Bytecode Language

To formalize our approach, we use an EVM-like bytecode language with two basic opcodes to access the storage: LOAD and STORE to, resp.,

$\langle S, G\rangle \rightarrow_{\texttt{POP}} \langle S[1{:}n], G\rangle$
$\langle S, G\rangle \rightarrow_{\texttt{DUPx}} \langle [S[\texttt{x}-1] \mid S], G\rangle$
$\langle S, G\rangle \rightarrow_{\texttt{SWAPx}} \langle S', G\rangle, S'[0] = S[\texttt{x}], S'[\texttt{x}] = S[0], S'[i] = S[i]\ \forall i > 0, i \neq \texttt{x}$
$\langle S, G\rangle \rightarrow_{\texttt{ADD}} \langle [(S[0]{+}S[1]) \mid S[2{:}n]], G\rangle$
$\langle S, G\rangle \rightarrow_{\texttt{LOAD}} \langle [G[S[0]] \mid S[1{:}n]\ ], G)\rangle$
$\langle S, G\rangle \rightarrow_{\texttt{STORE}} \langle S[2{:}n], G'\rangle, G'[S[0]] = S[1]G'[i] = G[i]\ \forall i \geq 0, i \neq S[0]$
$\langle S, G\rangle \rightarrow_{\texttt{KECCAK}} \langle [keccak(S[0]) \mid S[1{:}n]], G\rangle$

Figure 1: Semantics of Selected Opcodes

read and write from it; stack-manipulating opcodes (e.g., DUP, SWAP, POP); arithmetic and bit operations (e.g., ADD); control-flow opcodes (e.g., JUMP); and KECCAK256 that computes a 256-bit cryptographic hash of a given input data. Our implementation works with the full EVM bytecode language [37] that additionally includes a volatile memory, omitted because it does not influence the definition of our analysis, and other blockchain-specific operations.

The rules in Fig. 1 define the semantics of the bytecode language for some selected opcodes. A *program state* $\mathcal{S}$ has the form $\langle S, G\rangle$ where the stack S is represented as a list (standard list operations are used to access and modify it) and $S[0]$ corresponds to the top of the stack, and the storage G is a mapping from keys to values such that, given a key k, $G[k]$ contains the value stored at k. The instruction POP removes the topmost element from the stack; DUPx duplicates the element in position x and adds it as the new topmost element; SWAPx swaps the topmost element with the one in position x+1; ADD adds the two topmost values and leaves the result on top of the stack; LOAD retrieves a storage key from the top of the stack, reads the value contained at that key in the storage and leaves it at the top of the stack; STORE retrieves both the storage key and the value to be written from the stack and makes the write operation on the storage; and KECCAK256 computes the *keccak*-256 cryptographic hash of the data on top of the stack. Note that EVM KECCAK256 opcode takes the input data from volatile memory. For simplicity of the formalization, we have omitted the memory and consider in this paper that the input data is taken from the stack.

Executions in Ethereum invoke one of the public functions in the corresponding *smart contract* (i.e., a smart contract is a program that can be executed on the blockchain) and this is how we define the notion of execution trace. Given a public function f and an initial state $\mathcal{S}_0$, we denote by $t \equiv \mathcal{S}_0 \rightarrow^* \mathcal{S}_n$ the trace t corresponding to the execution of f from $\mathcal{S}_0$, where as usual $\rightarrow^*$ denotes the successive application of the rules in Fig. 1 to the opcodes in f.

Example 2.1 (running example). The source code of our running example, shown in Fig. 2, has been taken from a real contract named SalaryDistribution [4]. The shown fragment includes the code of function setSalary, line 5 (L5), which receives the address of an employee and an amount and sets this amount as her salary, saving the information in two data structures in storage: a mapping with the salaries for each employee and an array of actAddr, which saves the addresses of the active employees. When the received amount is zero (and there was a previous salary for the employee), it removes the employee from the array (L9) using function removeAddressFromArray (L13) that includes a loop to search for the address and remove it. The source code is shown only for readability, because our technique is applied at the EVM bytecode level. Below the Solidity, we show the EVM bytecode that is generated to perform a storage access to actAddr[i] which illustrates the use of *keccak* to compute the key for accessing array elements, and will be explained in Sec. 2.3.

To the right of Fig. 2 we show the control flow graph (CFG) for setSalary and include in each block of the CFG the following information: the line number of the Solidity code that the block corresponds to, all storage accesses performed by such line annotated with the key accessed. For instance, L15.2 LOAD *keccak*(1)+*i* // actAddr[i] corresponds to the second storage access made at the source code line 15 in the Solidity code (namely by actAddr[i]), which is compiled to a LOAD that will access the key *keccak*(1)+*i*. Observe the loop formed by nodes L14 and L15 and note that L16 and L17 are outside the loop because of the **break** statement.

2.2 Gas Model and Goal

A *cost model* is a function which assigns a cost to each language instruction. We distinguish two features of cost models: a *static* cost model assigns a cost to an opcode that can be determined statically as it does not depend on the execution context, and a *dynamic* cost model assigns different costs to an opcode depending on the execution context. This context might involve the current state only (e.g., the cost might depend on the current value of an operand) but it might also involve previous states, as it happens in the storage cost model of Ethereum. Regardless of being static or dynamic, a cost model is *symbolic* when it assigns non-constant costs that are defined in terms of state variables. Let us characterize the EVM cost model, as defined in [37], which is static for some opcodes and dynamic for others, and assigns constant and symbolic costs:

i *Static and constant costs*: Most stack opcodes have a static and constant cost (2 or 3 units). The opcodes to operate on the volatile memory (omitted here) also have a static constant cost (3 units).
ii *Static and symbolic costs*: Some arithmetic opcodes have a static cost that is symbolic because it depends on the value they operate on. As an example, the cost of the exponential opcode EXP is defined as $10{+}50 * (1{+}\lfloor log_{256}(v)\rfloor)$ where v is the value corresponding to the exponent (if it is not 0).
iii *Dynamic and constant costs*: As described in Sec. 1, the cost model for the load/store opcodes is dynamic and far more complex. It will be precisely defined in Sec. 2.3.

Cases i and ii are accurately captured by `gastap` and `asparagus`: i is straightforward and ii requires the resource analysis to work with symbolic expressions. In the above EXP example, as v will be stored in the second topmost stack position, the cost model assigns to it the value $10{+}50 * (1{+}\lfloor log_{256}(S[1])\rfloor)$, where $S[1]$ is a variable of the program being analyzed and hence a symbolic resource analysis will be able to handle it properly. Our challenge is to handle case iii, as existing analyzers approximate it inaccurately by making a worst-case static assumption, as already mentioned in Sec. 1.

2.3 Storage Layout and Gas Cost

Storage layout. We consider the storage layout produced by the most-used Solidity compiler, `solc` [6]. In this layout, state variables of value data types and fixed-length data structures (fixed-length arrays and struct variables) are directly stored in the initial keys, while the keys used for nested data structures such as dynamic arrays and mappings are computed using the *keccak*-256 cryptographic hash function. For instance, if an array is to be stored at key

```
1  contract SalaryDistribution {
2    mapping(address => uint256) private salaries;
3    address[] private actAddr;
4
5    function setSalary(address employee,uint256 amount) {
6      if (amount > 0 && salaries[employee] == 0) {
7        actAddr.push(employee);
8      } else if (amount == 0 && salaries[employee] > 0) {
9        removeAddress(employee);
10     }
11     salaries[employee] = amount;
12   }
13   function removeAddress(address employee) internal {
14     for (uint256 i = 0; i<actAddr.length; i++) {
15       if (actAddr[i] == employee) {
16         actAddr[i] = actAddr[actAddr.length - 1];
17         actAddr.pop();
18         break;
19       }
20     }
21   }
22 }
23 // EVM bytecode of access to actAddr[i] at L15
24 // stack[top] contains employee
25 // stack[top-1] contains i
26 PUSH 0x01   // Places key of actAddr.length on stack
27 DUP3        // Duplicates i on top of stack
28 DUP2        // Duplicates actAddr.length key on stack
29 LOAD        // Loads actAddr.length
30 DUP2        // Duplicates i on top of stack
31 LT          // Out-of-bounds check (i<actAddr.length)
32 ISZERO      // Negates condition for checking error
33 PUSH 0x01ce// Address of error block
34 JUMPI       // Jumps to error block if out of bounds
35 JUMPDEST    // Start of block
36 SWAP1       // Swaps i and actAddr.length key on stack
37 KECCAK256   // Computes keccak(1)
38 ADD         // Computes keccak(1)+i
39 LOAD        // Loads actAddr[i] from key keccak(1)+i
```

