Precise Static Identification of Ethereum Storage Variables

Sifis Lagouvardos
University of Athens
Athens, Greece
Dedaub
Athens, Greece
sifis.lag@di.uoa.gr

Yannis Bollanos
Dedaub
Athens, Greece
ybollanos@dedaub.com

Michael Debono
Friendly Maltese Citizens
Msida, Malta
mixy1@ctf.mt

Neville Grech
Dedaub
Msida, Malta
me@nevillegrech.com

Yannis Smaragdakis
University of Athens
Athens, Greece
Dedaub
Athens, Greece
smaragd@di.uoa.gr

Abstract

Smart contracts are small programs that run autonomously on the blockchain, using it as their persistent memory. The predominant platform for smart contracts is the Ethereum VM (EVM). In EVM smart contracts, a problem with significant applications is to identify data structures (in blockchain state, a.k.a. "storage"), given only the deployed smart contract code. The problem has been highly challenging and has often been considered nearly impossible to address satisfactorily. (For reference, the latest state-of-the-art research tool fails to recover nearly all complex data structures and scales to 50% of contracts.) Much of the complication is that the main on-chain data structures (mappings and arrays) have their locations derived dynamically through code execution.

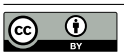
We propose sophisticated static analysis techniques to solve the identification of on-chain data structures with extremely high fidelity and completeness. Our analysis scales nearly universally and recovers deep data structures. Our techniques are able to identify the exact types of data structures with 95.70% precision and at least 94.96% recall, compared to a state-of-the-art tool managing 83.30% and 55.65% respectively. Strikingly, the analysis is often more complete than the storage description that the compiler itself produces, with full access to the source code.

CCS Concepts

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

Keywords

Program Analysis, Smart Contracts, Decompilation, Ethereum, Reverse Engineering



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773121

ACM Reference Format:
Sifis Lagouvardos, Yannis Bollanos, Michael Debono, Neville Grech, and Yannis Smaragdakis. 2026. Precise Static Identification of Ethereum Storage Variables. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3744916.3773121

1 Introduction

Smart contracts on programmable blockchains have been successfully used to implement complex applications, mostly of a financial nature [42]. The dominant platform for smart contracts is the Ethereum VM (EVM): the execution layer behind blockchains such as Ethereum, Arbitrum, Optimism, Binance, Base, and many more. Millions of smart contracts have been deployed on these chains and can be invoked on-demand. Many thousands of them are in active use every day.

To enable the persistence of data between different blockchain transactions, contracts employ the blockchain as their persistent memory, to save their state. In EVM terms, this persistent memory is called "*storage*" and is accessed using special random-access instructions. A challenge of high value has emerged out of the use of storage in smart contracts: recovering high-level storage structures from the deployed form of the smart contract, i.e., from EVM bytecode. This task is crucial for several applications:

- **Security Analysis**: A number of smart contract vulnerabilities arise from incorrect handling of storage variables, such as storage collisions in upgradable contracts [35]. Precise modeling of storage is required for detecting such vulnerabilities.
- **Decompilation and Reverse Engineering**: Tools [5, 18, 23] that decompile EVM bytecode back to high-level code rely on storage modeling to reconstruct variable declarations and data structures [29].
- **Off-chain Applications**: Blockchain explorers, debuggers, and other off-chain tools need to interpret storage data to provide meaningful information to users. They often rely on compiler-generated metadata, which may be incomplete or unavailable [15] for interesting smart contracts like proprietary bots or hacker contracts.

- **Static Analysis and Verification**: Precise storage modeling enables advanced static analysis and formal verification of smart contracts, facilitating the detection of bugs and the proof of correctness [6, 12, 17, 44].

Smart contracts are written overwhelmingly in the Solidity language, which allows developers to define storage variables ranging from simple value types to complex, arbitrarily nested data structures such as arrays, mappings, and structs.

However, when Solidity code is compiled into EVM bytecode, much of the high-level structure and type information is lost. This is because the EVM's permanent storage is a simple key-value store mapping 256-bit keys to 256-bit values. Due to the luxury of having a large key space, the default pattern for high-level languages targeting the EVM is to translate high-level constructs into low-level storage access patterns using cryptographic hashing and arithmetic operations to compute storage slots dynamically. This transformation creates a significant gap between the high-level representation of storage variables and the low-level state of permanent storage reflected on the blockchain.

