Practical Verification of Smart Contracts using Memory Splitting

SHELLY GROSSMAN, Tel Aviv University, Israel
JOHN TOMAN, Certora Inc., USA
ALEXANDER BAKST, Certora Inc., USA
SAMEER ARORA, Certora Inc., USA
MOOLY SAGIV, Tel Aviv University, Israel
CHANDRAKANA NANDI, Certora Inc., USA

SMT-based verification of low-level code requires modeling and reasoning about memory operations. Prior work has shown that optimizing memory representations is beneficial for scaling verification—pointer analysis, for example can be used to split memory into disjoint regions leading to faster SMT solving. However, these techniques are mostly designed for C and C++ programs with explicit operations for memory allocation which are not present in all languages. For instance, on the Ethereum virtual machine, memory is simply a monolithic array of bytes which can be freely accessed by Ethereum bytecode, and there is no allocation primitive.

In this paper, we present a memory splitting transformation guided by a conservative memory analysis for Ethereum bytecode generated by the Solidity compiler. The analysis consists of two phases: recovering memory allocation and memory regions, followed by a pointer analysis. The goal of the analysis is to enable memory splitting which in turn speeds up verification. We have implemented both the analysis and the memory splitting transformation as part of a verification tool, CertoraProver, and show that the transformation speeds up SMT solving by up to 120× and additionally mitigates 16 timeouts when used on 229 real-world smart contract verification tasks.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; • **Theory of computation** → *Automated reasoning*.

Additional Key Words and Phrases: pointer analysis, verification, SMT solving, smart contracts

ACM Reference Format:
Shelly Grossman, John Toman, Alexander Bakst, Sameer Arora, Mooly Sagiv, and Chandrakana Nandi. 2024. Practical Verification of Smart Contracts using Memory Splitting. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 356 (October 2024), 32 pages. https://doi.org/10.1145/3689796

1 Introduction

Smart contracts are programs that govern (usually financial) transactions between parties in the context of a distributed ledger or blockchain. They are typically written in high-level languages such as Solidity [88], Vyper [97], Rust [80] and Move [75], but are then compiled into a low-level language supported by the underlying blockchain where the program will actually run, *e.g.,* EVM bytecode, eBPF [47], and WebAsssembly [100]. Finding (and ensuring the absence of) bugs, including those introduced by miscompilation is of paramount importance for this domain because vulnerabilities

Authors' Contact Information: Shelly Grossman, shellygr@mail.tau.ac.il, Tel Aviv University, Israel; John Toman, john@certora.com, Certora Inc., USA; Alexander Bakst, abakst@certora.com, Certora Inc., USA; Sameer Arora, sameer@certora.com, Certora Inc., USA; Mooly Sagiv, msagiv@post.tau.ac.il, Tel Aviv University, Israel; Chandrakana Nandi, chandra@certora.com, Certora Inc., USA.

ACM 2475-1421/2024/10-ART356
https://doi.org/10.1145/3689796

```
1   rule allSame(uint n, uint i1, uint i2, uint sz) {
2     C.Streams s = init(n, i1, i2, sz);
3     assert(forall uint j .
4       (0 <= j && j < sz) => s.k1[j] == i1);
5   }
```

```
1   contract C {
2     struct Streams {uint[] k1; uint[] k2;}
3
4     function init(uint n, uint i1, uint i2, uint sz)
5       pure public returns (Streams memory) {
6         Streams memory s;
7         s.k1 = new uint[](sz);
8         s.k2 = new uint[](sz);
9         s.k1[0] = i1;
10        s.k2[0] = i2;
11        for(uint i = 1; i < sz; i++) {
12          s.k1[i] = s.k1[i-1];
13          s.k2[i] = s.k2[i-1] ** (n + i);
14        }
15        return s;
16    }
17  }
```

(a) (Bottom) A contract that initializes two streams.
(Top) A property in CertoraProver's specification language.

```
1   s := MLOAD 0x40
2   fp0 := 0x40 + s
3   MSTORE 0x40 fp0
4
5   MSTORE s 0x60
6   s_k2 := 0x20 + s
7   MSTORE s_k2 0x60
8
9   k1 := MLOAD 0x40
10  MSTORE k1 sz
11  k1_elems := 0x20 * sz
12  k1_len := 0x20 + k1_elems
13  fp1 := k1 + k1_len
14  MSTORE 0x40 fp1
15  MSTORE s k1
16
17  k2 := MLOAD 0x40
18  // Increment, initialize k2
19  // similar to k1
20  ...
```

(b) Simplified TAC derived from the EVM corresponding to line 6 - line 8 of `init`.

Fig. 1. (Left) A Solidity program and it's specification, and (Right) the TAC corresponding to the program generated by decompiling the Ethereum bytecode corresponding to the Solidity program. 0xN represents hexadecimal numbers.

in smart contracts can be exploited to cause severe financial losses, as has been evident from a myriad of recent examples.[1]

Formal verification can greatly increase the confidence that a contract is free of such vulnerabilities *before* deployment. Due to the tremendous advancements made in the performance of off-the-shelf Satisfiability Modulo Theories (SMT) solvers [30, 44], developing a verification tool using these tools is an attractive approach, as is evident from recent surveys that discuss dozens of tools [45, 64]. At a high level these tools work as follows: given a specification, they extract a verification condition (VC) from the input program. The VC is then checked by an SMT solver: the validity of the VC implies that the program satisfies its specification.

Problem. Verifying EVM programs necessarily requires reasoning about *pointers* and *memory*. EVM's basic modeling of memory as single array means that the underlying SMT solvers must effectively explore an intractably large number of states as it is forced to reason about whether any two memory operations access the same locations. This adds additional burden on off-the-shelf SMT solvers [28, 30, 44, 46], leading to long verification times and frequent timeouts.

Insight. The key insight of this paper is that a *pointer analysis guided memory splitting transformation* applied to the target EVM program as a *pre-processing* step can mitigate timeouts and reduce SMT solving time. We incorporate this insight within a verifier, CertoraProver, that targets EVM bytecode. CertoraProver has a specification language that allows users to write high-level functional correctness properties using invariants, and pre- and post-conditions. CertoraProver compiles the specification and the EVM bytecode into a register-based intermediate representation (IR), called TAC, from which it generates a VC by computing the weakest precondition [29]. We introduce a ***new memory splitting transformation*** on the low-level TAC program before generating VCs, as part of a pre-processing phase. To ensure that this transformation is semantics preserving, it is guided by a ***new memory analysis***. As a result of this pre-processing step, we observed that the

[1] https://rekt.news/

SMT formulas generated by CertoraProver were faster to solve and therefore lead to a significant reduction in solver time and fewer timeouts.

Claims. We make the following three claims in this paper.

- **C1** The proposed memory splitting transformation is crucial for mitigating SMT timeouts when verifying EVM bytecode from real-world smart contract protocols.
- **C2** The memory analysis exposes bugs in many versions of the Solidity compiler.
- **C3** Prior work that uses pointer analysis to speed up SMT solving does not directly apply to EVM bytecode.

Evidence. To support the claims stated above, we provide the following evidence.

- **E1** On a dataset of 229 verification tasks, we show that the memory splitting transformation mitigates timeouts in 16 cases (i.e., the solver did not finish running without memory splitting) and additionally speeds up SMT solving by up to 120× and 2.03× on average.
- **E2** As a result of the memory analysis backing the memory splitting transformation, we have found 5 compiler bugs in different versions of the Solidity compiler.
- **E3** Similar ideas for speeding up SMT solving time have been explored in the context of other programming languages with explicit allocation instructions (*e.g.*, `malloc` in C, `new` in JVM bytecode) [18, 58]. However, this model does not apply to the EVM: memory is simply an unstructured, monolithic array of bytes. The entire (256-bit) address space is available to a contract, and there is no traditional allocation instruction. In order to implement high-level data structures such as dynamically- sized arrays, the Solidity compiler allocates blocks of memory using a simple bump allocation scheme. Therefore, allocations of dynamically-sized data structures appear to be nothing more than a sequence of ordinary memory accesses and pointer arithmetic, complicating the application of existing techniques.

CertoraProver is an industrial tool regularly used by smart contract developers for specifying and verifying high-level functional correctness properties of their programs. While most users of CertoraProver are experts in blockchain protocol development and proficient in Solidity, few are formal verification experts. Long verification times and timeouts are therefore particularly undesirable for a tool like CertoraProver.

This paper presents a memory splitting rewrite and a memory analysis that enables it. We have found that the rewrite is crucial for speeding up CertoraProver, making it suitable for use in this domain. Overall, we make the following key contributions:

1. A *memory splitting transformation* targeting EVM bytecode that leads to faster SMT solving and fewer timeouts when verifying smart contracts.
2. A new and practical *memory analysis* for EVM bytecode that guides the memory splitter.
3. An implementation of both techniques in a verification tool, CertoraProver and an evaluation on 229 real-world verification tasks from external users demonstrating that memory splitting speeds up SMT solving by up to 120× and additionally mitigates 16 SMT timeouts.
4. A summary of 5 compiler bugs we have found as a result of the memory analysis.

2 Background and Overview

This section gives an example of a Solidity program, a high-level property expressed in CertoraProver's specification language, and describes the end-to-end workflow of CertoraProver. It also explains the key challenge encountered in naively generating VCs.

2.1 CertoraProver's Workflow

Figure 1a (bottom) shows a Solidity program and Figure 1a (top) shows a high-level property (a "rule" named `allSame`) of this contract in CertoraProver's specification language. The Solidity

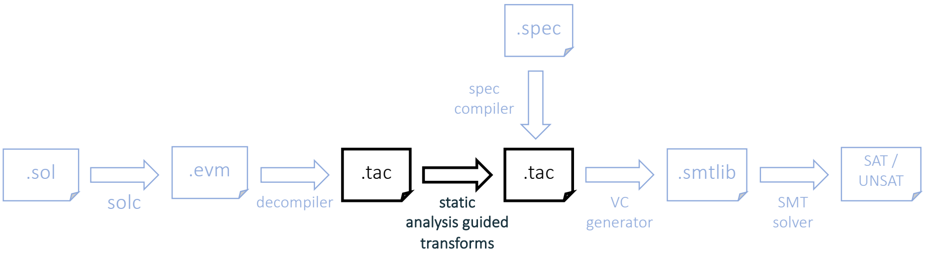


Fig. 2. Workflow diagram of CertoraProver. CertoraProver compiles Solidity code using off-the-shelf compilers (shown by solc) to generate EVM bytecode. Then, it converts the bytecode to a three-address representation called TAC. it applies various transformations on the TAC. The memory splitting rewrite presented in this paper is part of this phase. The specification (shown by `.spec`) is translated to TAC and function calls are inlined. CertoraProver then generates the verification condition and sends it to off-the-shelf SMT solvers.

program consists of a single a "contract" (analogous to an object in object-oriented languages), `C`. A struct, `Stream` is defined in `C` with two dynamically sized arrays, `k1` and `k2`. The function `init` instantiates the struct as `s` and initializes the arrays in memory (indicated by the keyword, `memory` on line 6) to point to two new arrays shown by `new uint[](sz)` where `sz` is the size of the array. On line 9 and line 10, index 0 of both arrays is set. The for loop goes over the rest of the indices in both arrays and sets them as shown. Finally the function returns `s`.

In CertoraProver, specifications (pre- and post-conditions) are expressed in a declarative language. The core features of this language are standard, similar to tools like Dafny [69]. In the rule `allSame`, we first invoke the `init` function from the contract and then add a post-condition stating that for all valid indices, the entries in the array `s.k1` must be equal to `i1`.

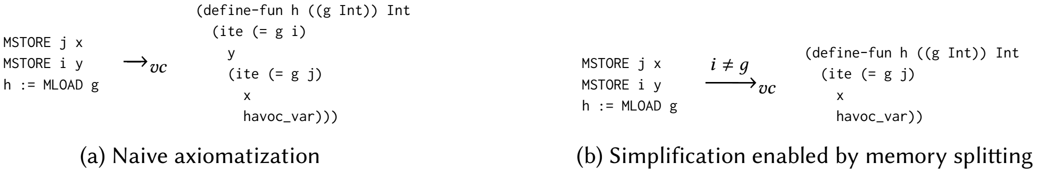
Figure 2 shows the workflow CertoraProver follows to prove this property. First, CertoraProver uses the Solidity compiler to generate the EVM bytecode of the contract. EVM bytecode targets the EVM stack machine, which CertoraProver then converts to a register machine, called TAC (short for three-address code). For this, CertoraProver follows the approach introduced by Vallée-Rai et al. [95] in the Soot framework. A similar approach for decompiling EVM bytecode can be found in other tools [21, 37, 54, 68, 86]. The memory analysis and the memory splitter this paper presents are introduced at this phase in CertoraProver. Next, the specification is also compiled to TAC, followed by inlining the function calls from the TAC of the Solidity program. Ultimately, CertoraProver uses a standard algorithm [29] to compute the VC which it then sends to SMT solvers [30, 44].

2.2 The Problem and the High-Level Solution

One major challenge with generating VCs lies in handling stateful memory load and store operations that are part of EVM bytecode (and therefore also part of TAC). These operations are not directly representable in an SMT encoding. Figure 3a shows a naive way to encode them—it illustrates how the result of an `MLOAD` (memory load) operation at a pointer `g` can be translated into an expression that *case-splits* over all of the preceding `MSTORE` operations that could have written to `g`.[2]

However, such a translation fails to scale in realistic programs (see Section 7). This is because the translation assumes that *any* two pointers (that is, EVM memory array indices that are operands to an `MSTORE` or `MLOAD` operation) may alias, leading to a blow-up in the number of cases that the SMT solver needs to consider. If we can prove that two pointers *must not* alias, then the complexity of the VCs significantly reduces. Figure 3b shows how we can apply facts learned from a memory

[2] The familiar reader will recognize this translation as instantiations of the read-over-write axioms [74].



```
MSTORE j x
MSTORE i y
h := MLOAD g
```

$\longrightarrow_{vc}$

```
(define-fun h ((g Int)) Int
  (ite (= g i)
    y
    (ite (= g j)
      x
      havoc_var)))
```

(a) Naive axiomatization

```
MSTORE j x
MSTORE i y
h := MLOAD g
```

$\xrightarrow{i \neq g}_{vc}$

```
(define-fun h ((g Int)) Int
  (ite (= g j)
    x
    havoc_var))
```

(b) Simplification enabled by memory splitting

Fig. 3. (a) Example showing axiomatization of memory operations (using SMTLIB syntax to the right of the arrow). MLOAD and MSTORE operations are ubiquitous and therefore this naive transformation fails to scale both in terms of the number and size of such axioms. (b) CertoraProver's memory splitter enables rewrites by inferring facts like $i \neq g$. $\longrightarrow_{vc}$ refers to the VC generation process. havoc_var is an unconstrained value.

analysis to simplify this axiomatization: if we know that i is never equal to g, we can prune that check, ultimately simplifying the resulting SMT query.

This is one of the key ideas that makes CertoraProver scale to real-world verification tasks: it employs a new *memory splitting transformation* that allows it to generate these faster-to-solve SMT queries. Even for the simple example program and property in Figure 1a, we found that when running CertoraProver to verify the AllSame property, the memory splitting transformation sped up the SMT solving time by 2×. As hinted above, the transformation itself is enabled by a novel memory analysis that CertoraProver performs on the TAC program. The rest of the paper will discuss this transformation and the novel memory analysis that enables it.

3 TAC and Its Translation to SMT-LIB

Before discussing the memory analysis and transformations that CertoraProver employs to speed up smart contract verification, in this section we first provide a brief overview of how memory is defined in the Ethereum virtual machine. We then fix a formalization of TAC to ground our discussion. Finally, we sketch the basic translation from TAC to VCs whose validity imply the correctness of the analyzed contract.