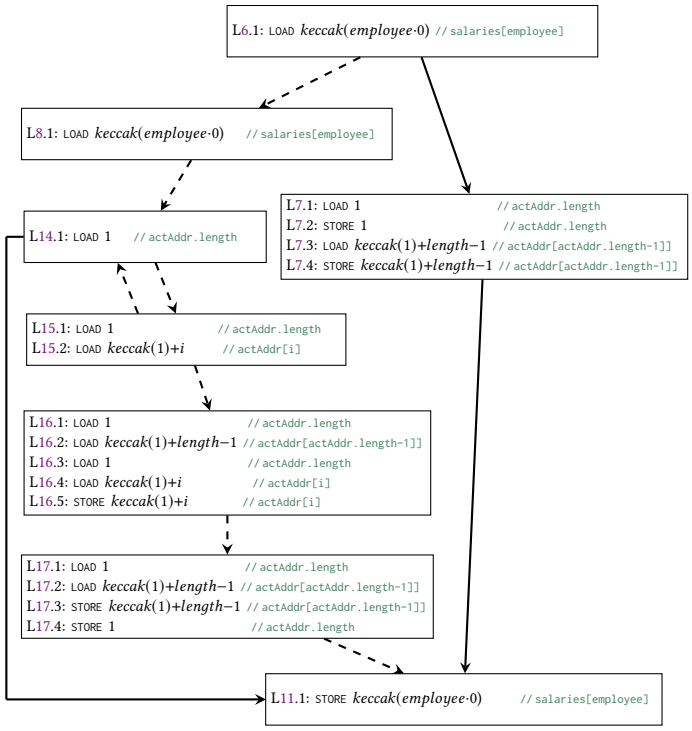

Figure 2: Excerpt of smart contract and a reduced CFG containing storage accesses (annotated with identifier and key)

i, the content of i is just the array length, and the array elements are stored in a sequence of consecutive keys starting at $keccak(i)$. It is assumed that the *keccak* function will never produce any conflict when used with different arguments for computing storage keys, nor the keys computed using this mechanism will overlap with each other, regardless of data structures length. In case a mapping is given a key i, the content of key i remains empty and the key for any index k in that mapping is obtained by $keccak(k{\cdot}i)$, where the operation $k{\cdot}i$ is the concatenation of the values k and i.

Example 2.2. The EVM code shown at Fig. 2 (L26-L39) is generated to compute `actAddr[i]`. As `actAddr` is the second state variable, it is given key 1 (`salaries` is given key 0). According to the storage layout described above, the length of `actAddr` is actually stored at key 1 and its contents are located in consecutive keys starting at $keccak(1)$. The EVM code leads to a partitioning into two blocks, a first block (from L26 to L34) which loads the value of the length and checks if the position accessed is within the bounds of the array. The second block (from L35 to L39) that performs the hash operation $keccak(1)$ and adds i to it in order to compute the key of `actAddr[i]`. The comments to the opcodes in Fig. 2 explain the bytecode step-by-step. In contrast, accesses to map elements, like `salaries[employee]` are performed by applying *keccak* to the concatenation of the key and the identifier of the map, i.e., $keccak(\texttt{employee}{\cdot}0)$.

Gas cost. The next definitions precisely describe the dynamic cost model for storage opcodes which requires to consider the execution trace from the first step. The first definition defines what is known as *cold* storage access, that applies to both load and store.

Definition 2.3 (cold). Given the execution trace $t \equiv \mathcal{S}_0 \rightarrow^* \mathcal{S}_n$. We say that a trace step $s \equiv \langle S_i, G_i\rangle \rightarrow_b \langle S_{i+1}, G_{i+1}\rangle \in t$, with $b \in \{\texttt{LOAD}, \texttt{STORE}\}$, is a *cold access* iff $\forall j{<}i$ s.t. $\langle S_j, G_j\rangle \rightarrow_{b'} \langle S_{j+1}, G_{j+1}\rangle \in t$ and $b' \in \{\texttt{LOAD}, \texttt{STORE}\}$, $S_i[0] \neq S_j[0]$. The cost of s is $\texttt{cost}(s){=}c(cold)$.

The definition above checks that the same key has not been accessed yet along the current execution. The concrete value of $c(cold)$ (and the other coming costs $c(X)$) was given in Sec. 1. After a cold access to a key, the next load accesses to this key are accounted as warm accesses (with a much smaller cost) as stated below.

Definition 2.4 (warm-load). Given the execution trace $t{\equiv}\mathcal{S}_0{\rightarrow^*}\mathcal{S}_n$. We say that a trace step $s \equiv \langle S_i, G_i\rangle \rightarrow_{\texttt{LOAD}} \langle S_{i+1}, G_{i+1}\rangle \in t$ is a *warm-load* iff exists a previous step $\langle S_j, G_j\rangle \rightarrow_{b'} \langle S_{j+1}, G_{j+1}\rangle \in t$, $0{\leq}j{<}i$ s.t. $b' \in \{\texttt{LOAD}, \texttt{STORE}\}$ and $S_i[0]{=}S_j[0]$. The cost is $\texttt{cost}(s){=}c(warm\text{-}load)$.

Example 2.5. Let us focus on map `salaries` first, which is accessed at L6.1, L8.1 and L11.1. identifiers. All these accesses are to the same mapping (which is at key 0) and at the same index (`employee`), and thus, they all access the same storage key ($keccak(\texttt{employee}{\cdot}0)$). It can be observed from the CFG that there exist paths (and hence traces) making three accesses to this key but only the first one has cost $c(cold)$. Let us now observe `actAddr.length`, at key 1, that is the key of many opcodes: L7.1, L7.2, L14.1, L15.1, L16.1, L16.3, L17.1 and L17.4. Following the dashed path (that includes the loop), we can see that the key 1 is accessed $4+2{*}$`actAddr.length` times, but only the

first of such accesses made at the first iteration of the loop of L14, has cost *c(cold)*. The remaining LOAD accesses will have *warm-load* cost. The cost of STORE instructions will be explained later. Finally, array contents are read at L7.3, L15.2, L16.2, L16.4, and L17.2, and modified at L7.4, L16.5 and L17.3. Accesses at L15.2 will be always *cold* because each access corresponds to a different array element (and hence key). If the last element of the array is not removed, L16.2 is *cold*, and otherwise is *warm-load*.

The cost of store accesses is more complex. It distinguishes the three types of accesses iii-v described in Sec. 1. The most expensive store accesses are named set and defined as follows.

Definition 2.6 (set-store). Given the execution trace $t \equiv \mathcal{S}_0 \rightarrow^* \mathcal{S}_n$ with $\mathcal{S}_0 \equiv \langle S_0, G_0 \rangle$. We say that a trace step $s \equiv \langle S_i, G_i \rangle \rightarrow_{\texttt{STORE}} \langle S_{i+1}, G_{i+1} \rangle \in t$ is a *set-store* iff $G_0[key] = 0$, $G_i[key]{=}G_0[key]$, and $G_{i+1}[key]{\neq}G_i[key]$, where $key{=}S_i[0]$. The cost is $\texttt{cost}(s){=}c(\textit{set-store})$.

Basically, a set store is a way to easily capture assignments to the storage for which a high price needs to be paid because they are considered new allocations. Note that, besides the assignments in which the key is first changed from zero to a different value, set-store also applies every time the key is reassigned to zero and then changed back again to non-zero.

The next type of reset has a middle price. It is intended to capture assignments of values to the storage which, as in the previous case, are changing the contents of a key whose current value is the same as the initial one, but this initial value is different form zero.

Definition 2.7 (reset1-store). Given the execution trace $t \equiv \mathcal{S}_0 \rightarrow^* \mathcal{S}_n$ with $\mathcal{S}_0 \equiv \langle S_0, G_0 \rangle$. We say that a trace step $s \equiv \langle S_i, G_i \rangle \rightarrow_{\texttt{STORE}} \langle S_{i+1}, G_{i+1} \rangle \in t$ is a *reset1-store* iff $G_0[key]{\neq}0$, $G_i[key]{=}G_0[key]$, and $G_{i+1}[key]{\neq}G_i[key]$, where $key{=}S_i[0]$. The cost $\texttt{cost}(s){=}c(\textit{reset1-store})$.

The cheapest form of stores corresponds to the case in which the current value is already different from the original. Hence, we must have necessarily paid an expensive set or a middle cost reset1 before, and thus this next change to that key is cheaper.

Definition 2.8 (reset2-store). Consider the execution trace $t{\equiv}\mathcal{S}_0{\rightarrow}^*\mathcal{S}_n$ with $\mathcal{S}_0{\equiv}\langle S_0, G_0 \rangle$. We say that a trace step $s \equiv \langle S_i, G_i \rangle \rightarrow_{\texttt{STORE}} \langle S_{i+1}, G_{i+1} \rangle \in t$ is a *reset2-store* iff $G_{i+1}[key]{=}G_i[key]$ or $G_0[key] \neq G_i[key]$, where $key{=}S_i[0]$. The cost is $\texttt{cost}(s) = c(\textit{reset2-store})$.

Example 2.9. The cost of access L11.1 depends on the value of `amount` and on the value stored at `salaries[employee]` and is dynamically computed as follows: (1) if the initial value was 0 and `amount` $\neq 0$, the cost is *c(set-store)*; (2) if the initial value was $\neq 0$ and `amount` changes the initial value, it is *c(reset1-store)*; and (3) when `amount` does not change the initial value, it is *c(reset2-store)*. Additionally, all accesses to write the contents of `actAddr` (L7.4, L16.5 and L17.3) might have cost *c(set-store)* or *c(reset1-store)* as they all write a value different from the originally stored at its storage key.

Finally, we can define the *storage cost* of an execution trace by assigning zero to all non-storage opcodes and applying the corresponding case of the above definitions when executing load/store.