Existing approaches to storage modeling face significant limitations. Early frameworks [3, 41] often reasoned about storage operations only when storage indexes were constants, sacrificing precision or completeness when dealing with dynamic data structures. While some tools [6, 17] introduced methods to infer high-level storage structures, they lacked support for arbitrarily nested data structures and complex storage patterns. Recent tools like VarLifter [29] attempt to recover storage layouts but struggle with scalability and completeness, failing to produce output for a substantial portion of real-world contracts.

Contributions. This paper introduces DYELS, a static analysis approach that accurately infers high-level storage structures from EVM bytecode. Our key contributions are:

- **Static Storage Modeling**: We develop a novel static analysis that fully supports arbitrarily nested composite data structures in Solidity. By employing a recursive storage analysis, DYELS scalably and precisely reconstructs complex storage layouts from low-level bytecode.
- **Evaluation on All Deployed Contracts**: DYELS's scalable design allows us to evaluate its performance on enormous datasets. The first consists of 377,132 smart contracts with ground truth provided by the Solidity compiler and allows us to assess the faithfulness of DYELS's recovered inferences. The second includes 903,805 deduplicated contracts, corresponding to all 73 million non-empty contracts deployed on the Ethereum mainnet.[1] This enables scalable recognition of storage patterns on the whole blockchain.
- **Evaluation Against Existing Tools**: We find the current state-of-the-art tool, VarLifter [29], to terminate on only 50.5% of contracts. When successful, VarLifter misses over 40% of storage variables, including most non-trivial structures. This performance underscores the difficulty of the problem being solved. In comparison, DYELS analyzes 99.36% of contracts and provides higher-fidelity results, recovering the vast majority (93.55%) of storage structures with excellent precision (97.70%).
- **Enhanced Completeness Beyond Compiler Metadata**: We show that DYELS can infer storage variables and structures not present in compiler-generated metadata, particularly those involving low-level storage patterns common in upgradable contracts. Analyzing the entire blockchain, we were able to identify over 300,000 contract addresses making use of such widespread low-level variables.

The core of DYELS is released as open-source software as part of the Gigahorse lifting toolchain [2] and has seen significant adoption in both industry and academia. Its output is integrated into the decompilation of https://app.dedaub.com, the Dedaub security and decompilation tool suite used by over 10,000 registered users.

2 Background

We next provide background on the Ethereum Virtual Machine and its storage model.

2.1 Ethereum and the Ethereum Virtual Machine

Ethereum is a decentralized blockchain platform that enables the execution of smart contracts—autonomous programs that run on the blockchain. Smart contracts are predominantly written in the Solidity high-level language and are compiled into bytecode for execution on the Ethereum Virtual Machine (EVM). The EVM has dominated as an execution platform and has been adopted by most other programmable blockchains.

The EVM is a stack-based virtual machine designed to execute smart contracts securely and deterministically. It operates on 256-bit words, utilizes its own *stack*, *memory*, *transient storage* and *persistent storage*, and provides a Turing-complete execution environment. This paper focuses on *persistent storage* or just *storage*.

2.2 EVM Bytecode Format and Execution Model

EVM bytecode is a sequence of instructions, each represented by a single-byte opcode (with an immediate argument for PUSH opcodes). The EVM supports a rich, albeit unconventional set of operations, which includes anything from arithmetic, logic, control flow, hashing, and state and memory interaction.

As a stack-based machine, most EVM opcodes perform computations using a stack of 256-bit words, when such an opcode has a fixed operand or return size. The EVM also features different kinds of state. Memory is a *dense* addressable byte array that is cleared at the end of each transaction with a smart contract (from the outside or nested, via calls from one contract to the next). A recent addition is transient storage, which is cleared at the end of the outermost transaction.

2.3 Storage in the EVM

At the EVM level, persistent storage (or simply *storage*) is a persistent key-value store where both keys and values are 256-bit words. Storage maintains the state of a contract between transactions. Storage is simply a sparse word array indexed by 256-bit words, spanning from keys 0 to $2^{256} - 1$.

[1] All numbers reported over deployed contracts are as of Jun. 28, 2025.

[2] https://github.com/nevillegrech/gigahorse-toolchain

High-level languages like Solidity provide structured data types such as integers, arrays, mappings, and structs. The Solidity compiler maps these high-level constructs to EVM storage using specific patterns, now also widely-adopted in other compilers.