3.1 EVM Memory

The EVM is a stack machine where the stack can hold up to 1024 words, each 256 bits (32 bytes) long. The EVM has two main notions of mutable "state" — (1) *memory*, a volatile form of storage that is deleted at the end of each contract call, and (2) *storage*, which persists throughout the lifetime of a contract's deployment.[3] Memory is byte-addressable, but load and store operations read and write word-sized values, respectively. Storage addresses are not valid in memory (and vice versa), and therefore the remainder of this paper only concerns itself with pointers to *memory*.

As the EVM does not expose memory allocation abstractions (such as malloc), compilers must implement their own. The Solidity compiler allocates memory using *bump-allocation*. The compiler uses a monotonically increasing *free pointer*, *fp*, to indicate the beginning of unused space in the memory array. To allocate memory for an object during a contract call, the compiler first saves the current value of the free pointer, increments the pointer by the size of the allocated object, and then returns the saved value as the fresh pointer to the allocated memory. The memory address of the free pointer is (by convention) the constant address 0x40 (in hexadecimal). A pointer, therefore, is simply a memory index indicating the beginning of an allocated block. Accesses to values *within* the allocated memory are achieved by *pointer arithmetic*, i.e., adding values to these special integers. There is no scheme for freeing memory since contract calls are typically short-lived.

[3] EVM also has separate areas for *calldata* and for *returndata* that are temporary read-only locations holding information passed to and returned from contracts respectively.

$$\textbf{Variables } x \in \mathcal{V} \quad \textbf{Literals } c \in \{0,1\}^{256} \quad \textbf{Labels } l \in \mathcal{L}$$

$$\begin{array}{rrcl}
\textbf{Immediates} & i & ::= & c \mid x \\
\textbf{Expressions} & e & ::= & i \mid x[i] \mid x[i := i] \mid i \oplus i \\
\textbf{Basic Instructions} & I_b & ::= & x \leftarrow e \mid \mathtt{BRANCH}\, x\, l\, l \\
\textbf{Memory and Storage} & I_m & ::= & x \leftarrow \mathtt{MLOAD}\, x \mid \mathtt{MSTORE}\, x\, i \\
\textbf{Instructions} & I & ::= & I_m \mid I_b \\
\textbf{Labeled Instructions} & li & ::= & l : I \\
\textbf{Program} & P & ::= & li*
\end{array}$$

Fig. 4. Syntax for a subset of the TAC IR and the Control Flow Graph (CFG) definition. A label is a unique identifier for each instruction in the CFG.

3.2 TAC

CertoraProver works on a three-address intermediate representation (IR) dubbed TAC, generated by decompiling the EVM bytecode. The main purpose of the decompiler is to convert the EVM stack machine into traditional register machine code.

Figure 4 shows a subset of the TAC language that is relevant to the contributions of this paper. Immediates are constants or variables. A constant is a 256 bit bitvector, which is consistent with the EVM. Note that a variable can represent an array or a scalar. Expressions include both binary and unary operators that cover arithmetic expressions, logical expressions, and comparisons. Expressions can also be array lookups and updates. Importantly, expressions are not recursive since this is a three-address representation. Instructions in TAC can be *basic* which includes assignments and BRANCH, or they can be *memory related*: MLOAD and MSTORE. v ← MLOAD w reads 32 bytes of memory starting at location w in to v. MSTORE l s writes s to memory from location l to l + 32. The BRANCH instruction indicates control flow in case of conditionals. The control flow graph is defined as a list of *(label, instruction)* pairs.

Figure 1b shows a simplified representation of the TAC code corresponding to the (bytecode of) the body of the init method shown in Figure 1a. Recall that the free pointer's value is always stored at location 0x40 by the Solidity compiler. On line 1 the program first loads the value of the free pointer to the register, s. The pointer for each of the arrays of the stream are of size 0x20 (i.e., 32) bytes. The free pointer is therefore bumped on line 2 by adding 2 times 0x20, i.e., 0x40 to the current position of the free pointer in s. The new value is stored at 0x40. Note that, like C structs, the location/offset of s also the location of the first field of s, k1.

The Solidity compiler reserves the 32 bytes starting at location 0x60 to always hold zero; this magic constant 0x60 is used as the default value for dynamic arrays.[4] Therefore, line 5–line 7 initializes the two arrays in the struct in s— first, the program initializes the first array by storing 0x60 in s (which, recall, is also the location of the first field), then computes s_k2 to point to the location of the second field, and finally stores 0x60 at the location s_k2. Next, starting at line 9, allocation of s.k1 occurs—the position of the free pointer is loaded into k1, where the value of the size of the array sz is stored; k1_elems is computed to allocate sufficient memory for all sz elements (0x20 for each); k1_len is computed to represent the total memory required for s.k1 which includes an additional 0x20 bytes for storing the size of the array itself; the free pointer is bumped and stored again at 0x40 and finally the location k1 of the freshly array is stored at s. A similar set of operations also happen for s.k2 from line 17 onwards.

The TAC programs are encoded in SMT-LIB by first replacing all memory and storage instructions (I_m) with basic instructions (I_b) by applying $\mathsf{Simplify}(i) : I_m \longrightarrow I_b$ which we define as:

[4] https://docs.soliditylang.org/en/latest/internals/layout_in_memory.html

$$\mathsf{Simplify}(i) \quad = \quad \begin{cases} x \leftarrow M[p] & i \equiv x \leftarrow \texttt{MLOAD}\,p \\ M \leftarrow M[p := v] & i \equiv \texttt{MSTORE}\,p\,v \end{cases}$$

Where $M \in \mathcal{V}$ is a distinguished variable that explicitly models memory as an array.

Example 3.1. We show the TAC for the example in Figure 1b after going through Simplify. Here, `tacM` represents the distinguished variable that models memory.

```
s := MLOAD 0x40
fp0 := 0x40 + s
MSTORE 0x40 fp0

MSTORE s 0x60
s_k2 := 0x20 + s
MSTORE s_k2 0x60
...
```

$\rightsquigarrow$

```
s := tacM [0x40]
fp0 := 0x40 + s
tacM := tacM [0x40 := fp0]

tacM := tacM [s := 0x60]
s_k2 := 0x20 + s
tacM = tacM [s_k2 := 0x60]
...
```

Next, the resulting program, which contains only basic instructions, is transformed into static single-assignment (SSA) form [43] and finally translated into SMT-LIB following Barnett and Leino [29]'s algorithm for computing the weakest precondition. In particular `tacM` (and its SSA renaming) is translated as SMT-LIB array.

4 Analysis Walk-through

Now that we have described how memory operations in TAC are simplified (using Simplify) and the TAC is translated to SMT-LIB, we explain the memory analysis in CertoraProver that guides the *memory splitting* transformation to generate faster-to-solve SMT-LIB formulae. The memory analysis is crucial for the memory splitter and is one of the key contributions of this paper.

Since EVM does not have any special instructions that indicate memory allocations, CertoraProver must compute which *allocated objects* each memory pointer *may* point to. Pointers that point to disjoint sets of objects do not alias: this aliasing information is then used to optimize the translation to SMT as mentioned previously and evaluated in later sections. In general, programs may allocate an unbounded number of objects. Thus, pointer analyses typically approximate the set of allocated runtime memory locations by a set of *abstract* locations [25, 91]. In languages like Java, the source code location corresponding to the `new` keyword can serve as the abstract location corresponding to the memory addresses returned by that particular allocation.

In the absence of any such primitives, CertoraProver performs a novel ***allocation analysis***, which computes both the set of abstract locations and the program points where allocations occur. This analysis identifies allocations by identifying increments of the free pointer. As described earlier, Figure 1b shows the TAC corresponding to the allocation and initialization of s, `s.k1`, and `s.k2`. CertoraProver identifies the abstract locations created at: (1) line 1, corresponding to the allocation of the struct s; (2) line 9, corresponding to the allocation of the array `s.k1`; and (3) line 17, corresponding to the allocation of the array `s.k2`. After identifying the allocations, CertoraProver constructs a points-to graph—for each pointer, it computes the set of abstract locations that it may refer to. For each abstract location, CertoraProver computes the type of the stored object (such as struct or an array), and the abstract locations that may be referenced by the stored object.

Computing these may-alias sets of abstract locations is a standard abstract interpretation-based pointer analysis. However, in EVM, pointer arithmetic is unrestricted: unlike languages such as C, using pointer arithmetic to extend a pointer beyond the bounds of the referenced object is a well defined operation. Thus, to ensure the soundness of the heap typing information, CertoraProver must check the following conditions:

R1 Each object is initialized correctly, *establishing* the validity of the points-to information
R2 Memory operations are *within bounds* (thus *maintaining* the correctness of the abstraction).

```
1   // H = l_s ↦ Ŝtr([0 ↦ ArrPtr(l_k1), 32 ↦ ArrPtr(l_k2)]), l_k1 ↦ Ârr(INT), l_k2 ↦ Ârr(INT)
2   // N = v ↦ ⟨[0, MAX_INT], ∅⟩
3   // P = s ↦ ℬ(l_s, 64)
4   solidity_guard:
5     kptr := MLOAD s        // P = ..., kptr ↦ ArrPtr(l_k1)
6     length := MLOAD kptr // N = ..., length ↦ ⟨[0, MAX_INT], {LenOf(kptr⟩})
7     ok := i < length
8     BRANCH ok write_k1 revert
9
10  // N = ..., i ↦ ⟨[0, MAX_INT], {SafeIndOf(kptr⟩})
11  write_k1:
12    elem_off := 0x20 * i          // N = ..., elem_off ↦ ⟨[0, MAX_INT], {SafeElemOf(kptr⟩})
13    array_off := 0x20 + elem_off // N = ..., array_off ↦ ⟨[0, MAX_INT], {SafeArrOffset(kptr⟩})
14    elem_ptr := kptr + elem_off  // P = ..., elem_ptr ↦ ℰ(l_k1, ⊥, kptr)
15    MSTORE elem_ptr v
```

Fig. 5. Simplified TAC code derived from the EVM corresponding to the write of `s.k1[i]` at line 12. Relevant abstract state is shown as comments. Ellipses "..." indicate that the abstract state component that follows is an *update* to the preceding state. Components are omitted if they do not change relative to the previous state.

Requirement **R1** is handled by an ***initialization analysis*** that finds the program locations where a newly allocated object has been completely initialized. The initialization code is generated by the Solidity compiler, and is unrelated to any initial values that the *user* may specify. Crucially, the initialization analysis identifies a (unique) program location after which it is safe to perform reads from a newly allocated object. Like in C, a freshly allocated block of memory may contain arbitrary values. If fields of a freshly allocated object are read before being initialized with coherent, well-typed values, points-to relationships become difficult or impossible to infer; any apparent pointer value might actually be junk read from memory. In other words, the initialization analysis guarantees that any reads of an object's fields yield type-correct values. This strongly-typed heap property is crucial for the precision of the ***points-to analysis*** that follows. If the initialization analysis succeeds, the points-to analysis proceeds to build the points-to graph. The analysis is an abstract interpretation whose domain is a reduced product of numerical (N), pointer (P), and abstract heap (H) domains. The numeric store N maps variables to a numeric abstraction, P maps variables to abstract pointer values, and H maps abstract locations to abstract heap values.

We now return to the example in Figure 1b to show how CertoraProver proves array accesses in Figure 1a to be safe. Figure 5 shows the TAC representation of the write to `s.k1` (the access `s.k1[i]` at line 12 of Figure 1a). The Solidity compiler inserts its own bounds checks before array accesses. This is why we see the block labeled `solidity_guard`, which performs the check. Figure 5 is annotated with the abstract state on entry to each block, and after each command. For ease of exposition, we use a simplified presentation of our analysis formalization (Section 5), and omit abstract values that are not relevant (or are uninteresting) to the discussion.

On entry to `solidity_guard`, the abstract heap has three mappings (line 1). The location l_s stores an (abstract) struct value with two fields: one (at word offset 0) is $\mathit{ArrPtr}(l_{k1})$, a pointer to an array at abstract location l_{k1}, and the other (at word offset 32) is $\mathit{ArrPtr}(l_{k2})$, a pointer to an array at abstract location l_{k2}. The abstract locations l_{k1} and l_{k2} both contain arrays of integers. N includes mapping for v (the value that will eventually be stored to `s.k[i]`), $\langle[0, \mathtt{MAX\_INT}], \emptyset\rangle$) (line 2).

The numeric abstract value $\langle[lb, ub], Q\rangle$ denotes integers whose values are bounded by the interval $[lb, ub]$, and that are additionally refined by Q, a set of *qualifiers*, which are predicates about the value to which it is attached. In this case, the complete interval $[0, \mathtt{MAX\_INT}]$ and empty qualifier set indicate only that v is some integer, (i.e., nothing else is known about it). P includes $s \mapsto \mathcal{B}(l_s, 64)$, which says that s points to the beginning of (that is, field 0 of) a struct at abstract location l_s. The second component indicates that the total size of this struct is 64 bytes. (line 3).

```
s := tacM [0x40]
fp0 := 0x40 + s
tacM := tacM [0x40 := fp0]
s2 = 0x20 + s // pointer to s.k2
// initially both pointers point to zero slot
tacM := tacM [s  : (s  + 0x20)  := 0x60]
tacM := tacM [s2 : (s2 + 0x20)  := 0x60]




// allocating for the array s.k1
k1 := tacM [0x40]
tacM := tacM [k1 : (k1 + 0x20) := sz]
k1_elems := 0x20 * sz
k1_len := 0x20 + k1_elems
fp1 := k1 + k1_len
tacM := tacM [0x40 := fp1]
/*
computing index 0 of s.k1
First entry is the size of s.k1,
so we add 0x20 to get index 0.
We then initialize all elements of s.k1 to 0
*/
k1_idx_0 = k1 + 0x20
tacM := tacM [k1_idx_0 : k1_idx_0 + k1_elems := 0]
tacM := tacM [s : (s + 0x20) := k1]
// s.k1[0] = i1;
s_k1_0 = 0x20 + tacM [s]
tacM := tacM [s_k1_0 : (s_k1_0 + 0x20) := i1]





// same treatment for k2 as shown for k1.
k2 := tacM [0x40]
tacM := tacM [k2 : (k2 + 0x20) := sz]
k2_elems := 0x20 * sz
k2_len := 0x20 + k2_elems
fp2 := k2 + k2_len
tacM := tacM [0x40 := fp2]
k2_idx_0 = k2 + 0x20
tacM := tacM [k2_idx_0 : k2_idx_0 + k2_elems := 0]
tacM := tacM [s2 := k2]
// s.k2[0] = i2;
s_k2_0 = 0x20 + tacM [s2]
tacM := tacM [s_k2_0 : (s_k2_0 + 0x20) := i2]
```

```
s := tacM [0x40]
fp0 := 0x40 + s
tacM := tacM [0x40 := fp0]
s2 = 0x20 + s // pointer to s.k2
// initially both pointers point to the zero slot
tacMk1 := tacMk1 [s  : (s  + 0x20) := 0x60]
tacMk2 := tacMk2 [s2 : (s2 + 0x20) := 0x60]
/*
memory is partitioned for each array and
within each array, partitioned twice again.
tacM1_1 is the region for the length of s.k1
*/
k1 := tacM[0x40]
tacM1_1 := tacM1_1 [k1 : (k1 + 0x20) := sz]
k1_elems := 0x20 * sz
k1_len := 0x20 + k1_elems
fp1 := k1 + k1_len
tacM := tacM [0x40 := fp1]
/*
tacM1_2 is the region for the elements of s.k1
tacMk1 is now k1 which points to s.k1
We show simplified code for
0 initialization of all elements.
*/
k1_idx_0 = k1 + 0x20
tacM1_2 := tacM1_2 [k1_idx_0 : k1_idx_0 + k1_elems := 0]
tacMk1 := tacMk1 [s : (s + 0x20) := k1]
// s.k1[0] = i1;
s_k1_0 := 0x20 + tacMk1[s]
tacM1_2 := tacM1_2 [s_k1_0 : (s_k1_0 + 0x20) := i1]
/*
Same treatment below for k2 as shown for k1
but in a different partition of the memory
shown by tacM2_1 (length of s.k2) and
tacM2_2 (elements of s.k2).
*/
k2 := tacM[0x40]
tacM2_1 := tacM2_1[ k2 : (k2 + 0x20) := sz]
k2_elems := 0x20 * sz
k2_len := 0x20 + k2_elems
fp2 := k2 + k2_len
tacM := tacM [0x40 := fp2]
k2_idx_0 = k2 + 0x20
tacM2_2 := tacM2_2 [k2_idx_0 : k2_idx_0 + k2_elems := 0]
tacMk2 := tacMk2 [s2 : (s2 + 0x20) := k2]
// s.k2[0] = i2;
s_k2_0 := 0x20 + tacMk2[s2]
tacM2_2 := tacM2_2 [s_k2_0 : s_k2_0 + 0x20 := i2]
```

Fig. 6. Simplified TAC code without memory splitting rewrite (left) and with memory splitting rewrite (right). To keep the program simple and easier to read, we use ranges (e.g., `tacM1_1 := tacM1_1 [k1 : (k1 + 0x20) := sz]`) to indicate that from offset `k1` to `k1 + 0x20`, the value of `sz` is written to memory. Notice that the code on the left is essentially what we obtained by applying Simplify to the code shown in Figure 1b.