Definition 2.10 (storage cost of a trace). Consider the execution trace $t \equiv \mathcal{S}_0 \rightarrow^* \mathcal{S}_n$. Let $s_i \equiv \mathcal{S}_i \rightarrow_b \mathcal{S}_{i+1}$ be the i-th execution step of t, the cost of the step $\texttt{cost}(s_i)$ is zero if $b \notin \{\texttt{LOAD}, \texttt{STORE}\}$ and otherwise is the cost of the step according to Defs. 2.3-2.8. The storage cost of the trace is defined as $\texttt{cost}(t) = \sum_{i=0}^{n} \texttt{cost}(s_i)$.

3 Precise Storage Costs Upper Bounds

This section presents our approach to the synthesis of storage cost upper bounds. First, Sec. 3.1 presents how to infer the information that the storage analysis must provide to the next steps. Second, Sec. 3.2 introduces the cost model that we use in order to infer the initial "unsafe" upper bounds using an off-the-shelf resource analyzer. Sec. 3.3 presents our Max-SMT encoding to find the correction to soundly account for *cold* accesses and *set* stores.

3.1 Storage Analysis

As in any static analysis, the first step is the construction of CFG of the program to be analyzed. For the EVM, the construction of a precise CFG has some challenges and has been subject of previous research [17, 18, 29]. In what follows, we simply assume that we have the CFG of the code, obtained by any of these techniques, given as a graph whose vertices are *blocks* containing jump-free sequences of EVM opcodes as, for instance, in the CFG in Ex. 2.1.

Abstract domain. We now define the abstract domain $\mathcal{M}$ to represent *abstract keys* in the analysis. An abstract key is a pair of the form $\langle m, o \rangle$ which enables: (1) representing the keys which are numbers by setting m to ϵ and using only o for the key number, and (2) having a symbolic representation for nested data structures where m is a symbolic representation of the key position returned by *keccak* and o is an offset used to access the particular position inside the data structure. The component m is represented in the domain as $h(k)$, being k the key of the element accessed in the data structure (i.e., the parameter of *keccak*) and h a symbol. The analysis also uses a finite set of symbols $\mathcal{P}$ that include h and identifiers used in the program to compute the keys. As usual, the domain uses $\top$ to represent a value unknown.

Definition 3.1 (storage analysis abstract domain). The storage analysis abstract domain $\mathcal{M}$ is defined as $\mathcal{M} = \{\bot\} \cup \mathcal{M}'$, where $\mathcal{M}' = \{\langle a, n \rangle \mid a \in \{\epsilon, \top\} \cup \{h(m), h(p \odot m) \mid h \in \mathcal{P} \wedge m \in \mathcal{M}' \wedge p \in \mathcal{N}\} \wedge n \in \mathcal{N}\}$, $\mathcal{N} = \mathbb{N} \cup \{\top\} \cup \mathcal{P}$.

$p \odot m$ represents in the abstract domain the concatenation operator $\cdot$ used for accessing maps, where $m \in \mathcal{M}$ is the key of the map in storage and $p \in \mathcal{N}$ is the map element being accessed.

Example 3.2. The abstract representation of the keys in our running example are: $\langle \epsilon, 1 \rangle$ for the key storing the length of the array; $\langle h(\textit{employee} \odot \langle \epsilon, 0 \rangle), 0 \rangle$ for the key of `salaries[employee]`, all accesses to `actAddr[i]` at L15.2, are represented by $\langle h(\langle \epsilon, 1 \rangle), \top \rangle$, which abstracts away the offset as variable `i` changes at each iteration of the loop, i.e., we only know that the array `actAddr` is being accessed but the concrete value of the index is lost.

Fixed-point computation. Given the abstract domain, the definition of the storage analysis is a standard flow-sensitive analysis that traverses the CFG of the program and analyzes each block by applying the *analysis transfer function* to the sequence of opcodes within the block. The transfer function defines how each instruction modifies the *abstract state* which contains the abstract information gathered in the analysis (see e.g. [28]). In our case, as the analysis is developed at the level of the EVM bytecode in Sec. 2.1 and the keys are being computed and propagated by using the stack, the abstract state is made up of the stack contents. The technical definition of

the transfer function is standard and, rather than formalizing it, we just illustrate how the most relevant opcodes for creating and computing the storage keys operate. The first example is the opcode ADD that, when used to compute keys, retrieves the (abstract) keys from the two top-most stack positions of the abstract state (e.g., $\langle m_0, n_0\rangle$ and $\langle m_1, n_1\rangle$) and pushes to the stack $\langle m_0, n_0\rangle \oplus \langle m_1, n_1\rangle$ where $\oplus$ is defined as follows, with $i \in \{0, 1\}$:

$$\langle m_0, n_0\rangle \oplus \langle m_1, n_1\rangle = \begin{cases} \langle m_{1-i}, n_0{+}n_1\rangle & m_i = \epsilon \wedge n_0, n_1 \in \mathbb{N}, \\ \langle m_{1-i}, \top\rangle & m_i = \epsilon \wedge (n_0 \notin \mathbb{N} \vee n_1 \notin \mathbb{N}), \\ \langle \top, \top\rangle & \text{otherwise} \end{cases}$$

As another example, the analysis of KECCAK256 assigns to the stack variable representing the top of the stack the abstract key $\langle h(m), 0\rangle$ where m is the input data used by the *keccak*. Depending on the type of data structure accessed, m can be an abstract key if an array is being accessed, or the result of concatenating an abstract key with the symbol k in the case of an element with key k in a mapping.

Example 3.3. Let us give some intuition of the analysis of the bytecode in Fig. 2 which corresponds to the access to element `i` in array `actAddr` in the for loop at L15. At the beginning of the block starting at L35, the stack contains $\langle \epsilon, i_0\rangle$ (an initial value for variable `i`) and $\langle \epsilon, 1\rangle$ (the key for the array) in the topmost elements of the stack. This initial state for the stack was obtained from the output state of the previous block. Now, the abstract key $\langle \epsilon, 1\rangle$ is taken by the KECCAK256 opcode from the top of the stack and assigns $\langle h(\langle \epsilon, 1\rangle), 0\rangle$ to the topmost element. The next opcode is ADD that performs $\langle h(\langle \epsilon, 1\rangle), 0\rangle \oplus \langle \epsilon, i_0\rangle{=}\langle h(\langle \epsilon, 1\rangle), i_0\rangle$. Accesses to mappings behave differently. For example, accesses to `salaries[employee]`, L6.1, L8.1 and L11.1 apply *keccak* to the concatenation of the symbol *employee* to the map index (0). The application of the transfer function of our analysis reaches the KECCAK256 instruction with a stack containing the abstract values $\langle \epsilon, \texttt{employee}\rangle$ for the mapping key and $\langle \epsilon, 0\rangle$ for the mapping identifier, and returns $\langle h(employee \odot \langle \epsilon, 0\rangle), 0\rangle$ as abstract key for the map element.

Another standard feature of a static analysis is to be *context-sensitive*, i.e., as a block in the CFG might be reached from several blocks, rather than keeping a single abstract value (for each represented variable in the state), we keep a *set* of possible keys which represent such disjunction of paths. In order to guarantee termination in the presence of loops, after a fixed number of iterations, the analysis takes the join $\sqcup$ of the reached (abstract) states. As usual, $a \sqcup \bot = \bot \sqcup a = a$, and other cases boil down to the join of each component of the pairs separately: given $n_1, n_2 \in \mathcal{N}$, $n_1 \sqcup n_2{=}n_1$ if $n_1{=}n_2$ and $\top$, otherwise. Given $m_1, m_2 \in \{\epsilon, \top\} \cup \{h(m), h(p \odot m) \mid h \in \mathcal{P} \wedge m \in \mathcal{M}' \wedge p \in \mathcal{N}\}$, then

$$m_1 \sqcup m_2 = \begin{cases} \epsilon & m_1{=}\epsilon \wedge m_2{=}\epsilon \\ h(m_1' \sqcup m_2') & m_1{=}h(m_1') \wedge m_2{=}h(m_2') \\ h((p_1 \sqcup p_2) \odot (m_1' \sqcup m_2')) & m_1{=}h(p_1 \odot m_1') \wedge m_2{=}h(p_2 \odot m_2') \\ \top & \text{otherwise} \end{cases}$$

Example 3.4. The abstract value in the first iteration of the analysis of block L15 for the access L15.2 is $\{\langle h(\langle \epsilon, 1\rangle), 0\rangle\}$. In the second iteration it is joined to $\{\langle h(\langle \epsilon, 1\rangle), 0\rangle, \langle h(\langle \epsilon, 1\rangle), 1\rangle\}$, and so on for the next iterations. After reaching a fixed number of iterations, the states are joined into $\{\langle h(\langle \epsilon, 1\rangle), \top\rangle\}$ and the analysis finishes.

Analysis output. Once the analysis finishes, it provides us with a function ϕ that, at each program point that contains a load or store opcode, yields a sound over-approximation of the (abstract) storage keys that might be accessed at this instruction:

Definition 3.5 (storage analysis function ϕ). Let PP be the set of program points. The *storage analysis function* $\phi : PP \mapsto \wp(\mathcal{M})$ is a mapping that, for each program point pp that contains {LOAD, STORE}, $\phi(pp)$ returns the set of abstract keys that might be accessed at pp.