The EVM only allows reading from and writing to storage using the `SLOAD` and `SSTORE` instructions, both of which index storage using a 32-byte index value, with the value read or written also being fixed at 32 bytes. As high-level languages need to implement arbitrarily complex data structures using such very low-level primitives, there is a huge disparity between the source and bytecode representations.

2.4 EVM Storage Model and Solidity Storage Layout

During compilation, the Solidity compiler generally predictably orders smart contract storage variables using a deterministic heuristic (e.g., by employing C3 linearization upon inheritance) and assigns a slot p to each variable, and accumulates p by an appropriate amount at each step.

The following cases briefly capture the mapping of high-level constructs to EVM storage:

- **Value Types**: Simple value types (e.g., `uint256`, `bool`, `address`) are stored in sequential storage slots starting from slot 0. To optimize space, multiple small values may be packed into a single 32-byte storage slot. For example, two `uint128` variables can share one slot, and smaller types like `bool` and `uint8` can be packed together.
- **Static Arrays**: Fixed-size arrays are stored by sequentially allocating storage slots for each element. For an array declared as `T[n]`, where `T` is the element type and `n` is the fixed size, elements are stored starting from slot p, the assigned slot of the array variable. The element at index i is stored at slot $p + i$.
- **Dynamic Arrays**: Dynamic arrays store their length at a fixed slot p, and their elements are stored starting from the Keccak-256 hash of slot p. (This is a cryptographic hash function, expected to be collision-resistant.) Specifically, element i is stored at position $keccak256(p) + i$ where p is the slot assigned to the array variable. This allows for arrays of arbitrary length without preallocating storage slots.
- **Mappings**: Mappings use a hashing scheme to avoid key collisions. A mapping declared as `mapping(K => V)` at slot p stores a value associated with key k at $keccak256\,(encode(k) \parallel p)$ where $\parallel$ denotes concatenation, and $encode(k)$ is the padded representation of key k. This ensures that each key-value pair in the mapping has a unique storage slot.
- **Structs**: Structs are stored by sequentially allocating storage slots for each of their members, similar to value types. For a struct declared as `struct S { ... }` and a variable of type `S` assigned to slot p, its members are stored starting from slot p. If a struct contains members that are arrays or mappings, the storage rules for arrays and mappings are applied recursively to those members.

The recursive nature of these storage rules makes static modeling of storage complex, especially when only the EVM bytecode is available. For instance, mappings and dynamic arrays involve runtime computations of storage slots using hash functions, which are challenging to resolve statically. Additionally, the packing of multiple variables into a single storage slot requires bit-level instruction analysis to accurately extract individual variables.

3 Smart Contract Storage Patterns

We next show a simple smart contract that will serve as an example to explain how the most common data structures provided by Solidity are implemented in the low-level EVM bytecode.

```
contract StorageExample {
  uint256 public supply;  // slot 0x0
  address public owner;   // slot 0x1
  bool public isPaused;   // slot 0x1
  uint256[] public supplies; // slot 0x2
  mapping (address => bool) public admins; // slot 0x3
  struct vals {uint256 field0; uint256 field1;}
  mapping (address => mapping(uint256 => vals)) public complex; //
      slot 0x4
}
```

Figure 1: Example Smart Contract

3.1 Low-Level Implementation Patterns

For our example contract in Figure 1, the storage layout translates to the following low-level operations, also shown schematically in Figure 2:

- **Simple Values (slot 0x0)**: The `supply` variable occupies a full slot:
  - Load: `SLOAD(0x0)`
  - Store: `SSTORE(0x0) = newSupply`
- **Packed Values (slot 0x1)**: The `owner` (20 bytes) and `isPaused` (1 byte) share slot 0x1:
  - Load owner: `SLOAD(0x1)` followed by `AND(loaded, 0xffffffffffffffffffffffffffffffffffffffff)`
  - Load `isPaused`: `SLOAD(0x1)` followed by `AND(SHR(0xa0, loaded), 0xff)`. The masked variable is often followed by two `ISZERO(ISZERO(masked))`.
  - Store requires reading existing value, masking, and combining with new value.
- **Dynamic Array (slot 0x2)**: For `supplies`:
  - Length access: `length = SLOAD(0x2)`
  - Element i access: `SLOAD`(`keccak256(0x2)` + i)
  - Store: `SSTORE`(`keccak256(0x2)` + i) `= value`
- **Mapping (slot 0x3)**: For `admins`:
  - Key k access: `SLOAD`(`keccak256(pad32(`k`)` $\parallel$ `0x3))`
  - Store: `SSTORE`(`keccak256(pad32(`k`)` $\parallel$ `0x3))` `= value`
- **Nested Mapping (slot 0x4)**: For `complex`:
  - For keys k_1, k_2:
    `field0` is accessed using: `SLOAD`(`keccak256(pad32(`k_2`)` $\parallel$ `keccak256(pad32(`k_1`)` $\parallel$ `0x4)))`
    `field1` is accessed using: `SLOAD`(`keccak256(pad32(`k_2`)` $\parallel$ `keccak256(pad32(`k_1`)` $\parallel$ `0x4))` + 1)

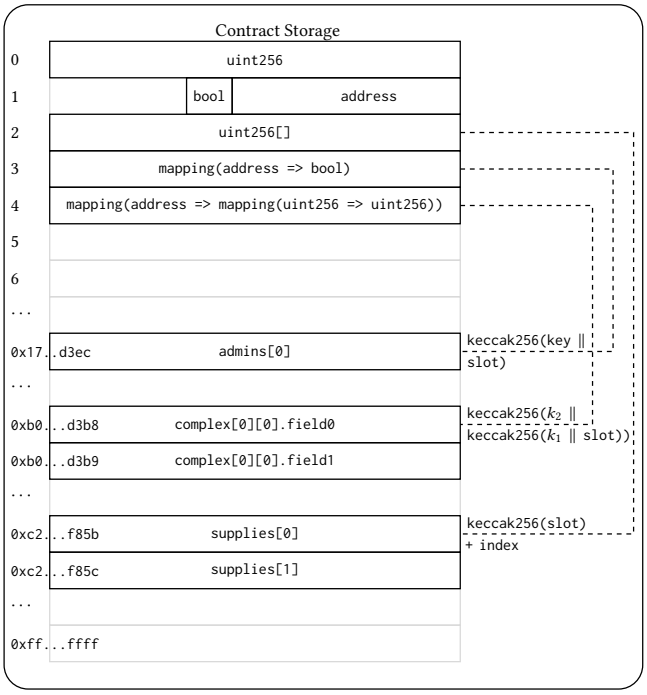


Figure 2: Low-level Storage Layout Implementation of our example in Figure 1

```
bytes32 internal constant _ADMIN_SLOT =
 0xb53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103;

// returns the value of the address variable stored at _ADMIN_SLOT
function _getAdmin() internal view returns (address) {
  return StorageSlot.getAddressSlot(_ADMIN_SLOT).value;
}
struct AddressSlot { address value; }

// Returns an `AddressSlot` with member `value` located at `slot`.
function getAddressSlot(bytes32 slot) internal pure
 returns (AddressSlot storage r) {
  assembly {
    r.slot := slot
  }
}
```

Figure 3: Low-level code implementing the ERC-1967 standard.

3.2 Low-level Storage Patterns in High-level Code

Although high-level storage patterns allow developers to implement powerful protocols, making use of complex high-level data structures, the storage allocation algorithm has its drawbacks. Solidity does not offer a high-level way to override the assigned storage slot of a variable declaring it at an arbitrary slot. This functionality is needed by various standards requiring compatible storage layouts. The most important such standard is ERC-1967 [32], standardizing the allocated storage slots to be used for the implementation, admin, and beacon contract addresses of widely-used [45] upgradable proxy contracts. To support these standard patterns developers make use of Solidity's inline assembly [8], as shown in Figure 3.