For requirement **R2**, the analysis uses path information to prove that the memory accesses maintain the consistency of the points-to abstractions. The TAC program first reads the length of `s.k1` into `length`. The first `MLOAD` instruction reads the `k1` field by "dereferencing" `s`, which, as `P` indicates above, is the first field of the struct at l_s. The pointer state is updated to now also include `kptr` $\mapsto$ *ArrPtr*(l_{k1}), which indicates it is an array pointer to l_{k1} (line 5); that is, the address held in the first field of the object at l_s. Next, CertoraProver assigns to `length` the abstract numeric value ⟨[0, `MAX_INT`], {`LenOf(kptr)`⟩ (line 6). The pointer state indicates that `kptr` is an array pointer, a read from which (via the `MLOAD`) is always considered safe because as explained in Section 3.2, this is a pointer to the beginning of the array which stores the length of the array.

Since `length` is assigned by reading an array length, its abstract value includes the qualifier `LenOf(kptr)`, indicating that is known to be the length of the array pointed to by `kptr`. The guard block ends by jumping to `write_k1` only if `i` is less than `length`. Due to the comparison against `length`, whose abstract value includes `LenOf(kptr)`, CertoraProver generates the qualifier `SafeIndOf(kptr)` for `i` in the numeric store, `N` (line 10). The `SafeIndOf` qualifier indicates that the qualified integer (that is, `i`) is known to be a safe *logical* index of `kptr`; that is, `i` must be less than `kptr`'s length which is exactly what the path condition implies. Similar reasoning justifies the generation of `SafeElemOf(kptr)` for `elem_off` at line 12 (*i.e.,* `elem_off` is the *byte* offset of element `i`, since each element is `0x20` bytes wide), and `SafeArrOffset(kptr)` at line 13 (recall from Section 2.1 that the first word of the array object is the array's size, so the first *element* is offset by `0x20` bytes).

Adding a value with the qualifier `SafeArrOffset(kptr)` to the beginning of the object pointed to by `kptr` produces the abstract value $\mathcal{E}(l_{k1}, \bot, kptr)$ (line 14). This abstract value represents a valid array element pointer for the array with abstract location l_{k1}, and the variable which points to the start of the array is `kptr`. The ⊥ component is used for proving certain accesses safe, and is explained in Section 5.3. As `elem_ptr` is a safe element pointer, CertoraProver finally checks that the type of `v` matches the element type of the referenced array, i.e., `INT`. With this final check, CertoraProver concludes that the write is safe and does not violate memory safety.

Once the abstract interpretation is complete, CertoraProver uses its results to generate *must-not-alias* facts as follows. The abstract pointer store, `P`, maps each pointer to a *set* of abstract locations. Suppose `p` references locations `L` and `q` references locations `M`. Then, if the intersection of `L` and `M` is empty, `p` and `q` must not alias, and CertoraProver may deduce that `p != q`. CertoraProver passes these facts to a downstream program transformation pass, called *memory splitting rewrite* which then uses the facts to split memory operations when generating VCs. Returning to our example in Figure 1b, CertoraProver concludes that the accesses to `s`, `s.k1`, and `s.k2` correspond to disjoint memory regions. This allows the *memory splitter* to rewrite the TAC by separating the memory operations on the two arrays to use two separate array variables (Section 3). Without the memory analysis, the memory splitter cannot soundly split these memory operations.

Figure 6 shows a simplified version of the TAC without the memory splitting transformation (left) and with the memory splitting transformation (right). The program on the left shows that without memory splitting, writes to both `s.k1` and `s.k2` use the same memory variable `tacM`. The program on the right shows that the memory analysis has split memory into four parts—`tacM1_1` represents an array containing the size of `s.k1`, `tacM1_2` represents an array containing the `k1_elems` elements of `s.k1` (initialized to 0 on Line 26), `tacM2_1` represents an array containing the size of `s.k2`, and `tacM2_2` represents an array containing the `k2_elems` elements of `s.k2` (initialized to `0` on Line 44). The free pointer, `fp` is updated as shown in the figure. `tacMk1` and `tacMk2` are two memory variables that store pointers to the two arrays. Initially both point to the zero slot (`0x60`). Later, they are updated to point to the start of the two arrays, `k1` and `k2`. For `s.k1`, Line 29-30 shows the split TAC code corresponding to the Solidity statement shown on Line 28 that uses `tacMk1` and `tacM1_2` (similarly for `s.k2`).

Ultimately, splitting the memory operations to disjoint arrays lowers the burden on the SMT solvers which would otherwise have to either infer these facts (leading to slowdowns) or assume that all memory accesses happen on a single giant array of bytes (leading to imprecision).

5 Detailed Description of the Analyses

We now describe the analyses in more detail. Recall that CertoraProver performs a memory analysis in order to enable memory splitting that speeds up SMT solving in the context of smart contract verification. To achieve this, the analysis computes a memory model of the input TAC program. This comprises three steps: an allocation analysis to detect new allocations, an initialization analysis to detect locations that are safe to read, and ultimately a pointer analysis that produces a points-to

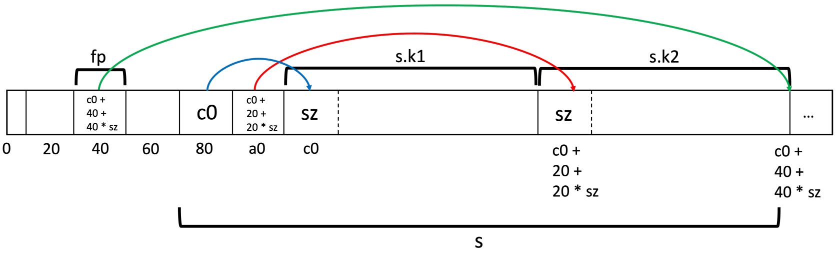


Fig. 7. Illustrating the memory layout after the Solidity compiler allocates memory for the two struct arrays in the program in Figure 1a. All numbers are in hexadecimal. The slot `0x60` is the "zero slot" as per Solidity's documentation. `0x40` stores the value of *fp*. The first element for both `s.k1` and `s.k2` is `sz` which is the size of the array. Each element needs 32 (`0x20`) bytes.

graph whose nodes summarize possibly many memory locations. This points-to graph is the input to the memory splitting transformation.

The following sections describe the steps of the following high-level algorithm of CertoraProver:

```
$evmAnalyzer(cfg) =$
    let $\mathcal{W} = runAllocationAnalysis(cfg)$ in
    let $\mathcal{I} = runInitializationAnalysis(cfg, \mathcal{W})$ in
    let $\widehat{\mathcal{P}} = runPointsToAnalysis(cfg, \mathcal{W}, \mathcal{I})$ in
    $memSplit(cfg, \widehat{\mathcal{P}})$
```

5.1 Allocation Analysis: Marking New Allocation Points

In order to construct a partitioned model of memory, CertoraProver first classifies and labels memory into disjoint regions. At a high level, each region will denote (possibly many) contiguous blocks of memory: in a language like C these would be the blocks returned by `malloc`. However, due to the absence of such a memory allocation abstraction, CertoraProver performs an *allocation analysis* to identify the points where memory is reserved for later use by the program. In addition, the analysis classifies each allocation with a type for the allocation: whether the allocated object is a fixed-size block (*e.g.,* a struct) or a dynamically-sized object (*e.g.,* an array). While this classification is made heuristically, the later analyses *guarantee* the memory safety of the program with respect to the output of the allocation analysis. In the remainder of this section we will take a closer at the various parts of the analysis.

5.1.1 Abstract Allocations. The allocation analysis partitions the heap into a set of *abstract locations*: each abstract location serves as a unique name for the set of memory addresses allocated at a particular program location. As different executions of the program may visit the same allocation site multiple times, an abstract location (which is static) may summarize a statically-unknown number of such address blocks.

We define *abstract allocation* as program points that performs an allocation. An abstract allocation is in 1-to-1 correspondence with an abstract location. The key invariant of the allocation analysis therefore is that **distinct allocation points (and hence abstract locations) refer to disjoint memory locations**. The Solidity compiler allocates objects by inlining a simple bump allocation scheme utilizing a distinguished free pointer, which we refer to as *fp*. To allocate a pointer to *k* bytes, the program first saves the value of *fp*.

```
fp1 := MLOAD fp
r1 := fp1
MSTORE r1 ...
fp2 := MLOAD fp
r2 := fp2
MSTORE r2 ...
MSTORE fp ...
```

Fig. 8. Distinct reads observing the same *fp* (`0x40`) value.

This serves as the value of the pointer, since *fp* points to the first "unreserved" memory address. The program updates *fp* to $fp + k$, reserving k bytes. Thus, the next allocated address (a read of *fp*) is disjoint from the previous allocation. At first blush, it would seem that each read of *fp* could serve as a potential abstract allocation point in the program. However, it is *not* the case that *every* read of *fp* yields a distinct value— during the course of a single allocation, *fp* may be read multiple times: each read may thus not correspond to a separate allocation.

In the TAC snippet in Figure 8, treating the two distinct reads of *fp* (i.e., `r1` and `r2`) as two different abstract locations is incorrect because the two reads observe the same value of *fp*. Dually, *conflating* two reads of *fp* which we know will *always* observe different values leads to precision loss by grouping together distinct memory into a larger partition.

Thus, when identifying abstract locations, it is critical to precisely group together families of reads that *may* read the same value of *fp*, while separating reads that *must not* read the same value.

5.1.2 Read Numbering. To identify free pointer reads that must observe the same value, our analysis first assigns a unique integer to each *update* to *fp*. We denote this domain of numbers using $\mathcal{NW}$, and will denote the number assigned to a free pointer write at l with $\mathcal{NW}_l$. Figure 9 shows this algorithm. It takes the control flow graph of the TAC program (`cfg`) as input and iterates over all the instructions in all the blocks to assign $\mathcal{NW}_l$ to updates of *fp* (checked by `cmd.isFpUpdate()`).

Then, based on this initial assignment, every program point is annotated with its *reaching free-pointer number*, $\mathcal{NR} \in \mathcal{NW}$, using a Static Single Assignment (SSA) scheme [43]. We treat each update of the free pointer as a write to a synthetic variable, and compute for each program point the *fp* update number that reaches it. We denote the $\mathcal{NR}$ number at a given program label l by $\mathcal{NR}_l$. After this numbering, two reads at program points l and l' with the same value of $\mathcal{NR}_l$ and $\mathcal{NR}_{l'}$ for which there does not exist an intervening write *must* observe the same value of *fp* (from the assumed correctness of the SSA-numbering). Conversely, any two pointers with the same $\mathcal{NR}$ *may* alias with one another.

```
def numberWrites (cfg):
  number = 0
  nws = {}
  blocksToNumber = {}
  for block in cfg.blocks:
    for cmd in block.cmds:
      if (cmd.isFpUpdate()):
        nw[cmd.l] = number++
        for f in dominanceFrontier[block]:
         if (f not in blocksToNumber):
           blocksToNumber[f] = number++
  return (blocksToNumber, nws)
```

Fig. 9. Algorithm for assigning each *fp* update a unique number.

However, two reads with *different* values of $\mathcal{NR}_l$ may not necessarily observe different values of *fp*. Figure 10 illustrates this—since only one of the paths includes a write the final read to `p2` will have a different read numbering from the read to `p1`, even though there is potentially an execution where the write is skipped, and hence an execution where the two reads yield the same result.

The analysis therefore checks the ***definite-write property***—*every read of fp must be followed by a single, unique write without any intervening updates to* $\mathcal{NR}$. In other words, any manipulation of the free pointer must be confined to a region of code that must have the same $\mathcal{NR}$. This property ensures that reads of distinct values have distinct read numberings, and thus $\mathcal{NR}$ by itself is a sound basis for abstract locations A. These reaching free-pointer numbers form the basis for abstract locations in the points-to analysis (Section 5.3).

5.1.3 Classifying Abstract Allocations. For statically-sized objects (i.e., structs or statically sized arrays) *fp* is incremented by a simple constant. However, dynamically sized objects (i.e., arrays) are allocated using multiple instructions (Figure 12). Recognizing these patterns for each allocation is crucial to distinguishing between arrays and structs. For the purposes of our formal development

CertoraProver identifies the following patterns of allocations: (1) *constant blocks ($CB(k)$)*, are for statically sized arrays or structs, where *fp* is incremented by a statically known, constant amount k, (2) *dynamic blocks (DB)*, is for arrays allocated with `new` and an explicit size argument.[5] The Solidity compiler generates code that allocates the number of bytes necessary to hold the requested number of array elements (plus the length of the array) and increments *fp* by that amount. Figure 7 shows the memory layout for our running example in Figure 1a. In this example, the two arrays represent DB and the struct represents a $CB(k = 2)$.

CertoraProver recognizes these allocation patterns with a lightweight pattern matching over the TAC code that updates the free pointer. If CertoraProver fails to classify an update of the free pointer according to one of the above patterns, the analysis pipeline returns the trivial result.

```
block0:
cond := ...
BRANCH cond block1 block2

block1:
p1 := MLOAD fp
JUMP block3

block2:
MSTORE fp ...
JUMP block3

block3
p2 := MLOAD fp
```

Fig. 10. Justification for the definite-write property. Since *fp* (`0x40`) is written in one branch of the conditional, the $\mathcal{NR}$ number at the second load to `p2` will be different from the load `p1`, despite these two values in fact being equal. `JUMP ID` is syntactic sugar for `BRANCH T ID ID`.

5.1.4 The Allocation Analysis. The analysis itself computes the *fp* reads that serve as allocation points, checks that they satisfy the definite-write property, and associates with each an abstract location. For every program point, CertoraProver's allocation analysis tracks *abstract allocations*, $\mathcal{W}$, as the analysis fact. We define $\mathcal{W} = \mathcal{L} \rightarrow A$, where $\mathcal{L}$ is set of all instruction labels and A is the set of all abstract locations. While the definite-write property means that the $\mathcal{NR}$ of an *fp* read by itself would be a sound choice for an abstract location, A, this would be imprecise, as it can conflate logically distinct allocations that have the same $\mathcal{NR}$ (e.g., two distinct allocations in different branches of a conditional). We therefore use the pair of $\mathcal{NR}$, and the number of the definite-write ($\mathcal{NW}$) together as part of A. As Section 5.1.3 described, the allocation analysis collects allocation patterns. This pattern information ($\mathcal{GR}$) is also part of A. This choice of including $\mathcal{GR}$ in A gives rudimentary type information for memory pointers; based on the pattern of writes identified in the first phase of this analysis, we can deduce whether the pointer is for an array (a dynamically-sized block or DB), or a struct (a block of k words, or $CB(k)$). Therefore, an abstract location is defined as: $A = \mathcal{NR} \times \mathcal{NW} \times \mathcal{GR}$, where $\mathcal{GR} = \{CB(k), DB\}$.