Example 3.6. The storage analysis output for selected points:

Accesses	ϕ	Source
L14.1, L15.1, L16.[1,3], L17.[1,4]	$\langle \epsilon, 1\rangle$	`actAddr.length`
L15.2, L16.4, L16.5	$\langle h(\langle \epsilon, 1\rangle), \top\rangle$	`actAddr[i]`
L6.1, L8.1, L11.1	$\langle h(\texttt{employee} \odot \langle \epsilon, 0\rangle), 0\rangle$	`salaries[employee]`

Theorem 3.7 (Soundness of Storage analysis). *Given an execution trace t, the keys used in storage access instructions* LOAD *and* STORE *are soundly represented by the analysis output function ϕ.*

proof sketch. The analysis produces, for each program point pp, a finite set $\phi(pp) \in \wp_f(\mathcal{M})$ where $\wp_f(\mathcal{M})$ is the set of all finite subsets of $\mathcal{M}$, and we have to prove that, for each step that executes a storage access instruction, there exists some element $S \in \phi(pp)$ of the abstract domain $\mathcal{M}$ that represents the concrete key accessed in that step. We define for $\wp_f(\mathcal{M})$ an abstract state that maps stack positions to elements in $\wp_f(\mathcal{M})$, and a transfer function that computes the changes to the abstract state when executing stack and storage handling opcodes, and keccak computations. The proof is then by induction on the length of the traces considering those opcodes, proving that the theorem holds for each case of the transfer function. Since $\mathcal{M}$ is an infinite set and $\wp_f(\mathcal{M})$ can contain non-finite ascending chains, we need to define a *widening* operator that satisfies the *ascending chain condition* to guarantee termination [14]. We use the operator $\nabla A = \{\sqcup A\}$, where $A \in \wp_f(\mathcal{M})$. □

3.2 Using an Off-the-Shelf Resource Analysis

Overview. Let us first give an overview of the reasoning in our approach. There are two unsound assumptions: As regards loads, the idea is that all load accesses not being $\top$ will be accounted as warm accesses within the cost model. Hence, we might be missing the amount of *c(cold)−c(warm-load)* if this access is in fact a cold access. Therefore, if there are CL loads with a cold access to a position, then we have to add as *load-correction* C_{cold}=CL∗(*c(cold)*−*c(warm-load)*).

For the store opcodes, following the EVM gas model for the store in Sec. 2.2, we have the following result.

Lemma 3.8. *Let $store(p, v_1), store(p, v_2), \ldots, store(p, v_n)$ be a sequence of store operations to the same key p where all accesses are warm. Then, the worst-case gas cost is*

$$\begin{cases} (c(\text{set-store}){+}c(\text{reset2-store}))*n/2 & \text{if } n \text{ is even} \\ (c(\text{set-store}){+}c(\text{reset2-store}))*(n-1)/2 + c(\text{set-store}) & \text{otherwise.} \end{cases}$$

Proof. First, we note that the worst-case gas cost for two consecutive stores in the same position (assuming the first one is warm) is *c(set-store)+c(reset2-store)*, since only one of both can change the current value from the initial one (v_0) to a value different from the initial one (which is required to pay the cost *c(set-store)*

when $v_0 = 0$). Now, we prove the result by induction on the size of the sequence. If $n = 0$ it holds trivially. Otherwise, if n is even, by the induction hypothesis, we have that the worst-case cost for the sequence $store(p, v_1)$, $store(p, v_2)$, ..., $store(p, v_{n-2})$ is (c(set-store)+c(reset2-store))$*(n-2)/2$ and the worst-case for the last two stores is c(set-store)+c(reset2-store), and hence it holds. Finally, if n is odd (and hence $n-1$ is even), by the induction hypothesis, the worst-case cost for $store(p, v_1)$, $store(p, v_2)$, ..., $store(p, v_{n-1})$ is (c(set-store)+c(reset2-store))$*(n-1)/2$, which implies the result, since the worst-case cost for the last store is c(set-store). □

Moreover, assuming that we do not have information about the values that are stored, this upper bound is tied since there exists a sequence with the given cost: the one where $v_0 = 0$ and, then, we alternate an store operation with a value different from v_0 with a store operation with v_0 again. Note also that if the initial value is not 0 then the result of Lemma 3.8 is the same but replacing the c(set-store) cost by c(reset1-store), which means we could easily produce a cost that is parametric on the initial values of the storage.

Using Lemma 3.8, the idea is to assign to every store operation a cost of $\frac{c(\text{set-store})+c(\text{reset2-store})}{2}$ and, since this is unsound if we have an odd number of stores to the same key (since we only add $\frac{c(\text{set-store})+c(\text{reset2-store})}{2}$ instead of c(set-store) for the last store to that key), we have to correct this approximation by computing separately the worst case OS in number of odd stores at different keys and add $C_{set1} = OS * (\frac{c(\text{set-store})+c(\text{reset2-store})}{2} - c(\text{reset2-store}))$ to the final cost. Additionally, if there are CS stores with a cold access to a key (i.e., when this is the first access to such position key) then we have to add $C_{set2} = CS * c(\text{cold})$ to the cost since we have not added any cost in this respect. Therefore, C_{set}=C_{set1}+C_{set2} will be needed as *store-correction*. Altogether we have to obtain values for CL, CS and OS, for a trace that maximize the value of $CL * (c(\text{cold}) - c(\text{warm-load})) + CS * c(\text{cold}) + OS * (\frac{c(\text{set-store})+c(\text{reset2-store})}{2} - c(\text{reset2-store}))$ and add it to the result obtained from the off-the-shell resource analyser in order to finally obtain a sound storage cost bound.

Using an off-the-shelf resource analyzer. According to the overview given above, we can use an off-the-shelf resource analyze with the following cost model that assigns described unsound costs to load and stores (with no $\top$) and later correct them.

Definition 3.9 (storage cost model, UB_u, and UB_v). Let ϕ be the storage analysis function in Def. 3.5, I be a storage access opcode at program point pp. The cost model C_s assigns the cost:

$$C_s(I) = \begin{cases} c(\text{warm-load}) & \neg hasTop(\phi(pp)) \wedge I \equiv \texttt{LOAD}, \\ c(\text{cold}) & hasTop(\phi(pp)) \wedge I \equiv \texttt{LOAD}, \\ \frac{c(\text{set-store})+c(\text{reset2-store})}{2} & \neg hasTop(\phi(pp)) \wedge I \equiv \texttt{STORE}, \\ c(\text{set-store}) + c(\text{cold}) & hasTop(\phi(pp)) \wedge I \equiv \texttt{STORE}, \\ 0 & \text{otherwise} \end{cases}$$

where $hasTop(A)$ returns true if at least one element in A contains $\top$. Given a function f, the *(unsound) storage cost* upper bound of f, denoted $UB_u(f)$ is the result of applying a *sound resource analyzer* using the cost model C_s. The analyzer also computes for each block in the CFG of f an upper bound on the number of *visits*, UB_v, to it.

Multiple resource analyzers, that can be provided with any *cumulative* cost model (i.e., any cost model that does not accumulate negative costs), have been defined for imperative programming languages by relying on different formalisms (e.g., [7, 11, 19, 19, 21, 25, 31, 34, 34]). Both `gastap` and `asparagus` integrate resource analyzers that can be used out of the box –by setting their cost model C_s– and produce UB_u, and bound the visits UB_v for each block. Hence, more technical details for this stage are not needed.

Example 3.10. The application of the cost model to the storage accesses of the running example we get:

Accesses	Opcode	C_s
L6.1, L8.1, L14.1, L15.1, L16.[1,3], L17.1	LOAD	c(warm-load)
L15.2, L16.[2,4], L17.2	LOAD	c(cold)
L11.1, L17.4	STORE	$\frac{c(\text{set-store})+c(\text{reset2-store})}{2}$
L16.5, L17.3	STORE	c(set-store) + c(cold)

Finally, the UB_u cost obtained by a sound resource analyzer, such as `gastap` and `asparagus`, using the *storage cost model* would be:

$$UB_u = (5+2*l)*c(warm-load) + (l+3)*c(\text{cold})+ 2*\frac{c(\text{set-store})+c(\text{reset2-store})}{2} + 2*(c(set-store) + c(\text{cold}))$$

The computation of the UB_u also allows us to extract that the visits for the blocks within the loop at L14 are UB_v = `actAddr.length`.

3.3 Max-SMT Encoding for Corrections

This section presents our Max-SMT encoding to the problem of finding CL, CS and OS for the load and store corrections C_{cold} and C_{set} introduced in Sec. 3.2. In particular, we express the problem as a Max-SMT problem using hard and weighted soft clauses, such that the optimal solution to the problem will give us the needed values. The difficulty of the encoding relies on how to handle loops since we have to characterize all possible sequences (running the loops) that can maximize the sum of the costs of having cold access for loads and stores and keys with an odd number of store operations. Finally, note that the problem can be encoded using weight because all costs are given by three different constants (see below for details).