V: set of variables S: set of program statements
C: set of 256-bit numbers Int: set of 16-bit numbers

Figure 4: Type domain definitions

I : r := LOAD(iv) | r $\in V$, $I \in S$, iv $\in V$ SLOAD statement I loads into r the value of storage location pointed-to by iv
I : STORE(iv) := u | $I \in S$, iv $\in V$, u $\in V$ SSTORE statement I stores the value of u into the storage location pointed to by iv
r := ADD/SUB/MUL(a, b) | r $\in V$, a $\in V \cup C$, b $\in V \cup C$ Binary arithmetic operation over variable or constant operands a, b
v $\rightarrow$ c | v $\in V$, c $\in C$ Constant folding and constant propagation analysis. v has constant value c.
l := ϕ(u) | l $\in V$, u $\in V$ SSA PHI instructions
r := HASH(a, $*$) | r $\in V$, a $\in V$ SHA3 operation that computes the keccak256 hash of a variable number of args, storing it into r

Figure 5: Input relation definitions

Such low-level code patterns allow users to use storage variables that are not declared as such. Thus, these variables are unknown to the Solidity compiler, and are not included in its storage layout metadata. This incompleteness of the compiler-produced metadata will be examined in our evaluation of Section 7.

4 Analysis Preliminaries and Input

The DYELS approach is a static analysis of the program's (smart contract's) code that identifies the low-level patterns that the Solidity compiler produces to implement the high-level features presented in Section 3. The challenge is to maintain the right level of analysis precision and scalability/computability, since the analysis needs to model the derivation structure of arbitrary dynamic numerical quantities.

Figures 4 and 5 define the input schema for our analysis. (In using these predicates, we drop elements that are not needed for the rule at hand, e.g., we may write "r := LOAD(iv)" instead of "I : r := LOAD(iv)" when the instruction identifier I is unused.)

While these types and relations originate from the Elipmoc/Gigahorse lifter toolchain [16, 18, 25, 26], our approach is not restricted to this framework. Instead, it can be applied to any mature decompilation framework that lifts the stack-based EVM bytecode into a register-based static-single-assignment (SSA) representation.

Some relations in Figure 5 directly correspond to the register-based representation, such as LOAD and STORE, while others, like ADD, SUB, and MUL, also incorporate the results of a constant folding and constant propagation analysis.

Finally, the HASH relation, which supports the low-level implementation patterns presented in Section 3, stems from the EVM "memory" analysis [26] built on top of Gigahorse. While this specific implementation is tied to Gigahorse, similar EVM memory analyses [1, 20] have been developed on top of other SSA-based

$$
\begin{array}{lcll}
\text{type } \mathit{SInd} & = & \mathit{Const}_I & (\text{c: } C) \\
& | & \mathit{ArrA}_I & (\text{par: } \mathit{SInd}, \text{iv: } V) \\
& | & \mathit{ArrD}_I & (\text{par: } \mathit{SInd}) \\
& | & \mathit{Map}_I & (\text{par: } \mathit{SInd}, \text{kv: } V) \\
& | & \mathit{Offs}_I & (\text{par: } \mathit{SInd}, \text{of: } \mathit{Int})
\end{array}
$$

Figure 6: Vocabulary of storage index value expressions

analysis frameworks.

5 Structure Identification

The DYELS analysis has two main parts: a) discovering the *structure* of a program's storage layout (e.g., which structures are nested arrays or mappings); b) discovering the *types* of data stored in every entry of each structure. This section presents the first part: how to identify the data structures in a smart contract's storage.

5.1 Storage Index Value-flow Analysis

The backbone of the analysis is a value-flow analysis that computes the values of all potential storage index expressions, and then uses the ones that end up being used in *actual* storage operations to identify the constructs in the program's storage layout.

Figure 6 presents our definition of the storage index values for our value-flow analysis. The Algebraic Data Type (ADT) in the figure aims to capture accesses to Solidity's arbitrarily-nested high-level structures. The ADT effectively defines what the analysis can infer about potential storage indexes.

Const_I is the only non-recursive kind of *SInd* type. Every storage index will include a Const_I as the leaf of its ADT value, since all high-level storage structures are assigned a constant offset by the compiler. The rest of the storage index types are recursively built on top of a pre-existing *SInd* instantiation, encoded as *par* ("par" for "parent"). These include ArrA_I and ArrD_I used to model operations on dynamic arrays, Map_I for mapping operations, and Offs_I, which enables supporting struct accesses. In addition to the *par* index, *SInd* values that model an index to a high-level data structure (array or mapping) also include the index or key variables.

To compute the possible storage indexes for arbitrarily-nested data structures we define our analysis as a set of recursive inference rules. The rules are faithfully transcribed from a fully mechanized implementation, so they should be precise, modulo mathematical shorthands used for conciseness.

The analysis first computes two new relations.