Figure 11 shows the core flow functions for this allocation analysis. It states that the abstract domain *Alloc* is a sum-type: it is (1) either $\top$ (the trivial result), (2) $(\mathfrak{P}(V) \times \mathcal{NR} \times \mathfrak{P}(\mathcal{L}) \times \mathcal{W})$ (used during an allocation window), or (3) outside of allocation windows, a mapping of *fp* reads to inferred abstract locations $\mathcal{W}$. The join on *Alloc* is strict: if there are different results for l in $\mathcal{W}$, or different sets of free-pointer derived variables, the result is simply $\top$. We explain these in detail below.

The core analysis occurs during *allocation windows*. During these sections of the program, the analysis uses an elaborated allocation window state. This state consists of the set of variables known to be derived from the current value of the free pointer ($\mathfrak{P}(V)$), the reaching free-pointer write number of the current window $\mathcal{NR}$, the free pointer read labels that have been observed so far during the allocation window ($\mathfrak{P}(\mathcal{L})$), and the accumulated results of already classified free pointer reads ($\mathcal{W}$). The analysis detects that an allocation window has "opened" when it encounters a free pointer and it is not already tracking a window and has not yet failed with the trivial result (the first branch of the *MLOAD(fp, dest)* case). The analysis then records the result of the free pointer read in

[5] Our implementation also supports other allocation patterns like byte arrays and strings which we omit for space reasons.

$$\mathit{flow} : \mathit{instr} \to \mathit{Alloc} \to \mathit{Alloc} \qquad \mathit{Alloc} : \top + (\mathfrak{P}(V) \times \mathcal{NR} \times \mathfrak{P}(A) \times \mathcal{W}) + \mathcal{W} \qquad \mathcal{W} : \mathcal{L} \mapsto A$$

$$\mathit{flow}[\![\mathit{MLOAD}(\mathit{fp}, \mathit{dest}); l']\!](\mathit{Alloc}) = \begin{cases} (\{\mathit{dest}\}, \mathcal{NR}_l, \{l\}, \mathcal{W}) & \mathit{Alloc} \equiv \mathcal{W} \\ (\mathfrak{p}_V \cup \{\mathit{dest}\}, n, r \cup \{l\}, \mathcal{W}) & \mathit{Alloc} \equiv (\mathfrak{p}_V, n, r, \mathcal{W}) \land n = \mathcal{NR}_l \\ \top & o.w. \end{cases}$$

$$\mathit{flow}\,[\![\mathit{var} \leftarrow \mathit{exp}; l']\!](\mathit{Alloc}) = \begin{cases} \mathit{Alloc} & \mathit{Alloc} \equiv \mathcal{W} \lor \mathit{Alloc} \equiv \top \\ (\mathfrak{p}_V \setminus \mathit{var}, n, r, \mathcal{W}) & \mathit{Alloc} \equiv (\mathfrak{p}_V, n, r, \mathcal{W}) \land n = \mathcal{NR}_l \land \\ & \mathcal{FV}(\mathit{exp}) \cap \mathfrak{p}_V = \emptyset \\ (\mathfrak{p}_V \cup \{\mathit{var}\}, n, r, \mathcal{W}) & \mathit{Alloc} \equiv (\mathfrak{p}_V, n, r, \mathcal{W}) \land n = \mathcal{NR}_l \land \\ & \forall v \in (\mathcal{FV}(\mathit{exp}) \cap \mathfrak{p}_V).\mathit{exp} > v \\ \top & o.w. \end{cases}$$

$$\mathit{flow}\,[\![\mathit{MSTORE}(\mathit{fp}, \mathit{val});\ l']\!](\mathit{Alloc}) = \begin{cases} \mathcal{W} \cup \{l \mapsto (n, \mathcal{NW}_l, \mathcal{GR}) | l \in r\} & \mathit{Alloc} \equiv (\mathfrak{p}_V, n, r, \mathcal{W}) \land \mathcal{NR}_l = n \land \\ & \mathcal{GR} = \mathit{classify}(l) \land \mathit{val} \in \mathfrak{p}_V \\ \top & o.w. \end{cases}$$

Fig. 11. Selected flow functions, *flow*$[\![\cdot]\!]$, for CertoraProver's allocation analysis. The dataflow fact here is the allocation information, $\mathcal{W}$. *exp* represents expressions from TAC. l is the label of the current statement.

the singleton set $\{\mathit{dest}\}$, seeds the set of (as yet) unclassified free pointer with the label of the read, and records the $\mathcal{NR}$ of the current read. If there is already an open window (the second case), the allocation analysis checks that the $\mathcal{NR}$ number has not changed since the window opened (the $\mathcal{NR}_l = n$ comparison in the second). If it has not, then the sets of free pointer derived variables and free pointer reads are updated accordingly. Otherwise, the analysis immediately returns the trivial result, e.g., if $\mathcal{NR}_l$ is different from the $\mathcal{NR}$ number recorded in the allocation window state.

Now we discuss the variable assignment case in Figure 11 ($\mathit{var} \leftarrow \mathit{exp}$). Outside of allocation windows, the allocation analysis does not consider variable assignments. However, it is crucial that the analysis accurately tracks the set of all variables that must be derived from the free pointer. If the RHS of the assignment does not mention any variables that are definitely derived from the free pointer (shown by the formula $\mathcal{FV}(\mathit{exp}) \cap \mathfrak{p}_V = \emptyset$ in Figure 11), the analysis effectively ignores the LHS of the assignment (killing it from the set if it was present). If the expression (*exp*) mentions any variables derived from the current incarnation of the free pointer and must always return a value larger than all such variables ($\forall v \in (\mathcal{FV}(\mathit{exp}) \cap \mathfrak{p}_V).\mathit{exp} > v$) then the left hand side of the assignment is added to the set of free pointer derived variables. In any other case, e.g., the expression *decreases* a free pointer derived value, or the $\mathcal{NR}$ number is different from the current window, the analysis conservatively gives up.

```
r1 = MLOAD fp
r2 = r1 + 0x20 // 0x20 = 32 in decimal
r3 = l * 0x20
r4 = r2 + r3
MSTORE fp r4
```

Fig. 12. Allocation of *DB* is spread across multiple TAC instructions (generated from EVM bytecode, Section 3.2). Here, we show memory allocation for an array which updates *fp* as explained in Section 3.2: `fp = fp + 0x20 + l * 0x20`.

The final piece of the allocation analysis occurs on an update of the free pointer (the *MSTORE(fp, val)* case). If the $\mathcal{NR}$ of the write location[6] matches that of the current window, then the analysis has reached a free pointer write without any intervening change of the $\mathcal{NR}$ number. Thus, all of the reads in r must have observed the same value of the free pointer. We also ensure that the value being written into the free pointer is itself derived from the current value of the free pointer ($\mathit{val} \in \mathfrak{p}_V$); this check guarantees that the free pointer must never decrease. In addition, the *classify*

[6] The $\mathcal{NR}$ value change for a free pointer write applies to the dominated *successors* of the write, not the write itself.

oracle classifies the “sort” of the allocation according to the patterns discussed above. With the write of the free pointer, the allocation window is closed: the collected free pointer reads are added to the set of classified location $\mathcal{W}$ with the corresponding location $L = (n, \mathcal{NW}_l, \mathcal{GR})$. The remaining flow functions are in Appendix A.

Referring back to the TAC program corresponding to Figure 1a shown in Figure 1b, the allocation analysis identifies three distinct allocations: the allocation for the struct `s` on line 1 as a $CB(k = 2)$ and the DB two allocations for `s.k1`and `s.k2` and on line 9 and line 17. These correspond the three allocations that happen from line 6-line 8 in Figure 1a.

5.2 Initialization Analysis

The goal of the initialization analysis is to find locations in the program where it is safe to read freshly allocated memory. After each object allocation, the newly reserved block of memory holds arbitrary data. The Solidity compiler follows all allocations with initialization code that fills the new block with default values of the appropriate type. From the perspective of the source program (in Solidity), the intermediate, uninitialized object is never visible: the allocation and initialization occur atomically. However, at the bytecode level these freshly allocated objects exist in an uninitialized state for some portion of program execution. We call the portion of the code where an object is allocated but not fully initialized its “initialization window”.

It would be sound to treat all freshly allocated abstract objects in the pointer analysis as being initialized with completely non-deterministic values. However, unless the the pointer analysis can prove all writes in the Solidity-generated initialization code are strong updates which definitively kill these non-deterministic values, the analysis suffers a major precision loss; any read from memory must be treated as returning an arbitrary value.

Instead, CertoraProver has a special treatment for initializing objects (and pointers to them). Following the conceptual model of Solidity, within their initialization window, objects are not yet “live” and do not support all operations that fully initialized objects do. In particular, CertoraProver forbids reads from initializing objects and their escape into the heap; upon detecting such behavior, the analysis pipeline fails with a conservative result. However, this restriction is satisfied for all code generated by the Solidity compiler.

The initialization analysis infers the initialization windows for each allocation. In particular, for each $(l \mapsto \ell) \in \mathcal{W}$, this analysis detects the point in the program at which all portions of the freshly allocated object have been written (relying on ℓ’s $\mathcal{GR}$). Formally, the result of the initialization analysis, is a mapping from $l \mapsto l_{ends}$ that indicates the point in the TAC CFG where initialization of ℓ completes. If the analysis cannot detect that a unique point exists for any ℓ, the entire analysis pipeline fails with a trivial result.

Referring back to our running example, for `s` allocated on line 6 in Figure 1a, the initialization analysis over the TAC shown in Figure 1b will return the mapping $\{ 1 \mapsto 7 \}$ (for demonstration, we use the line numbers in Figure 1b as labels).

If the initialization analysis succeeds, and given the restrictions on initializing object usage described above, it is unnecessary to model the nondeterministic contents of freshly allocated objects; all such programs are guaranteed to never observe these “junk” values, obviating including them in our abstract state. The initialization window information is communicated to the pointer analysis (see Section 5.3), which checks the usage restrictions in the initialization window.

5.3 Points-to Analysis

The final component of CertoraProver’s memory analysis is a pointer analysis. The pointer analysis is implemented as a reduced product between two abstract interpretations: a pointer semantics and a numeric bounds analysis. In this reduced product setting, the analyses exchange information

$$\widehat{\mathcal{P}} ::= (V \mapsto \widehat{v_p}) \times \widehat{\mathcal{H}} \qquad \widehat{\mathcal{H}} ::= A \mapsto (\mathit{Bool} \times O_{\widehat{m}}) \times \iota(A) \mapsto O_{\widetilde{m}} \qquad O_{\mu} ::= \widehat{\mathit{Arr}}(h_{\perp}) \mid \widehat{\mathit{Str}}(\mu)$$

$$\widehat{m} ::= N \mapsto h \quad \widetilde{m} ::= N \mapsto h_{\perp} \qquad h ::= \mathcal{B}(\mathfrak{P}(A), N) \mid \mathit{ArrPtr}(\mathfrak{P}(A)) \mid \mathtt{INT} \qquad \widetilde{\mathcal{I}} ::= \widetilde{\mathit{Arr}}(\iota(\ell), \mathit{Bool}) \mid \widetilde{\mathit{Str}}(\iota(\ell), N)$$

$$\widehat{v_p} ::= h \mid \widetilde{\mathcal{I}} \mid \mathcal{F}(\mathfrak{P}(A), N_{+}, N) \mid \mathcal{E}(\mathfrak{P}(A), \mathit{Bool}, V_{\top})$$

$$\begin{aligned}
\mathcal{F}(\mathfrak{p}_A, k, n) \sqcup \mathcal{F}(\mathfrak{p}'_A, k, n) &= \mathcal{F}(\mathfrak{p}_A \cup \mathfrak{p}'_A, k, n) \\
\mathcal{F}(\_, k, n) \sqcup \mathcal{F}(\_, k', n') &= \mathtt{INT} \\
\mathcal{B}(\mathfrak{p}_A, n) \sqcup \mathcal{B}(\mathfrak{p}'_A, n) &= \mathcal{B}(\mathfrak{p}_A \cup \mathfrak{p}'_A, n) \\
\mathcal{B}(\_, n) \sqcup \mathcal{B}(\_, n') &= \mathtt{INT} \\
\mathcal{E}(\mathfrak{p}_A, b, v) \sqcup \mathcal{E}(\mathfrak{p}'_A, b, v') &= \mathcal{E}(\mathfrak{p}_A \cup \mathfrak{p}'_A, b, v \sqcup v') \\
\mathcal{E}(\_, b, \_) \sqcup \mathcal{E}(\_, b', \_) &= \mathtt{INT}
\end{aligned}
\qquad
\begin{aligned}
\mathit{ArrPtr}(\mathfrak{p}_A) \sqcup \mathit{ArrPtr}(\mathfrak{p}_A) &= \mathit{ArrPtr}(\mathfrak{p}_A \cup \mathfrak{p}_A) \\
\widetilde{\mathit{Arr}}(\iota(\ell), b) \sqcup \widetilde{\mathit{Arr}}(\iota(\ell), b) &= \widetilde{\mathit{Arr}}(\iota(\ell), b) \\
\widetilde{\mathit{Arr}}(\iota(\ell'), b) \sqcup \widetilde{\mathit{Arr}}(\iota(\ell'), b') &= \mathtt{INT} \\
\widetilde{\mathit{Str}}(\iota(\ell), k) \sqcup \widetilde{\mathit{Str}}(\iota(\ell), k) &= \widetilde{\mathit{Str}}(\iota(\ell), k) \\
\widetilde{\mathit{Str}}(\iota(\ell), k) \sqcup \widetilde{\mathit{Str}}(\iota(\ell'), k') &= \mathtt{INT}
\end{aligned}$$

Fig. 13. Pointer abstract domain and selected least upper bound ($\sqcup$) definitions.

to gain additional precision. For example, the numeric analysis tracks basic path conditions that can establish an array variable definitely has non-zero length, or that a certain pointer is within bounds. We now describe core components of the analysis.

Combined Domain. The abstract domain $\widehat{\mathcal{S}}$ is a product of the pointer state $\widehat{\mathcal{P}}$ and numeric state $\widehat{\mathcal{N}}$: $\widehat{\mathcal{S}} ::= \widehat{\mathcal{P}} \times \widehat{\mathcal{N}}$. Below we describe both components of this domain.

5.3.1 *Pointer State, $\widehat{\mathcal{P}}$.*

The pointer state itself (Figure 13) is a product of the abstract pointer store (a partial mapping from program variables V to store values $\widehat{v_p}$) and an abstract heap $\widehat{\mathcal{H}}$. For notational convenience, we will abbreviate $\widehat{s}(v_1)$ by $\widehat{p}(v_1)$ where $\widehat{p}$ is some pointer state $\langle \widehat{s}, \widehat{h} \rangle$.

Abstract Pointer Store ($V \mapsto \widehat{v_p}$). The abstract pointer store maps each variable to heap-storable values h, pointers to objects being initialized $\widetilde{\mathcal{I}}$, pointers to struct *fields* $\mathcal{F}(\mathfrak{p}_A, k, k')$, or the intermediate result of pointer arithmetic $\mathcal{E}(\mathfrak{p}_A, b, v_{\top})$. We explain each of these below.

h is a subset of values that are safe to store into the heap. These "heap" values can be pointers to the beginning of a constant sized block of size k ($\mathcal{B}(\mathfrak{p}_A, k)$), pointers to the beginning of an array ($\mathit{ArrPtr}(\mathfrak{p}_A)$), or integers INT. $\mathfrak{p}_A$ represents a set of abstract locations; specifically all abstract locations that a field or value *may* point-to. It is an invariant that for each $\ell \in \mathfrak{p}_A$, for some $\mathcal{B}(\mathfrak{p}_A, k)$, ℓ will map to an abstract struct object of size k in all valid abstract heaps, and similarly for $\mathit{ArrPtr}(\mathfrak{p}_A)$. Pointers to initializing objects are modeled in the store by initialization pointers $\widetilde{\mathcal{I}}$. An array initialization pointer is represented by $\widetilde{\mathit{Arr}}(\iota(\ell), b)$. If b is $\perp$, then the array initialization pointer points to the beginning of the array block (recall from Section 3.2 that this is the length field), otherwise it points to within the data segment. An initialization pointer for a struct is represented with $\widetilde{\mathit{Str}}(\iota(\ell), k)$, where k is the field number within the struct pointed to by the pointer. Recall that the addresses being initialized $\iota(\ell)$ are *not* sets: all initialization pointers must point to exactly one value and hence admits strong updates. Initializing pointer objects are not heap values, thus the pointer analysis forbids storing partially initialized objects from "escaping" into the heap.