DAG representation. In order to encode the problem, we take the CFG of the program and generate from it, and from the bounds on the visits UB_v computed by the off-the-shelf resource analyser in Def. 3.9, a DAG representation with loop annotations. In this DAG, we only keep the storage accesses A (both loads and stores) such that $\neg hasTop(A)$ (i.e., we use the outcome of the storage analysis as well). The DAG soundly approximates the non-top storage accesses in any trace of the execution. In order to ease the encoding, the DAG is represented as a (tree) expression containing visits (**r**), branching (**b**) and access (**a**) operations following this grammar:

$$\begin{aligned} \langle dag \rangle &::= \langle seq\_instruction \rangle \\ \langle seq\_instruction \rangle &::= (\langle repetition \rangle | \langle branch \rangle | \langle access \rangle)^* \\ \langle repetition \rangle &::= \mathbf{r}(\langle visits \rangle, \langle seq\_instruction \rangle) \\ \langle branch \rangle &::= \mathbf{b}(\langle seq\_instruction \rangle, \langle seq\_instruction \rangle) \\ \langle access \rangle &::= \mathbf{a}(\text{“}l\text{”} | \text{“}s\text{”}, (\langle key \rangle)^+) \end{aligned}$$

where *visits* is an arithmetic expression (UB_v), *key* is any of the non-top keys detected in the analysis, and "l" and "s" indicate resp. whether the access is a load or a store. Note that access instructions with several keys are expressing that we can choose one of the keys every time we execute the instruction. Therefore, this behaviour can be also expressed using access instructions with a unique key (and branching). For this reason, to ease the presentation we will assume that there is a single key in all access instructions.

Example 3.11. The following expression captures the DAG of the code in Fig. 2 (to the right we show the corresponding accesses):

(1)	$\mathbf{a}(l, \langle h(employee \odot \langle \epsilon, 0\rangle), 0\rangle)$	L6.1
(2)	$\mathbf{b}($ $\mathbf{a}(l, \langle h(employee \odot \langle \epsilon, 0\rangle), 0\rangle)$	L8.1
(3)	$\quad \mathbf{r}(\texttt{addr.length}, \mathbf{a}(l, \langle \epsilon, 1\rangle)\ \mathbf{a}(l, \langle \epsilon, 1\rangle))$	L14.1, L15.1
(4)	$\quad \mathbf{a}(l, \langle \epsilon, 1\rangle)\ \mathbf{a}(l, \langle \epsilon, 1\rangle)\ \mathbf{a}(l, \langle \epsilon, 1\rangle)\ \mathbf{a}(l, \langle \epsilon, 1\rangle)$,	L16.[1,3], L17.[1,4]
(5)	$\quad \mathbf{a}(l, \langle \epsilon, 1\rangle)\ \mathbf{a}(s, \langle \epsilon, 1\rangle))$	L7.1, L7.2),
(6)	$\mathbf{a}(s, \langle h(employee \odot \langle \epsilon, 0\rangle), 0\rangle)$	L11.1

The DAG includes the following information: row (1) includes the accesses performed before the loop; row (2) shows the *if-else* with both branches, the first branch including rows (2), (3) and (4), and the second branch including row (5). Note that the loop, row (3), includes its number of iterations and its accesses. In addition, at row (6) we have the last access to write the map.

Max-SMT encoding. Let K be the set containing all different keys, let I be the set of all instructions (**r**,**b** or **a**) in the DAG. Let L_k and S_k be resp. the set of load (with "l") and store (with "s") access instructions at key k and $A_k = L_k \cup S_k$. In order to find the worst case for CL, CS and OS, we need to find the number of storage keys with an odd number of store operations and the keys that have been accessed and whether they were first accessed with a load operation or a store operation (which is more expensive) in feasible traces in the DAG. To this end we will need: (i) Boolean variables stating whether an instruction has been executed or not in the trace; (ii) for some instructions, integer variables stating the number of times the instruction has been executed in the trace and (iii) for some instructions, integer variables indicating the first time they were executed in the trace. The aim is to find an assignment compatible with the given DAG that maximizes $CL * (c(cold) - c(warm\text{-}load)) + CS * c(cold) + OS * (\frac{c(set\text{-}store) - c(reset2\text{-}store)}{2})$, which will be achieved by adding soft constraints with weights in $\{c(cold) - c(warm\text{-}load), c(cold), \frac{c(set\text{-}store)+c(reset2\text{-}store)}{2}\}$ to force the Max-SMT solver to find a solution that maximizes the sum of the weights of the satisfied soft constraints.

If there are no loops, we do not need to express the first time an instruction is executed since the DAG syntactically expresses which instruction goes first. But, in the presence of loops, this is not the case anymore. For this reason, we define the level of an instruction as the number of *repetition* instructions we need to traverse to reach it. In any trace, all instructions at level 0 can occur at most once and, if so, they are executed in a single step k (that is consistent with the sequence of instructions of the DAG). For instructions from level 1 on, this is not the case anymore. Since we are interested in expressing the first time an instruction is executed, and we want to express it in a way that we can constraint to only feasible cases in the given DAG, we use a quite sophisticated description of the moment an instruction is executed. This description is based on the levels. Assume the level of an instruction i is j, say 2. This means that our instruction is below 2 repetition instructions. Then, the first execution of our instruction will happen as follows: (i) we execute the first repetition instruction at the step f_{i0} (at level 0); (ii) in the r_{i1} iteration of the first repetition instruction, we execute the second repetition instruction at the step f_{i1} (of the round r_{i1} of the first iteration at level 1) and (iii) in the r_{i2} iteration of the second repetition, we execute the instruction i at the step f_{i2} (of the round r_{i2} of the first iteration at level 2); and so on if we have more nested repetitions. In general, in order to describe the first execution of an instruction i that occurs at level j, we need j values $f_{i0}, \dots f_{ij}$ and $j-1$ values $r_{i1}, \dots r_{ij}$. Moreover, the first occurrence of i at level j happens before the first occurrence of i' at level j' if $\langle f_{i0}, r_{i1}, f_{i1}, \dots r_{ij}, f_{ij}\rangle <_{lex} \langle f_{i'0}, r_{i'1}, f_{i'1}, \dots r_{i'j'}, f_{i'j'}\rangle$, where $<_{lex}$ is the lexicographic comparison on sequences of (non-negative) integers. With this representation of the first execution of instructions, we can capture all possible traces (and hence ensure soundness) but also restrict many unfeasible situations adding constraints (and hence keep precision), even when considering any level of nested iterations. For every instruction i at (known) level j we denote by $First(i)$ the sequence $\langle f_{i0}, r_{i1}, f_{i1}, \dots r_{ij}, f_{ij}\rangle$ and by $First_p(i)$ its prefix of size $p \leq 2 \cdot j + 1$.

The encoding adds variables representing properties on the instructions and on the keys. In particular, for each instruction i in the DAG (which can be a *repetition*, a *branch* or an *access*) we create:

(1) An integer variable t_i, which indicates the number of times the instruction i has been executed. Note that for instructions at level 0, the value of t_i can only be 0 or 1. Moreover if i is a *branch* instruction then we introduce t_i^0 and t_i^0 to indicate resp. the number of times we execute the first and the second branch.
(2) If level of i is j, then we have j integer variables $f_{i0} \dots f_{ij}$ and $j-1$ integer variables $r_{i1} \dots r_{ij}$, which indicate the first execution of the instruction i (if the instruction was executed, i.e. $t_i > 0$).
(3) We also add integer variables for the variables occurring in the *visits* expressions of the repetition instruction, so we assume the expression can be used in the encoding.

We add hard constraints on these variables to capture only feasible traces in the given DAG. The following constraint states the conditions on the first execution of all instructions in a sequence at level j: $Seq_j(i_0, \dots, i_m) = \bigwedge_{p=1}^{j} \bigwedge_{q=0}^{m-1} r_{i_q p} = r_{i_{q+1} p} \wedge \bigwedge_{p=0}^{j-1} \bigwedge_{q=0}^{m-1} f_{i_q p} = f_{i_{q+1} p} \wedge \bigwedge_{q=0}^{m-1} \bigwedge_{i' \in under(i_q)} (First_{2\cdot j}(i') \neq First_{2\cdot j}(i_{q+1}) \vee f_{i'j} < f_{i_{q+1} j})$, where $under(i)$ includes all instructions that occur below (including itself) the instruction i.

We add the following constraint for the instructions:

(1) for all instructions $i_0 \dots i_m$ in the initial sequence of DAG we add $\bigwedge_{q=0}^{m} t_{i_q} = 1$, $f_{i_0 0} = 1$ (if $m > 0$) and $Seq_0(i_0, \dots, i_m)$.

(2) for every repetition instruction i at level j with *visits* expression e and sequence of instructions $i_0 \dots i_m$ we add: $(e > 0 \wedge t_i > 0) \vee m = 0 \vee ((\bigwedge_{p=0}^{m} t_{i_p} = e * t_i) \wedge Seq_{j+1}(i_0, \dots, i_m) \wedge (\bigwedge_{p=1}^{j} r_{i_0 p} = r_{ip}) \wedge (\bigwedge_{p=0}^{j-1} f_{i_0 p} = f_{ip}) \wedge (\bigwedge_{q=0}^{m} r_{i_q j+1} = 1) \wedge f_{i_0 j+1} = 1)$.