$\mathcal{F}(\mathfrak{p}_A, k, k')$ is a pointer to a field of a struct objects whose location is abstracted by the set $\mathfrak{p}_A$. k is the specific field number being pointed to, whereas k' is the static size of the "parent" struct object. Note that field pointers and base pointers with differences of referenced field or total size *cannot* be mixed: as shown in the join rule for such values (Figure 13), attempting such a combination (e.g., at a control-flow join) will effectively "kill" the pointer by turning it into an undistinguished INT.

The final components of the abstract values in the pointer state store are intermediate values that exist during pointer arithmetic generated by the Solidity compiler. This is represented by $\mathcal{E}(\mathfrak{p}_A, b, v_{\top})$. It is a pointer to the element portion of an array. $v_{\top}$ is the variable which holds a pointer to the beginning of the entire array block or $\top$ if no such variable could be deduced (e.g., due to precision loss from control-flow join). If b is $\top$ then the pointer is known to be the beginning of the element block (i.e., the location of element zero). This does *not* imply that the pointer is safe to read, if the underlying array is empty then reading from such a pointer would observe arbitrary data (the Solidity compiler bug mentioned in Section 7.2 was detected due to such an unsafe read). On the other hand if b is $\bot$, then the pointer does *not* necessarily point to the beginning of the element segment. However, it *is* an invariant that such an element pointer is safe to read.

Abstract Heap. The abstract heap summarizes the (possibly many) objects allocated at each abstract location. This abstraction represents the high-level heap (Ethereum memory) implemented via the bump allocator. Due to the initialization analysis (Section 5.2), we know that during the points-to analysis all objects are either fully initialized or in the process of being initialized. $\widehat{\mathcal{H}}$ is therefore a product of two (partial) maps. The first component $A \mapsto (Bool \times O_{\widehat{m}})$ models the state of *fully initialized* abstract objects. Each abstract address maps to a pair consisting of a summary flag b and an initialized abstract object $O_{\widehat{m}}$. The summary flag indicates whether or not this object is a "summary" object, i.e., $\top$ if it summarizes the state of multiple concrete objects, or $\bot$ if the abstract object summarizes exactly one object. Summary objects only admit weak updates; however, as non-summary objects represent the state of exactly one concrete instance they can support strong updates (see the discussion of memory semantics below). The second component of the abstract heap is a map from *initializing* addresses $\iota(A)$ to *initializing* objects $O_{\widetilde{m}}$. Note that the initializing addresses $\iota(A)$ are *not* sets: all mapped objects in the initializing heap correspond to exactly one concrete object, and hence admit strong updates.

Heap Objects. The definition of heap objects O_μ is parameterized by their initialization status: either partially initialized ($\widetilde{m}$) or fully initialized ($\widehat{m}$). Thus, each fully initialized object is either an array $\widehat{Arr}(h_\bot)$ or fully initialized struct $\widehat{Str}(\widehat{m})$. For simplicity, we model all elements of the array with a single summary field,[7] whereas for structs we use a partial mapping from sequential field indices $0, \ldots, k$ to abstract values $h_0, \ldots, h_k$; the size of k is fixed for each abstract location and is determined by the (constant) amount by which the free pointer is bumped. Note that unlike structs, we allow a special $\bot$ value as the value of the summary field to model empty arrays.

Partially initialized arrays use the same representation as their fully initialized counterparts, as an array with no information written into it yet can be represented with $\widehat{Arr}(\bot)$. The partially initialized object representation for structs is $\widehat{Str}(\widetilde{m})$, where $\widetilde{m}$ is defined similarly to $\widehat{m}$ but where fields can be mapped to $\bot$ (indicating they have not been initialized yet).

Memory Semantics. A select subset of the abstract semantics for handling memory operations is shown in Figure 14 and the full treatment is in Appendix B.

Our memory analysis ensures that a program's use of the heap is strongly typed—any cell in memory with a reference type (a struct or an array) holds a pointer that has been allocated via the bump allocator and has the corresponding shape. Specifically, at a write of a value h into the heap, our analysis compares the type of h with the expected type of the heap location, and aborts if the types are not compatible.

Thus, before introducing the semantics for memory writes, we first describe the coarse-grained typing information inferred by the analysis pipeline. The τ_h definition gives types for values that appear in the heap, extended with top and bottom elements and it is equipped with a least upper bound operator.[8] The inductively defined $\mathcal{T}_{\widehat{h}}$ relation yields the type of a heap value h in some heap $\widehat{h}$. With these definitions, we now describe the *write* function, which models a write of v_2 to the location pointed to by v_1 by transforming the combined abstract state according to the effects of the write. In the following, we will write $x[y \hookleftarrow z]$ to mean replacing the value of y with z in

[7] While CertoraProver uses this simplification, our VC generation is precise w.r.t. different indices.
[8] These "types" only loosely correspond to the types in Solidity: they lack struct field names, and the type names are lost.

$$\tau_h ::= \bot \mid \tau_h\ \mathit{array} \mid \mathit{int} \mid [0 \mapsto \tau_h^0, \dots, k \mapsto \tau_h^k] \mid \top$$

$$\mathcal{T}_{\widehat{h}}(h) = \begin{cases} \mathit{int} & h = \mathtt{INT} \\ \bigsqcup\{\mathcal{T}_{\widehat{h}}(h')\ \mathit{array} \mid \ell \in \mathfrak{p}_A \wedge \widehat{h}(\ell) = \langle b, (\widehat{Arr}(h')\rangle \wedge h' \neq \bot\} & h = \mathit{ArrPtr}(\mathfrak{p}_A) \\ \bigsqcup\{[0 \mapsto \mathcal{T}_{\widehat{h}}(h_0), \dots, k \mapsto \mathcal{T}_{\widehat{h}}(h_k)] \mid \ell \in \mathfrak{p}_A \wedge \widehat{h}(l) = \langle b, \widehat{Str}([0 \mapsto h_0, \dots, k \mapsto h_k])\rangle\} & h = \mathcal{B}(\mathfrak{p}_A, k) \end{cases}$$

$$\mathit{write}[\mathtt{MSTORE}\ \ v_1\ \ v_2](\langle\widehat{n}, \langle\widehat{s}, \widehat{h}\rangle\rangle) =$$

$$\begin{cases} \langle \widehat{n} \sqcap \{v_2 \mapsto \langle\top, \{\mathtt{LenOf}(v_1)\}\rangle\}, \langle\widehat{s}, \widehat{h}\rangle\rangle & \widehat{s}(v_1) = \widetilde{Arr}(\iota(\ell), \top) \wedge \widehat{s}(v_2) = \mathtt{INT} \quad (1) \\ \langle\widehat{n}, \langle\widehat{s}, \widehat{h}[\iota(\ell) \hookleftarrow \widehat{Str}(\widetilde{m}[k \hookleftarrow h])]\rangle\rangle & \widehat{s}(v_1) = \widetilde{Str}(\iota(\ell), k) \wedge \widehat{s}(v_2) = h \wedge \widehat{h}(\iota(\ell)) = \widehat{Str}(\widetilde{m}) \wedge \widetilde{m}(k) = \bot \quad (2) \\ \langle\widehat{n}, \langle\widehat{s}, \widehat{h}'\rangle\rangle & \widehat{s}(v_2) = h \wedge \big((\widehat{s}(v_1) = \mathcal{B}(\mathfrak{p}_A, n) \wedge k = 0) \vee \widehat{s}(v_1) = \mathcal{F}(\mathfrak{p}_A, k, n)\big) \quad (3) \\ & \wedge_{\ell \in \mathfrak{p}_A} \big(\widehat{h}(\ell) = \langle b, \widehat{Str}(\widehat{m})\rangle \wedge \mathcal{T}_{\widehat{h}}(\widehat{m}(k)) \sqcup \mathcal{T}_{\widehat{h}}(h) \neq \top\big) \wedge \\ & \mathit{strong} \Leftrightarrow (\mathfrak{p}_A = \{\ell\} \wedge \widehat{h}(\ell) = \langle\bot, \widehat{m}\rangle) \wedge \\ & \widehat{h}' = \widehat{h}\Big[\ell \hookleftarrow \Big\langle b, \widehat{m}\Big[k \hookleftarrow \begin{cases} h & \mathit{strong} \\ h \sqcup \widehat{m}[k] & o.w. \end{cases}\Big]\Big\rangle \Big| \ell \in \mathfrak{p}_A \wedge \widehat{h}(\ell) = \langle b, \widehat{m}\rangle\Big] \\ \dots & \dots \\ \lightning & o.w. \end{cases}$$

Fig. 14. Selected set of memory semantics. Recall from Section 5.3.1 that $\langle\widehat{s}, \widehat{h}\rangle$ is the pointer state, $\widehat{p}$.

the mapping x, and $x[y \hookleftarrow z|\phi(y)]$ means replacing all such y selected by the predicate ϕ. For convenience, we will use this notation on the abstract heap which is actually a product of maps, context will make clear which of the underlying maps is to be updated.

The first case (Equation 1) handles writes to an array initialization pointer. This specific case is for when the array initialization pointer v_1 points to the beginning of the array block (as indicated by the $\top$ in the second field of the $\widetilde{Arr}()$ constructor). According to Solidity's memory layout, v_1 points to the length field of the array, and thus the value being written v_2 must be an integer. The abstract heap $\widehat{h}$ is not updated, but the numeric state $\widehat{n}$ is refined to record that v_2 is now known to be the length of the array v_1.[9] The next case Equation 2 shows the case for a struct initialization pointer. If the value for field k (as indicated by k field of $\widetilde{S}$) has not yet been written ($\widetilde{m}(k) = \bot$), then the field k in the initializing struct is strongly updated to be h.

Equation 3 handles struct field updates. The type of field k for each abstract struct associated with all abstract locations in $\mathfrak{p}_A$ is checked for type compatibility with the value to be written. Struct writes support strong updates: if the set of abstract locations is singleton *and* its associated abstract struct object is not a summary object (the summary flag is $\bot$), then the write can be modeled with a strong update as indicated by the *strong* variable. Regardless of update type, the k field of the abstract struct objects for each location in $\mathfrak{p}_A$ are updated within the abstract heap to be either the least upper bound of the current field value and the new value ($\widehat{m}[k] \sqcup h$) or the new value h depending on *strong*. The remaining cases for safe writes into the heap are elided with "...", the full semantics can be found in Appendix B.

$\lightning$ covers any other attempted write: it indicates that the analysis immediately and conservatively fails, as it detected a potentially unsafe write or a violation of the strongly typed heap. The semantics of MLOAD commands, given by the *read* function can also be found in Appendix B.

Address Management. Finally, we sketch the process by which new abstract locations are added into the heap, and initializing objects are folded into the main heap at the close of their initialization window. Reads of the free pointer into a variable v update the pointer state according to the information computed by the allocation analysis. As all reads of the free pointer necessarily yield

[9] As written, the semantics technically allow multiple writes to an array pointer's length field, however the initialization analysis forbids such writes.

$$\widehat{\mathcal{N}} ::= V \mapsto \widehat{v_n} \qquad \widehat{v_n} ::= \mathsf{PTR} \mid ([lb, ub] \times \mathfrak{P}(Q)) \quad \langle i, q\rangle \sqcup \langle i', q'\rangle = \langle i \sqcup i', q \cap q'\rangle \quad \langle i, q\rangle \sqcup \mathsf{PTR} = \mathsf{PTR} \sqcup \langle i, q\rangle = \langle \top, \emptyset\rangle$$

$$Q ::= \mathtt{LenOf}(V) \mid \mathtt{SafeIndOf}(V) \mid \mathtt{SafeDataOffset}(V) \mid \mathtt{SafeElemOf}(V) \mid \ldots$$

$$plus[v_r := v_1 + v_2](\langle \widehat{p}, \widehat{n}\rangle) = \langle \widehat{p}[v_r \mapsto v_p], \widehat{n}[v_r \mapsto v_n]\rangle$$

$$\text{where} \langle v_p, v_n\rangle =$$

$$\begin{cases}
\langle \mathcal{E}(\mathfrak{p}_A, \top, v_1), \mathsf{PTR}\rangle & \widehat{p}(v_1) = ArrPtr(\mathfrak{p}_A) \wedge \widehat{n}(v_2) = \langle 32, q\rangle & (4)\\
\langle \mathcal{F}(\mathfrak{p}_A, k, m), \mathsf{PTR}\rangle & \widehat{p}(v_1) = \mathcal{B}(\mathfrak{p}_A, m) \wedge \widehat{n}(s_2) = \langle k, q\rangle \wedge k = 0 \; mod\; 32 \wedge k < m & (5)\\
\langle \mathcal{F}(\mathfrak{p}_A, k + k', m), \mathsf{PTR}\rangle & \widehat{p}(v_1) = \mathcal{F}(\mathfrak{p}_A, k, m) \wedge \widehat{n}(v_2) = \langle k', q\rangle \wedge m > k + k' = 0 \; (mod\; 32) & (6)\\
\langle \mathcal{E}(\mathfrak{p}_A, \bot, v_1), \mathsf{PTR}\rangle & l(v_1) = ArrPtr(\mathfrak{p}_A) \wedge \widehat{n}(v_2) = \langle i, q\rangle \wedge \mathtt{SafeDataOffset}(v_1) \in q & (7)\\
\langle \mathsf{INT}, \langle i_1 + 32, \{\mathtt{SafeDataOffset}(a_p)\}\rangle\rangle & \widehat{n}(v_1) = \langle i_1, q\rangle \wedge \widehat{n}(v_2) = \langle 32, q'\rangle \wedge \mathtt{SafeElemOf}(a_p) \in q & (8)\\
\langle \mathcal{E}(\mathfrak{p}_A, \bot, a_1), PTR\rangle & \widehat{p}(v_1) = \mathcal{E}(\mathfrak{p}_A, \top, a_1) \wedge \widehat{n}(v_2) = \langle i, q\rangle \wedge \mathtt{SafeElemOf}(a_1) \in q & (9)
\end{cases}$$

Fig. 15. Numeric abstract domain, and selected arithmetic abstract semantics for *plus*. Singleton intervals ($[k, k]$) are written simply as k.

pointers to an as yet uninitialized object, the abstract store $\widehat{s}$ is updated to bind v to an initialization pointer of the appropriate sort for the abstract location inferred for the read. In addition, if no binding for the initializing address is in the heap, it is added to the initialization component of the heap, setting all fields to $\bot$. At an initialization completion point for an abstract location ℓ, the analysis "folds" the (now fully initialized) object into the main heap. All initialization pointers that are known to point to the beginning of the (freshly initialized) block are promoted to array or struct pointers, and all other pointers for the initialized address as "junked" to become integers. The object associated with $\iota(\ell)$ is joined with the existing binding for ℓ (if it exists), and the binding for $\iota(\ell)$ is removed from the initialization space. Finally, if a binding existed for ℓ before the initialization was completed, the summary flag for the abstract object is set to $\top$, we must now have at least two concrete objects represented by the abstract object. The rest of the semantics are in Appendix B.

5.3.2 Numeric State, $\widehat{\mathcal{N}}$. The numeric domain (shown in Figure 15) only tracks store values, and is thus a (partial) map from variables V to numeric abstractions $\widehat{v_n}$. Each numeric abstraction is either an opaque (to the numeric analysis) pointer PTR, or composed of two pieces: an interval $[lb, ub]$ and a set of *qualifiers* Q. A numeric abstraction is represented as $\langle i, q\rangle$ in the formalism.

Qualifiers, Q. Our analysis uses an unsigned 256-bit integer representation for numbers, thus the largest element in our interval representation is $[0, 2^{256} - 1]$. Each qualifier is an atomic "fact" about the value to which it is attached. For example $\mathtt{MultipleOf}(k)$ represents that the value is not only within the bounds given by the interval, but that the value must be a multiple of k. Similarly, $\mathtt{LenOf}(v)$ indicates that the value represents the length of the array variable v. These qualifiers are a lightweight and efficient way to elaborate a simple interval domain with enough information to express the linear relationships and inequalities necessary to prove pointer safety without resorting heavyweight, fully relational domains like (sub)polyhedra. The join operation on the numeric domain can be performed pointwise, and the join of qualifiers is simply set intersection.

Arithmetic Semantics. Figure 15 displays a small selection of the *plus* semantics. For simplicity, the formalism is intentionally *not* commutative, however our actual implementation handles any order of arguments. The *plus* semantics are parameterized over the actual variable arguments v_1 and v_2 and takes the combined abstract analysis state as input. The output is the abstract state extended with a binding for the result of the addition v_r to the abstract pointer (v_p) and numeric (v_p) values. Equation 4 handles a case where exactly one word (32 bytes) is added to the start of the array pointer argument v_1. According to the data layout of the Solidity compiler, the result must point to the beginning of the data segment, as represented by $\mathcal{E}(\mathfrak{p}_A, \top, v_1)$; note that v_1 is recorded as the "source" array in the result.

The next two cases concern the pointer arithmetic for struct pointers; Equation 5 establishes that the non-pointer operand is a word-aligned constant k that is smaller than the static size of the struct block m, and produces a field pointer to that field. The second case Equation 6 handles addition to an extant field pointer.

The final three cases cover array element safety, and illustrate the cooperation between the analyses. Equation 7 covers the case where a variable v_2 qualified with $\texttt{SafeDataOffset}(v_1)$ is added to the array pointer v_1. The qualifier, `SafeDataOffset`, indicates that the v_2 is a safe data offset, that is, when it is added to the start of the array object in v_1 the result is a safe, in bounds element pointer to v_1. The pointer result of this addition operation (an in bounds element pointer) reflects this. The following case, Equation 8 demonstrates how the `SafeDataOffset` qualifier is generated; when the constant 32 (the size of an EVM word in bytes) is added to a variable qualified with $\texttt{SafeElemOf}(a_1)$. The `SafeElemOf` qualifier itself indicates that the addition of the qualified variable to the beginning of the *elements* of the array a_1 yields a valid, in bounds element pointer to a_1. Recall from Figure 1b that the elements of an array are stored offset by a word from the beginning an array to account for the length of the array. Thus, adding 32 to a variable qualified with `SafeElemOf` yields a value that can be added to the beginning of the entire array, as the bump by 32 will now skip the length of the array. Finally, Equation 9 shows an alternative to the previous scenario. If a variable v_2 qualified with $\texttt{SafeElemOf}(a_1)$ is added to the start location of the elements of a_1 (given by the $\top$ in $\mathcal{E}(\mathfrak{p}_A, \top, v_a)$) then this addition also yields an in bound element pointer.

We sketch how the remaining qualifiers are generated by the numeric semantics. $\texttt{LenOf}(v_1)$ is generated at a load from a pointer variable v_1 which is mapped to $\mathit{ArrPtr}(\mathfrak{p}_A)$ in the *pointer* domain $\widehat{p}$. Reading from the beginning of an array segment yields the length of that array, which is captured by the `LenOf` qualifier. $\texttt{SafeIndOf}(a)$ is generated for a variable that is proven to be a safe (i.e., in bounds) *logical* index for some array.

As discussed in Section 4, this fact is inferred by interpreting path conditions of the form $i < l$, where l is qualified with $\texttt{LenOf}(a)$. If this path condition is true, then i must be an in bounds index for the array a, and is qualified with $\texttt{SafeIndOf}(a)$. $\texttt{SafeElemOf}(a_1)$ is generated when a variable qualified with $\texttt{SafeIndOf}(a_1)$ is multiplied by 32. This multiplication transforms the *logical* index into an element offset, that is, the offset within the element portion of the array object. The rest of the arithmetic abstract semantics are in Appendix B.

6 Memory Splitting Transformation

If the pointer analyses complete successfully, then we can soundly apply our memory splitting transformation. Recall, our goal is to replace the naive memory model (a single monolithic array) by several *disjoint* arrays (a "partitioned memory-model" in the language of Wang et al. [98]).

For this, we use the abstract state computed by the pointer analysis to determine disjoint regions of memory. Informally, each TAC command that accesses memory is associated with a set of *nodes*, each representing a range of memory addresses. We build an equivalence relation on the set of nodes accessed by a program: if a command accesses two nodes n_1 and n_2, then we say that n_1 and n_2 belong to the same equivalence class. Each equivalence class represents a region of memory *disjoint* from any other equivalence class, and we partition memory into an array per class. To denote regions of memory, we introduce the concept of a *field node*:

$$\mathsf{Node} ::= \mathsf{LengthNode}(A) \mid \mathsf{ElementNode}(A) \mid \mathsf{StructNode}(A, N)$$

A $\mathsf{LengthNode}(\ell)$ denotes the addresses corresponding to the length field of the arrays allocated at the addresses summarized by the *abstract location* ℓ. Likewise $\mathsf{ElementNode}(\ell)$ denotes the addresses of the *elements* of the arrays at locations ℓ. Finally, $\mathsf{StructNode}(\ell, k)$ denotes the addresses of field k of the structs stored at the locations in ℓ.

Let pointerVal(l, p) return the pointer abstraction for p (*i.e.,* , the value of p in the $\widehat{\mathcal{P}}$ component as shown in Figure 13) at program point l. The function nodes$(\widehat{p})$ shows how to extract the field nodes from an abstract pointer value. We lift this to a function on labeled program points: nodes$(l : i)$.

$$\mathrm{nodes}(\widehat{p}) \quad = \quad \begin{cases} \{\mathrm{LengthNode}(\ell) \mid \ell \in \mathfrak{p}_A\} & \widehat{p} \equiv \mathit{ArrPtr}(\mathfrak{p}_A) \vee (\widehat{p} \equiv \widetilde{\mathit{Arr}}(\iota(\ell), \bot) \wedge \mathfrak{p}_A \equiv \{\ell\}) \\ \{\mathrm{ElementNode}(\ell) \mid \ell \in \mathfrak{p}_A\} & \widehat{p} \equiv \mathcal{E}(\mathfrak{p}_A, b, v) \vee (\widehat{p} \equiv \widetilde{\mathit{Arr}}(\iota(\ell), \top) \wedge \mathfrak{p}_A \equiv \{\ell\}) \\ \{\mathrm{StructNode}(\ell, 0) \mid \ell \in \mathfrak{p}_A\} & \widehat{p} \equiv \mathcal{B}(\mathfrak{p}_A, \_) \\ \{\mathrm{StructNode}(\ell, k) \mid \ell \in \mathfrak{p}_A\} & \widehat{p} \equiv \mathcal{F}(\mathfrak{p}_A, k, \_) \vee (\widehat{p} \equiv \widetilde{\mathit{Str}}(\iota(\ell), k) \wedge \mathfrak{p}_A \equiv \{\ell\}) \end{cases}$$

$$\mathrm{nodes}(l : i) \quad = \quad \begin{cases} \mathrm{nodes}(\mathrm{pointerVal}(l, p)) & i \equiv M[p] \\ \mathrm{nodes}(\mathrm{pointerVal}(l, p)) & i \equiv M \leftarrow M[p := v] \\ \emptyset & \mathit{otherwise} \end{cases}$$

Next, we define $\mathit{merge}_{\mathit{alias}}$:

$$(n_1, n_2) \in \mathit{merge}_{\mathit{alias}} \leftrightarrow \exists l : i \in P.\ \{n_1, n_2\} \subseteq \mathit{nodes}(l : i)$$

and let $\mathit{merge}^*_{\mathit{alias}}$ be its (reflexive, symmetric) transitive closure.

Finally, we let R_{alias} be a function that assigns each labeled command $l : i$ to $M_{\mathfrak{e}}$, where $\mathfrak{e}$ identifies the (single) equivalence class of $\mathit{merge}^*_{\mathit{alias}}$ containing nodes$(l : i)$ (there is a single class by construction since all pairs of nodes in nodes$(l : i)$ are equated). R_{alias} is then used as follows to uniquely name each memory partition; in the following R_l abbreviates $R_{\mathit{alias}}(l : i)$:

$$\mathrm{SplitMemory}(R_{\mathit{alias}}, l : i) \quad = \quad \begin{cases} l : R_l[p] & i \equiv M[p] \\ l : R_l \leftarrow R_l[p := v] & i \equiv M \leftarrow M[p := v] \\ l : i & \mathit{otherwise} \end{cases}$$

7 Evaluation

We implemented our analysis and transformation as part of CertoraProver in approximately 20K lines of Kotlin. CertoraProver (and our additions to it) works on EVM bytecode generated by all versions of the Solidity compiler starting from 0.4.24 and above.[10] Documentation about CertoraProver can be found here: https://docs.certora.com/en/latest/index.html.

The goal of this section is to provide evidence **E1** and **E2** that support the claims **C1** and **C2** presented in Section 1. To that end, we are interested in answering the following research questions.

E1 How does memory splitting affect the performance of SMT-based functional correctness verification of Solidity smart contracts (Section 7.1)?

E2 Can CertoraProver help uncover compiler bugs (Section 7.2)?

7.1 Effect of Memory Splitting on the Performance of SMT-Based Verification

We are interested in validating the claim that the memory splitting rewrite we introduced in Section 6 that is guided by the novel memory analysis in Section 5 helps in speeding up formal verification of real-world smart contracts. Below we first describe the experiment, then we discuss how we selected the benchmarks to evaluate on, and finally report the results. We ran these experiments on an EC2 server running 64-bit Ubuntu, with 72 GB memory running two containers each with 35 GB memory. Each container ran a single verification task at a time. CertoraProver uses the portfolio method [102] with a variety of SMT solvers including Z3 [44], CVC4 [30], CVC5 [28], and Yices [46]. A timeout means that none of the solvers generated a result in the given time.

Experimental setup. To evaluate the effectiveness of the memory splitting transformation, we ran CertoraProver in two modes. First, we disabled the memory splitting transformation altogether and ran verification. Then we turned on the memory splitting transformation and reran the same

[10] The most recently released version of Solidity is 0.8.25.

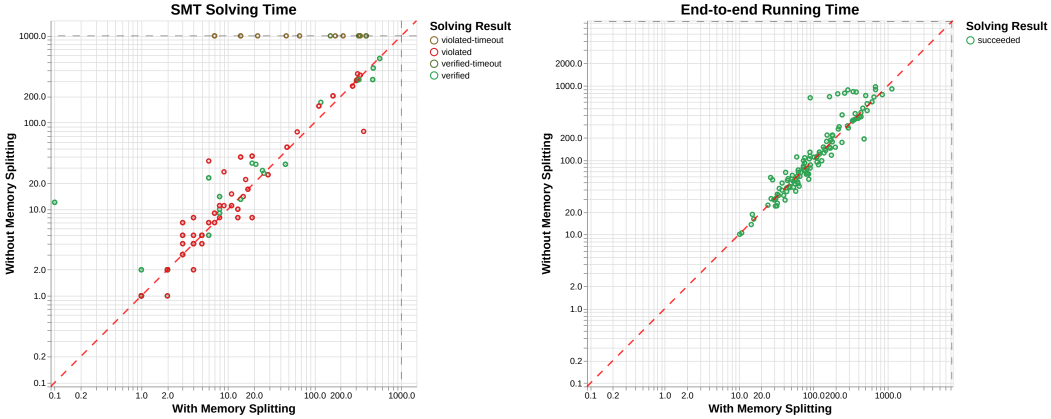


Fig. 16. (Left) Comparing the verification (SMT solving) time with and without the memory splitting rewrite enabled. (Right) End-to-end running time of CertoraProver with and without the memory splitting transformation sped up verification in many cases and in most cases had negligible effect on the overall running time of CertoraProver. For both figures, points above the $x = y$ line represent benchmarks for which memory splitting sped up SMT solving and end-to-end running time of CertoraProver. All times are in seconds and shown in log scale to better capture the wide range of running times across the verification tasks.

verification tasks. We then compared the running times of the SMT solvers and also the end-to-end running time of CertoraProver in both modes. To mitigate variance in the results, we made sure that the SMT solvers were run with a fixed seed. We additionally ran the entire experiment 3 times to make sure that the results were similar each time.

Benchmark selection. CertoraProver is an industrial formal verification tool with many users who are primarily developers of smart contract protocols. Users of CertoraProver include top "DeFi" (decentralized finance) protocols[11] like Aave (V2, V3, Gho) [6], Lido [13], Morpho [12], GMX [11], Euler [14], Maker [8], Silo [15], Uniswap V4 [17], Safe [16], Balancer [7], Curve [10], etc. many of which are open-source. The results of CertoraProver are only meaningful if there are high-level specifications (recall from Section 2 that we call them "rules") for the program, otherwise there is no property for CertoraProver to verify. Therefore, we evaluate our claim **C1** on real-world smart contracts that have specifications. Table 1 shows the specific list of smart contracts for which we ran the evaluation. Users run CertoraProver as a cloud service allowing us to analyze the performance of the tool on new specifications. To evaluate the effect of memory splitting on the SMT solving time, we looked at verification tasks run by customers starting at the beginning of 2024.

To ensure that we selected non-trivial verification tasks, we filtered out rules that had an end-to-end CertoraProver running time of less than 15 seconds at the time of sampling (this is the time to run the entire pipeline shown in Figure 2 and includes the SMT solving time). Users of CertoraProver run the same rule many times on the cloud (often as part of some continuous integration), so we made sure not to select the same rule more than once. We ended up with 229 unique verification tasks, where a verification task corresponds to checking a single rule for a single Solidity method. The dataset does contain multiple unique "rules" for some contracts.

Results. Figure 16 and Table 2 show the results of our evaluation. In Table 2 we show that memory splitting speeds up SMT-solving time by **2.03**× on average. Figure 16 (Left) shows the SMT solving time in both modes: with and without memory splitting. Red circles show benchmarks for which both modes found a violation of the specification (or rule). Green circles show those which were verified by both. For both these cases, we see that most of the dots are above the $x = y$

[11] https://dappradar.com/rankings/defi

Table 1. A summary of real-world smart contracts for which we evaluated the effect of memory splitting on verification. We selected these by filtering CertoraProver runs on the cloud by external users of CertoraProver. Crucially each of these contracts have multiple associated specifications (rules) that the contract developers are interested to verify. Here we also report on the number of memory partitions CertoraProver generated for each of these contracts.