In general $e * t_i$ can be non-linear. When this happens we use context information (for instance the values that t_i can take) to remove the non-linearity, and when this is not possible we introduce new fresh variables to abstract non-linear monomials, but in this case we just lose precision (not soundness) as all feasible traces are still solutions to the encoding. We also add the constraint $\bigwedge_{i' \in under(i)} r_{i'j+1} \leq e$, since the round at level $j+1$ below repetition instruction i cannot be greater than the total times we execute i (given by e).

(3) for every branch instruction i at level j with instructions $i_0^0 \dots i_{m_0}^0$ and $i_0^1 \dots i_{m_1}^1$ in the first and second branch resp., we add constraints $t_i = t_i^0 + t_i^1$, $t_i^0 = t_{i_0}$, $t_i^1 = t_{i_1}$, $Seq_j(i_0^0, \dots, i_{m_0}^0)$, $Seq_j(i_0^1, \dots, i_{m_1}^1)$,$(t_i = 0 \vee (First(i) = First(i_0^0) \wedge First(i_{m_0}^0) \leq$

$First(i_0^1)) \vee (First(i) = First(i_0^1) \wedge First(i_{m_1}^1) \leq First(i_0^0))$. Additionally, we add variables for each key k in K and constraints describing its behaviour.

(1) A Boolean variable s_k (if there $S_k \neq \emptyset$) meaning that the first access to k is a store and add a constraint $s_k = \vee_{i \in S_k} \wedge_{i' \in L_k} first(i) <_{lex} first(i')$.
(2) A Boolean variable a_k meaning that k has been accessed and add a constraint $a_k = \vee_{i \in A_k} t_i > 0$.
(3) A Boolean variable o_k (if there $S_k \neq \emptyset$) meaning that there is an odd number of stores to k and add a constraint $o_k = (t_i = 2 * d_k + 1)$, where d_k is an auxiliary integer variable that we also add for each such k.

Finally, for every k in K we add soft weighted clauses to get a solution that maximizes the cost of the execution trace:

(1) $s_k, weight(c(warm\text{-}load))$
(2) $a_k, weight(c(cold) - c(warm\text{-}load))$
(3) $o_k, weight(\frac{c(set\text{-}store) - c(reset2\text{-}store)}{2})$

Then, we have the following result.

Lemma 3.12. *Let σ be the optimal solution to the Max-SMT problem described above. Taking CS as the number of variables s_k that are set to true in σ, CL the number of variables a_k that are set to true σ minus CS and OS the number of variables o_k that are set to true in σ, provides the worst-case for the correction $CL * (c(cold) - c(warm\text{-}load)) + CS * c(cold) + OS * (\frac{c(set\text{-}store) - c(reset2\text{-}store)}{2})$.*

Proof. (Sketch) We first show that any feasible execution trace in the given DAG has a solution that satisfies the constraints, which means to first assign values to all integer variables we have in the encoding for every instruction i at level j: (1) t_i (for the times it has been executed) and (2) $f_{i0} \ldots f_{ij}$ and $r_{i1} \ldots r_{ij}$ (which indicate the first execution of the instruction i, if executed). It is not difficult (but tedious) to show that such solution can be extracted from the given trace and that it fulfills the introduced constraints exist. Then, from this assigned values to the integer variables we can propagate the values for all the Boolean variables: (1) s_k (that means that the first access to k is a store), (2) a_k (meaning that k has been accessed) and (3) o_k (meaning that there is an odd number of stores to k). Therefore every feasible trace has a solution with a corresponding cost for the soft constraints that exactly coincides with the gas cost for the correction of the sequence. Note that since we assign a cost *c(warm-load)* to s_k, we have that the correction for cold loads is *c(cold) − c(warm-load)*, while for cold stores is *c(cold)*. Finally, since a Max-SMT solver looks for the solution that maximizes the cost of the satisfied soft constraints, the optimal solution will provide us with the worst-case for the correction formula. □

Note that this lemma requires the optimality of the solution to the Max-SMT problem, since otherwise we cannot guarantee that we have obtained a sound worst-case cost.

Example 3.13. Considering the DAG defined in Ex.3.11, the optimal solution returned by our encoding corresponds to the following values: $CL = 2$, $CS = 0$ and $OS = 2$. Note that it detects that we have, at most, two *cold* load accesses (`actAddr.length` and `salaries[employee]`) and two keys (the same as before) with an odd number of store accesses. As a result, the load-correction $C_{cold} = 2 \cdot (c(cold) - c(warm\text{-}load))$ and the set-correction $C_{set} = 2 \cdot (\frac{c(set\text{-}store) - c(reset2\text{-}store)}{2})$ (as $C_{set1} = 0$, since the keys are always accessed first with a load). Adding this correction to the UB_u of Ex. 3.10, we have the following sound result:

$UB_S = (3+2*l)*c(warm{-}load) + (l+7)*c(cold) + 4*c(set\text{-}store)$

The cost model of previous approaches, `gastap` and `asparagus` [7, 12], considers the worst case cost for all storage operations, that is: *c(cold)* for `LOAD` operations and *c(set-store) +c(cold)* for `LOAD` operations and yields the following storage $\text{UB}_0 = (12 + 3*l) * c(cold) + 4*c(set\text{-}store)$. It can be seen that our storage UB_S replaces $(5 + 2*l)*c(cold)$ gas units by $(3 + 2*l)*(c(warm\text{-}load))$, which is a very significant gain for a single and relatively simple method (since currently $c(cold) = 2100$ and $c(warm\text{-}load) = 100$). However, in this example there is no gain due to the `STORE` accesses as there is only one non-top access. As a simple example consider the code `for(i=0;i<n;i++){if (*) key=*;}` in Sec. 1 and assume that key is 0. In this case, the DAG is simply $\mathbf{r}(\mathtt{n}, \mathbf{a}(s, \langle \epsilon, 0 \rangle))$. Our unsound UB is $UB_u = n * \frac{c(set\text{-}store) + c(reset2\text{-}store)}{2}$ and the Max-SMT solver returns CL = 0, CS=1 and OS=1, and hence the final cost bound is $n * \frac{c(set\text{-}store) + c(reset2\text{-}store)}{2} + \frac{c(set\text{-}store) - c(reset2\text{-}store)}{2} + c(cold)$. But, suppose that we have instead $\mathbf{r}(\mathtt{100}, \mathbf{a}(s, \langle \epsilon, 0 \rangle))$ or $\mathbf{r}(\mathtt{n}, \mathbf{a}(s, \langle \epsilon, 0 \rangle)\ \mathbf{a}(s, \langle \epsilon, 0 \rangle))$ then the Max-SMT solver returns CL = 0, CS=1 and OS=0, since the number of stores in key 0 is always even, and the resulting UB is $n \cdot \frac{c(set\text{-}store) + c(reset2\text{-}store)}{2} + c(cold)$.

Theorem 3.14 (soundness). *Consider a function f, the synthesized storage bound $UB = UB_u + C_{cold} + C_{set}$ for f, where UB_u is defined in Def. 3.9 and C_{cold} and C_{set} in Sec. 3.2 is a sound upper bound, i.e., $\mathtt{cost}(t) \leq UB$ for any possible execution trace t of f.*

The proof is based on the reasoning about the worst case sequences given in the overview in Sec. 3.2.

4 Experimental Evaluation

The techniques proposed in the paper have been implemented in Python, are open-source (available at https://github.com/costa-group/EthIR), and have been integrated into the `gastap` [7] and `asparagus` [12] systems. Such gas-analyzers take as input a smart contract (either in Solidity or in EVM form) and automatically infer gas upper-bounds for its public functions. While they rely on the same tools for the CFG generation and use the same intermediate representation of the EVM bytecode, they differ in the techniques used to obtain the gas cost bounds: `gastap` generates and solves cost recurrence relations, while `asparagus` relies on OptiMathSAT [30] compute via polyhedral and algebraic geometry. The integration of our approach into these systems has required: the implementation of the storage analysis in Sec. 3.1, the modification of the original cost model to use Sec. 2.3 for the storage opcodes while the remaining opcodes use the existing gas model, and the implementation of the Max-SMT problem described in Sec. 3.3, using Z3 SMT-solver [16]. We call `gastap`$^+$ (`asparagus`$^+$) to the version of `gastap` (`asparagus`) that includes our extensions.

Experimental setup. Our experimental results aim at evaluating the accuracy gains of `gastap`$^+$ and `asparagus`$^+$ over the basic systems and over the gas estimator of an up-to-date version of `solc` (0.8.24), simply named `solc` in what follows. We will compare `gastap`$^+$ vs. `solc`, `gastap`$^+$ vs. `gastap` and `asparagus`$^+$ vs. `asparagus`. The reason why we do not compare `asparagus`$^+$ vs.

solc is that asparagus (and hence its improvement asparagus$^+$) only accepts smart contracts written in versions of Solidity up to 0.4.26. Therefore, the comparison with solc would not be sound, as the gas model implemented by versions 0.4 of solc is outdated and has a different storage gas cost (namely 200 for SLOAD and 5,000 or 20,000 for SSTORE). The comparison of gastap$^+$ vs. asparagus$^+$ is also not carried out because it would evaluate two different approaches to compute gas cost upper-bounds rather than the precision of our extension. For the evaluation, we have used two different benchmark sets: (1) For gastap, we pulled from Etherscan [5] the Ethereum contracts bound to the last 5,000 open-source verified addresses whose source code was available on Feb 26, 2024. These addresses lead to 134,074 public functions from 8,130 smart contracts, as each address may correspond to several Solidity files that in turn may contain several contracts in them. Finally, we have selected only the 113,575 (84.71%) public functions that contain storage operations. (2) For asparagus, as the previous set cannot be used because of its limitation to use Solidity up to 0.4.26, we have used the original set used to evaluate asparagus [23] which contains 9,644 Solidity files with 29,258 contracts. These contracts have 325,381 public functions of which 287,387 (88.33%) have storage operations and are used for our evaluation. Experiments have been performed on an AMD Ryzen Threadripper PRO 3995WX 64-cores and 512 GB of memory, running Debian 5.10.70. We have set a global timeout of 120s per function.