Contracts	# Partitions
AccessManager	220
ActivePool	294
ATokenWithDelegation	439
MintOperationFixture	18
MintPoolArrayFixture	22
CoinFlip	344
CreateRental	432
CreateRental	564
CrossChainControllerWithEmergencyMode	689
CrossChainForwarder	371
CrossChainReceiver	330
Curves	273
CurvesERC20	267
Custodian	889
EigenPod	1007
ERC20ClubFactoryEth	246
ERC721	155
ERC721M	316
EulerSwap	230
EzEthToken	430
FeeFlowController	359
GhoFlashMinter	320
GhoToken	202
Governance	926
GovernancePowerStrategy	1082
Gsm	342
InterestRate	128
IonPool	552
IonPool	558
IonPoolStorage	899
JUSD	586
LendingPool	492
LeverageModule	1104
Liquidation	548
LMPStrategy	1822
MerkleDistributorModuleERC20	311
MerkleDistributorModuleERC721	362
MetaMorpho	553
MiniMeToken	171
Morpho	157
MorphoInternalAccess	358
MultichainGovernor	712
Obelisk	425
OperatorsRegistryV1	494
OrderBookHandle	430
PayloadsController	431
PirexEth	737
Pools	977
RedemptionNFT	1119
RegistryCoordinator	996
RNGSender	205
RNGSender	207
Safe	121
SafeTokenLock	135
ShareCollateralToken	2491
ShareDebtToken	2483
Silo	2789
Silo0	2873
Silo	2798
SmartVaultV3	837
SolverVaultToken	258
StakedAaveV3	917
StakeManager	146
StakeToken	338
StakeupToken	1129
StakeVault	183
Starport	298
Starport_19_20	569
Stop	245
Storage	540
StrategyManager	1091
StTBY	1134
TermAuction	349
TermRepoServicer	1227
Tranche	437
TruflationToken	528
USDS	520
VotingEscrowTruf	528
VotingMachine	801
VotingMachineTriple	812
WETHRebasing	232

line, which indicates that memory splitting was able to speed up SMT solving time. We also show `verified-timeout` and `violated-timeout`, which represent benchmarks for which SMT solvers timed out without memory splitting but were successful at verifying (or finding counterexamples) with splitting enabled. In total, there were **16** such cases. In addition to the 16 timeouts that memory splitting mitigated, the largest speed up in SMT solving we observed was **120**× where without memory splitting it took **12s** but with memory splitting it only took **0.1s**. The largest slow down in SMT solving with memory splitting that we witnessed is **4.65**× where without memory splitting it took 79s but with memory splitting it took 368s.

The right half of Figure 16 shows the end-to-end running time of CertoraProver in both modes. Each green dot corresponds to a successful end-to-end run of CertoraProver. Points above the $x = y$ line represent examples where memory splitting sped up end-to-end runs of CertoraProver. Table 2 summarizes this data: memory splitting sped up CertoraProver by 1.19× on average. While the end-to-end running time improvements may not seem significant at first, the cumulative effect is large in the context of a tool like CertoraProver where users run verification repeatedly on the cloud as continuous integration. Most of the rules are often run on many methods and checked every time there is a change in the contract code.

Number of partitions and failures. Table 1 shows the number of memory partitions that were generated for each smart contract. We also measured the total number of functions for which

Table 2. Mean end-to-end running time and SMT solving time (in seconds) of CertoraProver with and without the memory splitting rewrite over all the tasks.

Mode	Mean end-to-end time	Mean SMT time
w/o Rewrite	205.56s	111.47s
w Rewrite	172.25s	54.93s

Table 3. A summary of 5 compiler bugs found as a result of CertoraProver's pointer analysis. "Version" refers to the Solidity compiler version. We discuss one of the bugs in this section and the rest can be found in Appendix C.

Bug description	Version	Ack	Fixed
Storage Corruption	0.7.3	Yes	Yes [3]
Memory Isolation Violation	<= 0.8.3	Yes	Yes [4]
Non-deterministic Transaction	< 0.8.0	Yes	Yes [1]
Incorrect Calldata Validation	< 0.8.13	Yes	Yes [5]
Memory Corruption	< 0.6.5	Yes	Yes [2]

memory splitting failed. Over the 229 benchmarks, we had a total of 17844 functions (across all the smart contracts in Table 1). Out of these, CertoraProver was unable to split memory for 2.7% (488) of the functions. Note that even if splitting fails for some functions but succeeds for others, that may still be useful. There were no benchmarks for which splitting failed entirely for all functions. The maximum failure rate we observed in any given benchmark is 2.4%.

The practical implication of these failures for CertoraProver is how they affect the downstream memory splitting transformation which consumes the results: false positives means that the memory splitter will not be as effective since it will not be able to treat various memory regions as disjoint. If no memory splits succeed, then the effect on the SMT solvers will be as if we ran without this feature. For complex programs where memory splitting would otherwise benefit, this could cause slower verification times.

```
// ghost variable whose update is not specified in this snippet
ghost bool settleLoan_hasBeenCalled;
rule onlyBorrowerCanRepayALoan() {
  Custodian.Command cmd;
  require(!settleLoan_hasBeenCalled);
  require(cmd.action == Custodian.Actions.Repayment);
  address fulfiller;
  // generateOrder invokes settleLoan under some conditions.
  generateOrder(fulfiller, cmd);
  assert settleLoan_hasBeenCalled =>
    (fulfiller == getBorrower(cmd.loan) OR
     fulfiller == getApproved(getLoanId(cmd.loan)));
}
```

Fig. 17. Simplified example of a rule written in CertoraProver's specification language for which SMT solving timed out without memory splitting but succeeded to find a counterexample in less than a minute when enabled.

Representative Example. To give readers a sense of the kinds of properties CertoraProver is used for verifying, we highlight one example in Figure 17 from the above 229 tasks for which the SMT solver timed out without memory splitting but found a counterexample in 47s when the transformation was enabled. While the details of the smart contract's code and the specification language are beyond the scope of this paper, at a high-level the property states that only a borrower (`fulfiller`) of a loan or one approved for a loan should be able to repay it.

In summary, we conclude that memory splitting is *effective* for resolving timeouts in real-world verification tasks and significantly reduces SMT solving time without adding significant overhead.

7.2 Bugs Found in Solidity Compilers

CertoraProver has led us to detect 5 bugs in various versions of the Solidity compiler. We describe one of the compiler bugs here. The remaining four bugs are described in Appendix C. Table 3 summarizes the bugs. All bugs have been acknowledged by the Solidity team and fixed.

Table 4. Comparing CertoraProver to Halmos. Halmos offers limited expressivity in the specifications shown by ✗$_{noSpec}$ but in the case where it does support expressing the property, Halmos is still not able to verify it because they do not have a way to reason about memory accesses.

Contract (Spec)	CertoraProver Succeeded	Halmos Succeeded	CertoraProver Time	Halmos Time
ERC20 (check_transfer)	✓	✓	22.3s	29.37s
ERC20 (check_transferFrom)	✓	✓	22.2s	25.86s
ERC20 (check_NoBackDoor)	✓	✓	32.9s	18.81s
ERC20 (check_transferReverts)	✓	✗$_{noSpec}$	24.1s	N/A
ERC20 (check_checkTransferDoesntRevert)	✓	✗$_{noSpec}$	18.28s	N/A
ERC20 (check_sumOfBalancesEqTotalSupply)	✓	✗$_{noSpec}$	23.5s	N/A
ERC20 (check_sumOfBalancesGeqAnyBalance)	✓	✗$_{noSpec}$	24.43s	N/A
MakerDAO (chedk_FundamentalEquationOfDai)	✓	✗$_{noSpec}$	9.3s	N/A
Array (check_allSame)	✓	✗	13.68s	N/A

Bug discovery process. As mentioned in previous sections, CertoraProver's points-to analysis either return a sounds points-to set, or fails. We uncovered each bug by identifying the location that the points-to analysis failed, and examining the abstract state at that point to discern why the code was not in fact memory safe.

Storage Corruption. Similar to memory, array elements in storage are laid out in contiguous slots, with the length stored in a separate slot. However, there is a special case for bytes or string of length 31 or lower: in this case the Solidity compiler packs the element data of the array in the upper (i.e., most-significant) bytes of the length storage slot, leaving the least-significant byte to hold the length of the array. Effectively, the Solidity compiler attempts to save space by using only a single storage slot for short byte arrays. For longer arrays (length 32 or greater), the length and the data are stored separately.

The Bug. The Solidity compiler generates multiple conditionals over the length of the array to select the correct algorithm for copying from memory to storage. One check is whether the length of the array is less than 32. If not, the generated code falls back on an "unpacked representation". It is only after the Solidity compiler checks if the length is greater than 32 does it check if the length is zero, which at that point is impossible. If, however, the length is 31 or less, the generated code enters a branch that uses the packed representation. Since memory on the EVM is read in 32-byte chunks, a single read at the beginning of the array element segment is sufficient to read the entirety of the array contents. Accordingly, the Solidity compiler generates code that increments the array pointer by 32-bytes (skipping the length field), and then generates an unconditional read from that position. The data read from this read is then packed together with the length and stored into memory. The bug occurs in the case where the array being copied is of length 0. The Solidity compiler never checks whether there is data for it to read after the length field. In other words, the Solidity compiler (mistakenly) assumes there must be at least one byte of data. However, if the array length is zero, the bytes immediately following the length field are totally arbitrary. Thus, the read and subsequent store of the "array data" in fact stores meaningless 31-bytes followed by the length in the least significant byte (i.e., 0). This particular bug was caught by the logic in Equation 25 in Appendix B: the element pointer $\mathcal{E}$ had $\top$ as its second field (indicating it was not necessarily safe for reading) and there was no proof that the array was non-empty in the numeric state.

7.3 Comparison to Other Tools

In addition to evaluating the main two claims of this paper in the above two sections, we also compared CertoraProver with Halmos [60], a tool that symbolically evaluates EVM bytecode to detect vulnerabilities.

Halmos is similar to CertoraProver for two reasons: first, it verifies EVM bytecode like CertoraProver, and second, it allows users to write high-level functional correctness properties. Halmos allows users to write tests in Solidity which it then interprets as specifications by replacing concrete inputs by symbolic ones.

For this experiment, we focused on ERC20 tokens and the famous MakerDAO bug.[12] First, we took all 3 tests Halmos had for ERC20 tokens[13] and translated them to specifications for CertoraProver. CertoraProver was able to successfully verify them all. Then, we attempted to convert additional specifications for ERC20 tokens[14] to specifications for Halmos but this was not possible due to limitations in the expressivity of Halmos's specifications. For example, CertoraProver allows ghost variables (similar to Dafny [69]) which are essential for expressing token invariants like "sum of balances = total supply" or properties like "sum of balances >= balance of any address" but these cannot be expressed in Halmos.

We therefore tried a simpler experiment—we converted the specification in Figure 1a to a specification for Halmos and ran their tool which was successful. However, Halmos was not able to verify this property; we reached out to the developers of Halmos who confirmed that Halmos does not support indexing into memory with symbolic offsets because it does not have any memory analysis that can help resolve the offsets. Table 4 summarizes our results. We see that for the properties that we got from Halmos, CertoraProver actually took longer in some cases. This is not surprising because the properties that Halmos can verify are simple and therefore do not require any of the additional optimizations and analyses CertoraProver has. These analyses add to the additional running time of CertoraProver.

8 Related Work

Many tools have been proposed for verifying and fuzzing smart contracts [19, 20, 22–24, 27, 31, 32, 36, 37, 40, 41, 49, 50, 53, 55–57, 65, 66, 68, 70–73, 76–78, 81, 84–87, 89, 92–94, 101]. For readers curious to learn more, we recommend recent surveys [45, 64] that provide comprehensive analyses of various tools and cover a wide spectrum of techniques from fuzzing to theorem proving.

Several symbolic execution tools [38, 40, 50, 61, 70, 94] analyze EVM bytecode looking for specific classes of vulnerabilities like arithmetic overflow, reentrancy, etc. Securify has a DSL that allows users to express security patterns that the code must satisfy. Unlike CertoraProver, these are low-level properties about load and store operations at the bytecode level. Zeus [66] has a DSL for specifying properties. Unlike CertoraProver, Zeus however verifies LLVM bitcode. Other tools [26, 59, 99] perform bounded verification of Solidity smart contracts at the source level by translating to the Boogie verification language. We conducted an experiment comparing CertoraProver with a recent symbolic execution tool, Halmos [60] in Section 7.3. We found that it is limited in its ability to support properties that involve indexing into memory.

Recent work [34, 35] uses Dafny [69] for formalizing EVM semantics and shows early results of using Dafny for verifying smart contracts that involve external calls and failures. CertoraProver too can reason about programs with these features and as our evaluation shows, runs on real-world protocols. [67] is another effort that formalizes the semantics of the EVM using the K-framework [96]. Other researchers have modeled EVM's memory [19, 86]—Smaragdakis et al. [86] use a concolic execution based idea for scaling a fully inter-procedural and strongly context-sensitive static analysis by balancing precision and performance. Grech et al. [53] and Brent et al. [32] propose static analyses that rely on specific program patterns, but tend to generate false positives for programs that do not fit in those patterns. Both however, avoid path explosion, which is a common challenge in static analysis. Lagouvardos et al. [68] presented a model of EVM's memory based on syntactic patterns that infer high-level information from low-level memory operations. They use an entirely different toolchain [52, 90]. Using their model for memory splitting is likely possible in

[12] https://hackmd.io/@SaferMaker/DAICertoraSurprise

[13] https://github.com/a16z/halmos/tree/main/examples/tokens/ERC20

[14] https://github.com/Certora/tutorials-code/tree/master/lesson4_invariants/erc20

theory but the benefit in speeding up SMT solving would be hard to measure—to the best of our knowledge, there is no support for writing and verifying diverse range of high-level specifications in their tool, unlike CertoraProver. We are eager to explore our key ideas in other domains.

There are many foundational algorithms for pointer analysis [25, 33, 39, 82, 83, 91] and several detailed literature surveys [62, 63]. Tools like Ghidra [51] and IDA Pro [79] that disassemble x86 must recover stack allocations, which is similar in essence to our allocation analysis. The idea of using memory analysis as a preprocessing step for scaling verification has been demonstrated in prior work for C programs [18, 42, 58] and directly for SMT-LIB formulae [48]. As explained in Section 1, prior approaches are not applicable for EVM due to the absence of explicit allocations.

9 Future Work and Limitations

We have encountered many real-world examples where the memory analysis improves the *precision* of formal verification in addition to providing speedups. For example, the analysis is helpful in resolving external calls that would otherwise have to be "havoced". We leave a thorough analysis of this for future work. In addition to analyzing memory, CertoraProver also analyzes EVM *storage*. We look forward to presenting and evaluating that in future work.

Even though this paper focused primarily on EVM bytecode generated by Solidity compiler (versions 0.4.25 and above), CertoraProver also works on bytecode generated from Vyper programs. An evaluation of the effectiveness of the analyses for Vyper is left for future work.

Limitations CertoraProver relies on known, existing allocation strategies used by various versions of the Solidity compiler. Although the pattern recognition mechanism of the allocation analysis is resilient to slight changes in the EVM bytecode formats, significant changes will cause CertoraProver to fail. However, in our experience using CertoraProver on a wide range of versions, such changes only occur at major releases of the Solidity compiler, which happen relatively infrequently, and generally offer enough warnings that CertoraProver can be adapted as appropriate. CertoraProver supports all Solidity compilers starting from 4.24 to the latest 8.26.

Since CertoraProver operates on EVM bytecode, it can support programs with "inline assembly" (i.e., handwritten EVM bytecode). However, our analysis is ultimately trying to infer shape and typing invariants for the low-level bytecode of the source solidity program. To a certain extent, the analysis depends on the bytecode generated by the Solidity compiler accessing memory in predictable ways. For example, if the compiler could prove that two structs `s1` and `s2` were definitely allocated sequentially, then it could (safely) compute a pointer to the last field of `s1` by subtracting 32 from `s2`. However, this does not conform to how CertoraProver's abstract domain expects memory to be accessed and the analysis would fail. Similarly, memory accesses via inline assembly may actually be safe, but appear to break typing/separation invariants from the point of view of our abstract domain, which can also cause the analysis to fail.

10 Data-Availability Statement

Our artifact is available at the ACM DL [9]. CertoraProver is also available at https://www.certora.com/ and documentation can be found at https://docs.certora.com/en/latest/index.html.

Acknowledgments

We thank the anonymous reviewers for their thoughtful feedback. We are grateful to Ondrej Kurak for helping with the artifact, and to Jorge Navas, Arie Gurfinkel, Aurojit Panda, and Zachary Tatlock for their feedback on earlier drafts of the paper. We thank our colleagues at Certora Inc. for all their contributions to CertoraProver.

References

[1] 2020. Solidity 0.8.0 Release Announcement. https://soliditylang.org/blog/2020/12/16/solidity-v0.8.0-release-announcement/.