Evaluation. Out of 113,575 (resp. 287,387) functions, gastap$^+$ (resp. asparagus$^+$) has found an upper bound gas cost for 75,956 (resp. 108,715). Failures are mostly related to reaching the timeout or failures of the original systems (unrelated to our extensions). Focusing on the solved cases in gastap$^+$ (resp. asparagus$^+$), 88.65% (resp. 97.87%) are constant bounds and 11.35% (resp. 2.13%) are parametric functions. Fig. 3 shows the gains obtained thanks to our extension for the constant bounds. Parametric bounds are not displayed because: solc cannot handle them and a graphical comparison would require drawing the functions one by one. For plots in the first row, the y-axis shows the number of functions and the x-axis the percentage of gas gains obtained when comparing gastap$^+$ vs. solc (i), gastap$^+$ vs. gastap (ii) and, asparagus$^+$ vs. asparagus (iii). The scatter plots (second row) show for each analyzed function (x-axis), the amount of gas gained (y-axis) by gastap$^+$ *wrt.* solc (iv), by gastap$^+$ *wrt.* gastap (v) and by asparagus$^+$ *wrt.* asparagus (vi).

Let us compare our gains *wrt.* solc. Plot (i) includes only the 10,010 functions for which gastap$^+$ improves the constant result of solc, with a global average improvement of 16.97%, while (iv) shows the 42,981 functions that solc solves. We can observe in plot (i) that gains in the range 5-10% are achieved for more than 2,700 functions. Importantly, the largest group corresponds to functions with a gas improvement between 20-25%. By inspecting many of them, we have noticed that often correspond to implementations of an ERC20 token, which is widely used, and that make multiple storage accessed to the same key. While the original system makes a worst case cost assumption of *c(cold)* for the load and *c(set-store)* + *c(cold)* for store, we accurately treat them as explained in the paper. In plot (iv), we include also the functions (>30,000) for which gastap$^+$ and solc infer the same constant bound. An inspection of the code of such functions has revealed that >90% of the cases are functions that make a unique load in the storage, hence, there is no room for improvement. The most frequent improvements are under 10,000 gas units, in total 9,613 functions, which can be classified in three groups: savings of 2,000 units of gas (1,361 functions), 4,000 units of gas (1,362 functions), and 6,000 units of gas (5,058 functions). These improvements correspond, resp., to cases for which gastap$^+$ identifies 1, 2 or 3 warm accesses, while solc classifies them as cold. Importantly, we can also observe many functions with dense groups that correspond to gains of 8,000 units of gas, 26,000 units of gas, 28,000 units of gas or 32,000 units of gas. All these gains are multiple of combinations among the gains obtained when considering *c(warm-load)* instead of *c(cold)* or $\frac{c(\text{set-store})+c(\text{reset2-store})}{2}$ instead of *c(set-store)* + *c(cold)*, e.g., gains of 8,000 units correspond to 4 warm accesses classified as cold by solc, and improvements around 26,000 units to two store accesses for which gastap$^+$ assigns $\frac{c(\text{set-store})+c(\text{reset2-store})}{2}$, while solc assigns *c(set-store)* + *c(cold)* and one load access that costs *c(warm-load)*.

Finally, let us compare jointly gastap$^+$ and gastap (plots (ii) and (v)) and asparagus$^+$ and asparagus (plots (iii) and (vi)) as they feature a very similar behaviour. In these cases, we have 22,056 functions in (ii), 68,514 in (v), 62,820 in (iii) and 98,810 in (vi), which are considerably higher than when comparing to solc. gastap and gastap$^+$ (resp. asparagus$^+$ and asparagus) infer the same constant upper-bound for 46,765 (resp. 36,065) functions, for the same reasons as explained above for solc. For the remaining functions, gastap$^+$ (resp. asparagus$^+$) reduces the inferred gas upper-bound in 13.84% (resp. 13.82%) on average. As shown in (ii) and (iii), the most common gains are between 5-10% of the total gas consumed. As expected, (v) and (vi) contain the groups that were described for (iv) because the original systems implement the same cost model as solc. In (v) and (vi), groups are denser (as there are more functions) and the plots contain further groups than (iv) (groups around 16,000 or 34,000 units). Notice that the main gains are concentrated on the same values, those that correspond to multiple of combinations between the gains obtained from considering load accesses as *c(warm-load)* and the new cost for the store operations. All in all, we argue that the plots of Fig. 3 confirm that the proposed approach to storage bounds synthesis greatly improves the accuracy of the inferred bounds by the different systems used in the evaluation. Let us mention that the time overhead of our extensions is negligible, namely we add an additional 2%-3% time for both gastap$^+$ and asparagus$^+$ *wrt.* the original version of the tools.

Comparing the UB's with real transactions. Finally, we evaluate the precision of the UB'S inferred by comparing them with the actual cost of real transactions of the analyzed addresses (as they are stored in the blockchain). We have downloaded (June 15, 2024) from Etherscan [5] the real transactions of the 5,000 open-source verified addresses used before. As some addresses may have a huge number of transactions, to avoid bias in our evaluation, we have limited the number of transactions to 50 per address. Note that, as actual costs might depend on parameters or on storage values, we have only compared the actual cost of transactions that have constant UB's. We have also taken into consideration the cost of the transaction input: 4 gas units for zero bytes and 16 for non-zero bytes. This comparison confirms that the precision of our upper-bounds is very high: around 80% of our upper-bounds only exceed

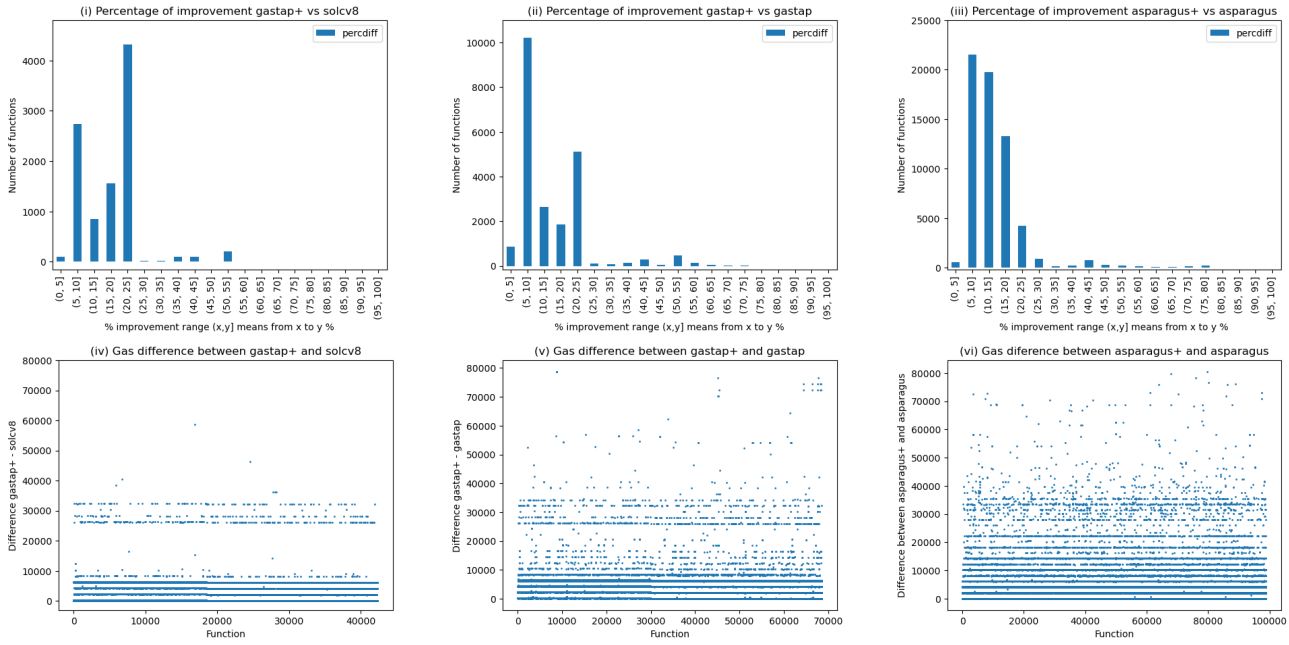

Figure 3: Accuracy gains obtained by using the synthesized storage bounds

between 0-5% the actual gas consumed, and around 15% between 20-50%. This is very precise also considering that we are compiling using solc v8.24 without any kind of optimization while deployed contracts could be compiled with other versions and settings.