[2] 2020. Solidity Code Generation Bug Can Cause Memory Corruption. https://medium.com/certora/bug-disclosure-solidity-code-generation-bug-can-cause-memory-corruption-bf65468d2b34.

[3] 2020. Solidity Empty Byte Array Copy Bug. https://soliditylang.org/blog/2020/10/19/empty-byte-array-copy-bug/.

[4] 2021. Solidity ABI Decoder Bug For Multi-Dimensional Memory Arrays. https://soliditylang.org/blog/2021/04/21/decoding-from-memory-bug/.

[5] 2022. Size Check Bug in Nested Calldata Array ABI-Reencoding. https://soliditylang.org/blog/2022/05/17/calldata-reencode-size-check-bug/.

[6] 2024. Aave Liquidity Protocol. https://aave.com/.

[7] 2024. Balancer DEX App. https://balancer.fi/.

[8] 2024. A better, smarter currency. https://makerdao.com/en/.

[9] 2024. CertoraProver Artifact On ACM DL. https://doi.org/10.1145/3580439.

[10] 2024. Curve. https://balancer.fi/.

[11] 2024. Decentralized Perpetual Exchange. https://gmx.io/.

[12] 2024. Earn, Borrow and Build on Morpho. https://morpho.org/.

[13] 2024. Liquid staking with stETH. https://lido.fi/.

[14] 2024. The modular lending platform. https://www.euler.finance/.

[15] 2024. Secure Money Markets For All Crypto Assets. https://www.silo.finance/.

[16] 2024. Smart Accounts to Own the Internet. https://safe.global/.

[17] 2024. Uniswap Protocol: Swap, earn, and build on the leading decentralized crypto trading protocol. https://uniswap.org/.

[18] Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, and Westley Weimer. 2002. Speeding Up Dataflow Analysis Using Flow-Insensitive Pointer Analysis. In *Static Analysis*, Manuel V. Hermenegildo and Germán Puebla (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 230–246.

[19] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2023. Inferring Needless Write Memory Accesses on Ethereum Bytecode. In *Tools and Algorithms for the Construction and Analysis of Systems*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer Nature Switzerland, Cham, 448–466.

[20] Elvira Albert, Pablo Gordillo, Alejandro Hernández-Cerezo, Albert Rubio, and Maria A. Schett. 2022. Super-Optimization of Smart Contracts. *ACM Trans. Softw. Eng. Methodol.* 31, 4, Article 70 (jul 2022), 29 pages. https://doi.org/10.1145/3506800

[21] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. 2018. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *Automated Technology for Verification and Analysis*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 513–520.

[22] Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. 2018. Running on Fumes–Preventing Out-of-Gas Vulnerabilities in Ethereum Smart Contracts using Static Resource Analysis. https://doi.org/10.48550/ARXIV.1811.10403

[23] Leonardo Alt, Martin Blicha, Antti E. J. Hyvärinen, and Natasha Sharygina. 2022. SolCMC: Solidity Compiler's Model Checker. In *Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part I* (Haifa, Israel). Springer-Verlag, Berlin, Heidelberg, 325–338. https://doi.org/10.1007/978-3-031-13185-1_16

[24] Leonardo Alt and Christian Reitwiessner. 2018. SMT-Based Verification of Solidity Smart Contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV* (Limassol, Cyprus). Springer-Verlag, Berlin, Heidelberg, 376–388. https://doi.org/10.1007/978-3-030-03427-6_28

[25] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. Citeseer.

[26] Pedro Antonino and A. W. Roscoe. 2021. Solidifier: bounded model checking solidity using lazy contract deployment and precise memory modelling. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (Virtual Event, Republic of Korea) *(SAC '21)*. Association for Computing Machinery, New York, NY, USA, 1788–1797. https://doi.org/10.1145/3412841.3442051

[27] Yulong Bao, Xue-Yang Zhu, Wenhui Zhang, Wuwei Shen, Pengfei Sun, and Yingqi Zhao. 2022. On Verification of Smart Contracts via Model Checking. In *Theoretical Aspects of Software Engineering*, Yamine Aït-Ameur and Florin Crăciun (Eds.). Springer International Publishing, Cham, 92–112.

[28] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng,

Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 415–442.

[29] Mike Barnett and K. Rustan M. Leino. 2005. Weakest-Precondition of Unstructured Programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Lisbon, Portugal) *(PASTE '05)*. Association for Computing Machinery, New York, NY, USA, 82–87. https://doi.org/10.1145/1108792.1108813

[30] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 171–177.

[31] Christian Bräm, Marco Eilers, Peter Müller, Robin Sierra, and Alexander J. Summers. 2021. Rich specifications for Ethereum smart contract verification. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 146 (oct 2021), 30 pages. https://doi.org/10.1145/3485523

[32] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities. 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, 454–469. https://doi.org/10.1145/3385412.3385990

[33] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. 1995. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Languages and Compilers for Parallel Computing*, Keshav Pingali, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 234–250.

[34] Franck Cassez, Joanne Fuller, Milad K. Ghale, David J. Pearce, and Horacio M. A. Quiles. 2023. Formal and Executable Semantics of the Ethereum Virtual Machine in Dafny. In *Formal Methods: 25th International Symposium, FM 2023, Lübeck, Germany, March 6–10, 2023, Proceedings* (Lübeck, Germany). Springer-Verlag, Berlin, Heidelberg, 571–583. https://doi.org/10.1007/978-3-031-27481-7_32

[35] Franck Cassez, Joanne Fuller, and Horacio Mijail Antón Quiles. 2022. Deductive Verification of Smart Contracts with Dafny. In *Formal Methods for Industrial Critical Systems: 27th International Conference, FMICS 2022, Warsaw, Poland, September 14–15, 2022, Proceedings* (Warsaw, Poland). Springer-Verlag, Berlin, Heidelberg, 50–66. https://doi.org/10.1007/978-3-031-15008-1_5

[36] Certik. 2024. Certik. https://www.certik.com/.

[37] Certora. 2024. Certora Prover Documentation. https://docs.certora.com/en/latest/.

[38] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. DefectChecker: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode. *IEEE Transactions on Software Engineering* 48, 7 (July 2022), 2189–2207. https://doi.org/10.1109/tse.2021.3054928

[39] Jong-Deok Choi, Michael Burke, and Paul Carini. 1993. Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, New York, NY, USA, 232–245. https://doi.org/10.1145/158511.158639

[40] ConsenSys. 2024. Mythril: Security analysis tool for EVM bytecode. https://github.com/Consensys/mythril.

[41] Filippo Contro, Marco Crosara, Mariano Ceccato, and Mila Dalla Preda. 2021. EtherSolve: Computing an Accurate Control-Flow Graph from Ethereum Bytecode. https://doi.org/10.48550/ARXIV.2103.09113

[42] Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2017. Rethinking Pointer Reasoning in Symbolic Execution. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 613–618.

[43] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. https://doi.org/10.1145/115372.115320

[44] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[45] Monika di Angelo and Gernot Salzer. 2019. A Survey of Tools for Analyzing Ethereum Smart Contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*. 69–78. https://doi.org/10.1109/DAPPCON.2019.00018

[46] Bruno Dutertre. 2014. Yices 2.2. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 737–744.

[47] EBPF. 2024. Dynamically program the kernel for efficient networking, observability, tracing, and security. https://ebpf.io/.

[48] Benjamin Farinier, Robin David, S\'ebastien Bardin, and Matthieu Lemerre. 2018. Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing. In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing, Vol. 57)*, Gilles Barthe, Geoff Sutcliffe, and Margus Veanes (Eds.). EasyChair, 363–380. https://doi.org/10.29007/dc9b

[49] Yu Feng, Emina Torlak, and Rastislav Bodik. 2020. Summary-Based Symbolic Evaluation for Smart Contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1141–1152. https://doi.org/10.1145/3324884.3416646

[50] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. ETHBMC: A Bounded Model Checker for Smart Contracts. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 155, 18 pages.

[51] Ghidra. 2024. A software reverse engineering (SRE) suite of tools developed by NSA's Research Directorate in support of the Cybersecurity mission. https://ghidra-sre.org.

[52] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1176–1186. https://doi.org/10.1109/ICSE.2019.00120

[53] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2020. MadMax: Analyzing the out-of-Gas World of Smart Contracts. *Commun. ACM* 63, 10 (sep 2020), 87–95. https://doi.org/10.1145/3416262

[54] Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. 2022. Elipmoc: Advanced Decompilation of Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 77 (apr 2022), 27 pages. https://doi.org/10.1145/3527321

[55] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 243–269.

[56] I. Grishchenko, Matteo Maffei, and Clara Schneidewind. 2022. EtherTrust: Sound Static Analysis of Ethereum bytecode. https://www.netidee.at/sites/default/files/2018-07/staticanalysis.pdf.

[57] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *Proc. ACM Program. Lang.* 2, POPL, Article 48 (dec 2017), 28 pages. https://doi.org/10.1145/3158136

[58] Arie Gurfinkel and Jorge A. Navas. 2017. A Context-Sensitive Memory Model for Verification of C/C++ Programs. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 148–168.

[59] Ákos Hajdu and Dejan Jovanović. 2020. *solc-verify: A Modular Verifier for Solidity Smart Contracts.* Springer International Publishing, 161–179. https://doi.org/10.1007/978-3-030-41600-3_11

[60] Team Halmos. 2024. Halmos: A symbolic testing tool for EVM smart contracts. https://github.com/a16z/halmos.

[61] Team HEVM. 2024. HEVM: Symbolic EVM Evaluator. https://github.com/ethereum/hevm.

[62] Michael Hind. 2001. Pointer Analysis: Haven't We Solved This Problem Yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Snowbird, Utah, USA) *(PASTE '01)*. Association for Computing Machinery, New York, NY, USA, 54–61. https://doi.org/10.1145/379605.379665

[63] Michael Hind and Anthony Pioli. 2000. Which Pointer Analysis Should I Use? *SIGSOFT Softw. Eng. Notes* 25, 5 (aug 2000), 113–123. https://doi.org/10.1145/347636.348916

[64] Nikolay Ivanov, Chenning Li, Qiben Yan, Zhiyuan Sun, Zhichao Cao, and Xiapu Luo. 2023. Security Threat Mitigation for Smart Contracts: A Comprehensive Survey. *ACM Comput. Surv.* 55, 14s, Article 326 (jul 2023), 37 pages. https://doi.org/10.1145/3593293

[65] Bo Jiang, Ye Liu, and W. K. Chan. 2018. *ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection.* Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/3238147.3238177

[66] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Ndss*. 1–12.

[67] KEVM. 2024. K Semantics of the Ethereum Virtual Machine (EVM). https://github.com/runtimeverification/evm-semantics.

[68] Sifis Lagouvardos, Neville Grech, Ilias Tsatiris, and Yannis Smaragdakis. 2020. Precise Static Modeling of Ethereum "Memory". *Proc. ACM Program. Lang.* 4, OOPSLA, Article 190 (nov 2020), 26 pages. https://doi.org/10.1145/3428258

[69] Rustan Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *16th International Conference, LPAR-16, Dakar, Senegal* (16th international conference, lpar-16, dakar, senegal ed.). Springer Berlin Heidelberg, 348–370. https://www.microsoft.com/en-us/research/publication/dafny-automatic-program-verifier-functional-correctness-2/

[70] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[71] Matteo Marescotti, Rodrigo Otoni, Leonardo Alt, Patrick Eugster, Antti E. J. Hyvärinen, and Natasha Sharygina. 2020. Accurate Smart Contract Verification Through Direct Modelling. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part III* (Rhodes, Greece). Springer-Verlag, Berlin, Heidelberg, 178–194. https://doi.org/10.1007/978-3-030-61467-6_12

[72] Anastasia Mavridou and Aron Laszka. 2017. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. *CoRR* abs/1711.09327 (2017). arXiv:1711.09327 http://arxiv.org/abs/1711.09327

[73] Anastasia Mavridou, Aron Laszka, Stachtiari Emmanouela, and Abhishek Dubey. 2019. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In *Proceedings of the 23nd International Conference on Financial Cryptography and Data Security (FC)*.

[74] John McCarthy. 1962. Towards a Mathematical Science of Computation. In *Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962*. North-Holland, 21–28.

[75] Move. 2024. Move. https://move-book.com/.

[76] Zeinab Nehaï, Pierre-Yves Piriou, and Frédéric Daumas. 2018. Model-Checking of Smart Contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. 980–987. https://doi.org/10.1109/Cybermatics_2018.2018.00185

[77] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 653–663. https://doi.org/10.1145/3274694.3274743

[78] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1661–1677.

[79] IDA Pro. 2024. A powerful disassembler and a versatile debugger. https://hex-rays.com/ida-pro.

[80] Rust. 2024. Rust. https://www.rust-lang.org/.

[81] RV. 2022. Runtime Verification. https://runtimeverification.com/.

[82] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 1998. Solving Shape-Analysis Problems in Languages with Destructive Updating. *ACM Trans. Program. Lang. Syst.* 20, 1 (jan 1998), 1–50. https://doi.org/10.1145/271510.271517

[83] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 1999. Parametric Shape Analysis via 3-Valued Logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 105–118. https://doi.org/10.1145/292540.292552

[84] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. arXiv:2005.06227 [cs.PL]

[85] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer Smart Contract Programming with Scilla. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 185 (oct 2019), 30 pages. https://doi.org/10.1145/3360611

[86] Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, and Ilias Tsatiris. 2021. Symbolic Value-Flow Static Analysis: Deep, Precise, Complete Modeling of Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 163 (oct 2021), 30 pages. https://doi.org/10.1145/3485540

[87] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2020. VERISMART: A Highly Precise Safety Verifier for Ethereum Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1678–1694. https://doi.org/10.1109/SP40000.2020.00032

[88] Solidity. 2024. Solidity. https://soliditylang.org/.

[89] Kunjian Song, Nedas Matulevicius, Eddie B. de Lima Filho, and Lucas C. Cordeiro. 2022. ESBMC-solidity: an SMT-based model checker for solidity smart contracts. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 65–69. https://doi.org/10.1145/3510454.3516855

[90] Souffle. 2024. Souffle. https://souffle-lang.github.io/index.html.

[91] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 32–41. https://doi.org/10.1145/237721.237727

[92] Ryan Stortz. 2022. Rattle – an Ethereum EVM binary analysis framework. https://blog.trailofbits.com/2018/09/06/rattle-an-ethereum-evm-binary-analysis-framework/.

[93] Bryan Tan, Benjamin Mariano, Shuvendu K. Lahiri, Isil Dillig, and Yu Feng. 2022. SolType: Refinement Types for Arithmetic Overflow in Solidity. *Proc. ACM Program. Lang.* 6, POPL, Article 4 (jan 2022), 29 pages. https://doi.org/10.1145/3498665

[94] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/3243734.3243780

[95] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13.

[96] Runtime Verification. 2024. The K Framework. https://kframework.org/.

[97] Vyper. 2024. Vyper. https://vyper.readthedocs.io/en/stable/.

[98] Wei Wang, Clark Barrett, and Thomas Wies. 2017. Partitioned memory models for program analysis. In *Verification, Model Checking, and Abstract Interpretation: 18th International Conference, VMCAI 2017, Paris, France, January 15–17, 2017, Proceedings 18*. Springer, 539–558.

[99] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. 2019. Formal Specification and Verification of Smart Contracts for Azure Blockchain. arXiv:1812.08829 [cs.PL]

[100] wasm. 2024. wasm. https://webassembly.org/.

[101] Scott Wesley, Maria Christakis, Jorge A. Navas, Richard Trefler, Valentin Wüstholz, and Arie Gurfinkel. 2021. Compositional Verification of Smart Contracts Through Communication Abstraction. In *Static Analysis*, Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi (Eds.). Springer International Publishing, Cham, 429–452.

[102] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. 2009. A Concurrent Portfolio Approach to SMT Solving. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 715–720. https://doi.org/10.1007/978-3-642-02658-4_60