5 Related Work and Conclusions

The blockchain technology has brought in new forms of verification problems, such as proving the absence of reentrancy vulnerabilities [10, 20, 20, 29, 35] or of out-of-gas exceptions [7, 13, 38]. This paper achieves more accurate solutions to the out-of-gas verification problem by synthesizing tighter gas upper bounds that provide guarantees of the amount of gas required to safely execute a code. While part of our solution for the synthesis of gas bounds relies on using standard resource analysis (here any of the resource analysis approaches could be used, e.g., [11, 19, 21, 25, 31, 34]), the storage opcodes introduce new challenges that to the best of our knowledge have been not studied yet. The challenges are related to the dynamicity of the cost model for storage: the cost of a storage opcode may depend on whether it is the first access to that key, and also for store opcodes, on how the value being stored modifies the initial and previous contents. Existing tools have been developed for a static cost model in which each instruction is mapped to a fixed known cost (both `gastap` and `asparagus` assume a worst case cost for storage instructions). The novelty of our approach relies on achieving accuracy by using such standard analysis in an unsound way and then fixing the soundness issues in a posterior stage.

The solution we have provided to the synthesis of storage bounds could be useful beyond the blockchain context. For instance, the fact that the first storage access has a larger cost is also inherent to databases, and the same reasoning could be applied to estimate database access costs. Besides, the cost of the last store opcode is higher because it writes persistently on the database/blockchain. We have not distinguished this case in our analysis because this higher cost is not part of the verification problem, i.e., does not account for out-of-gas exceptions, but rather counted at the end to provide some refunds (see [37]). Importantly, the problem of estimating the number of persistent stores can be handled exactly in the same way as for the cold accesses. Finally, our solution based on a worst-case scenario in which the cost of the store opcode commutes between two possible values has some relation to having a probabilistic cost model. However, we are not aware of any resource analysis with a probabilistic cost model. The probabilities are rather associated to the instructions executed in existing frameworks [15, 36]. Finally, there is some tangential relation to the idea of amortized cost analysis [24, 32] in the sense that we are paying high cost in one execution of a store opcode and compensating it with a low cost in the next one. However, we do not see that the existing solutions to amortized cost analysis would be able to solve our problem. ,As future work, we plan to implement a value analysis to infer the contents of the storage keys and be able to infer when the value of a key has changed *wrt.* the previous update and/or if it takes the zero value. This can be integrated in our framework both in the resource analysis and in the Max-SMT corrections to be more precise.

Acknowledgments

This work was funded partially by the Spanish MCIU, AEI and FEDER (EU) projects PID2021-122830OB-C41 and PID2021-122830-OA-C44 and the GREEN project AOC-1662 and SOPA project AOC-2304 by the Ethereum Foundation.

References

[1] [n. d.]. Avalanche C-Chain Network. https://docs.avax.network/build/dapp/c-chain-evm.

[2] [n. d.]. Binance BNB Chain. https://www.bnbchain.org.

[3] [n. d.]. Ethereum. https://ethereum.org.

[4] [n. d.]. SalaryDistribution. https://etherscan.io/address/0x0d0cc4202c6d8acfd2e107ad5326299252bdcfbc.

[5] 2018. Etherscan. https://etherscan.io.

[6] 2024. Solidity Storage documentation. https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html.

[7] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2021. Don't Run on Fumes — Parametric Gas Bounds for Smart Contracts. *Journal of Systems and Software* 176 (2021), 110923. https://doi.org/10.1016/J.JSS.2021.110923

[8] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 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 (Lecture Notes in Computer Science, Vol. 13993)*. Springer, 448–466. https://doi.org/10.1007/978-3-031-30823-9_23

[9] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. 2007. Shape Analysis for Composite Data Structures. In *Computer Aided Verification*, Werner Damm and Holger Hermanns (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 178–192. https://doi.org/10.1007/978-3-540-73368-3_22

[10] Thomas Bernardi, Nurit Dor, Anastasia Fedotov, Shelly Grossman, Alexander Nutz, Lior Oppenheim, Or Pistiner, Mooly Sagiv, John Toman, and James Wilcox. 2020. Preventing Reentrancy Bugs - Another Use Case for Formal Verification. https://www.certora.com/blog/reentrancy.html.

[11] Jason Breck, John Cyphert, Zachary Kincaid, and Thomas Reps. 2020. Templates and recurrences: better together. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 688–702. https://doi.org/10.1145/3385412.3386035

[12] Zhuo Cai, Soroush Farokhnia, Amir Kafshdar Goharshady, and S. Hitarth. 2023. Asparagus: Automated Synthesis of Parametric Gas Upper-Bounds for Smart Contracts. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 882–911. https://doi.org/10.1145/3622829

[13] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiapu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. 2020. GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts. *IEEE Transactions on Emerging Topics in Computing* PP(99) (03 2020), 1–14. https://doi.org/10.1109/TETC.2020.2979019

[14] Agostino Cortesi. 2008. Widening Operators for Abstract Interpretation. In *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*, Antonio Cerone and Stefan Gruner (Eds.). IEEE Computer Society, 31–40. https://doi.org/10.1109/SEFM.2008.20

[15] Ankush Das, Di Wang, and Jan Hoffmann. 2023. Probabilistic Resource-Aware Session Types. *Proc. ACM Program. Lang.* 7, POPL (2023), 1925–1956. https://doi.org/10.1145/3571259

[16] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[17] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1176–1186. https://doi.org/10.1109/ICSE.2019.00120

[18] Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. 2022. Elipmoc: Advanced Decompilation of Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 6, OOPSLA (2022), 77:1–77:27. https://doi.org/10.1145/3527321

[19] Jessie Grosen, David M. Kahn, and Jan Hoffmann. 2023. Automatic Amortized Resource Analysis with Regular Recursive Types. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–14. https://doi.org/10.1109/LICS56636.2023.10175720

[20] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL* 2, POPL (2018), 48:1–48:28. https://doi.org/10.1145/3158136

[21] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 127–139. https://doi.org/10.1145/1480881.1480898

[22] Ákos Hajdu and Dejan Jovanovic. 2020. SMT-Friendly Formalization of the Solidity Memory Model. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 224–250. https://doi.org/10.1007/978-3-030-44914-8_9

[23] S. Hitarth. 2023. Artifact-Asparagus: Automated Synthesis of Parametric Gas Upper-bounds for Smart Contracts. https://doi.org/10.5281/zenodo.8202373

[24] Jan Hoffmann and Steffen Jost. 2022. Two decades of automatic amortized resource analysis. *Math. Struct. Comput. Sci.* 32, 6 (2022), 729–759. https://doi.org/10.1017/S0960129521000487

[25] Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. 2022. Runtime Complexity Bounds Using Squeezers. *ACM Trans. Program. Lang. Syst.* 44, 3 (2022), 17:1–17:36. https://doi.org/10.1145/3527632

[26] Sifis Lagouvardos, Neville Grech, Ilias Tsatiris, and Yannis Smaragdakis. 2020. Precise static modeling of Ethereum "memory". *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 190:1–190:26. https://doi.org/10.1145/3428258

[27] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Transactions on Software Engineering Methodology* 14 (2005), 1–41. Issue 1. https://doi.org/10.1145/1044834.1044835

[28] F. Nielson, H. R. Nielson, and C. Hankin. 1999. *Principles of Program Analysis*. Springer. I–XXI, 1–450 pages. https://doi.org/10.1007/978-3-662-03811-6

[29] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 621–640. https://doi.org/10.1145/3372297.3417250

[30] Roberto Sebastiani and Patrick Trentin. 2015. OptiMathSAT: A Tool for Optimization Modulo Theories. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 447–454. https://doi.org/10.1007/978-3-319-21690-4_27

[31] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 745–761. https://doi.org/10.1007/978-3-319-08867-9_50

[32] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Amortized Efficiency of List Update and Paging Rules. *Commun. ACM* 28, 2 (1985), 202–208. https://doi.org/10.1145/2786.2793

[33] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 17–30. https://doi.org/10.1145/1926385.1926390

[34] P. W. Trinder, M. I. Cole, K. Hammond, H-W. Loidl, and G. J. Michaelson. 2013. Resource analyses for parallel and distributed coordination. *Concurrency and Computation: Practice and Experience* 25, 3 (2013), 309–348. https://doi.org/10.1002/cpe.1898 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1898

[35] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 67–82. https://doi.org/10.1145/3243734.3243780

[36] Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin, and Wenjun Shi. 2019. Cost analysis of nondeterministic probabilistic programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 204–220. https://doi.org/10.1145/3314221.3314581

[37] Gavin Wood. 2024. Ethereum: A secure decentralised generalised transaction ledger. Paris version. https://ethereum.github.io/yellowpaper/paper.pdf.

[38] Ziyi Zhao, Jiliang Li, Zhou Su, and Yuyi Wang. 2023. GaSaver: A Static Analysis Tool for Saving Gas. *IEEE Trans. Sustain. Comput.* 8, 2 (2023), 257–267. https://doi.org/10.1109/TSUSC.2022.3221444

[39] M. Tamer Özsu and Patrick Valduriez. 2011. *Principles of Distributed Database Systems* (3rd ed.). Springer. https://doi.org/10.1007/978-1-4419-8834-8

Received 2024-04-12; accepted 2024-07-03