$\mathsf{v} \xrightarrow{S} \mathsf{si} \mid \mathsf{v} \in \mathsf{V}, \mathsf{si} \in \mathit{SInd}$
Storage Index Overapproximation:
Variable v holds potential storage index si.
$\Downarrow \mathsf{si} \mid \mathsf{si} \in \mathit{SInd}$
Actual Storage Index:
si ends up being used in a storage loading/storing operation.

5.1.1 Storage Index Overapproximation. Figure 7 contains our analysis logic for overapproximating the possible storage index values.

We start with the simpler cases of the analysis for inferring the structure of storage indexes, with detailed explanation, to also serve as introduction to the meaning of inference rules and of the input schema.

$$\text{(BASE)} \quad \frac{\mathsf{v} \rightarrow \mathsf{c}}{\mathsf{v} \xrightarrow{S} \mathit{Const}_I(\mathsf{c})}$$

$$\text{(MAPPING)} \quad \frac{\mathsf{pv} \xrightarrow{S} \mathsf{si} \qquad \mathsf{v} := \mathsf{HASH}(\mathsf{kv}, \mathsf{pv})}{\mathsf{v} \xrightarrow{S} \mathit{Map}_I(\mathsf{si}, \mathsf{kv})}$$

$$\text{(ARRAY DATA)} \quad \frac{\mathsf{pv} \xrightarrow{S} \mathsf{si} \qquad \mathsf{v} := \mathsf{HASH}(\mathsf{pv})}{\mathsf{v} \xrightarrow{S} \mathit{ArrD}_I(\mathsf{si})}$$

$$\text{(ARRAY ACCESS)} \quad \frac{\begin{array}{c}\mathsf{pv} \xrightarrow{S} \mathit{ArrD}_I(\mathsf{si}) \quad \mathsf{v} := \mathsf{ADD}(\mathsf{pv}, \mathsf{i}) \\ \mathsf{i} := \mathsf{MUL}(\mathsf{iv}, \mathsf{c}) \qquad \mathsf{iv} : V \quad \mathsf{c} : C\end{array}}{\mathsf{v} \xrightarrow{S} \mathit{ArrA}_I(\mathsf{si}, \mathsf{iv})}$$

$$\text{(OFFS1)} \frac{\mathsf{pv} \xrightarrow{S} \mathsf{si} \quad \mathsf{v} := \mathsf{ADD}(\mathsf{pv}, \mathsf{c}) \quad \mathsf{si} : \mathit{ArrA}_I \mid \mathit{Map}_I \quad \mathsf{c} : \mathit{Int}}{\mathsf{v} \xrightarrow{S} \mathit{Offs}_I(\mathsf{si}, \mathsf{c})}$$

$$\text{(OFFS2)} \frac{\mathsf{pv} \xrightarrow{S} \mathit{Offs}_I(\mathsf{si}, \mathsf{o}) \qquad \mathsf{v} := \mathsf{ADD}(\mathsf{pv}, \mathsf{c}) \quad \mathsf{c} : \mathit{Int}}{\mathsf{v} \xrightarrow{S} \mathit{Offs}_I(\mathsf{si}, \mathsf{c} + \mathsf{o})}$$

Figure 7: Inference Rules for Storage indexes

The "BASE" rule produces the initial set of possible storage indexes by considering the facts of the constant folding and constant propagation analysis provided by the Gigahorse/Elipmoc framework. Per the input schema, " $\rightarrow$ " is the predicate capturing the result of the constant propagation/folding, matching an IR variable (if it always holds a constant value) to its value. This means that every static constant in the contract code will be considered as a possible constant storage index, to be used either as-is or as a building block of more complex indexes.

The "MAPPING" rule models mapping accesses with the help of the HASH predicate provided by the EVM "memory" modeling analysis. The rule states that:

- if a variable, pv points to a likely storage index si,
- and if its concatenation to the contents of another variable, kv, is hashed in the smart contract code (using the EVM's hash operation),
- then the hash result variable will hold a Map_I (mapping access index) over si and kv is the key variable of the modeled mapping access operation.

The next two rules model dynamic arrays in storage. The storage locations of a dynamic array are determined by first hashing an array identifier, and then performing index arithmetic via addition and multiplication.

Similarly to the case of mappings, the "ARRAY DATA" rule will create a new index value pointing to the start of an array after inferring a HASH operation that hashes the contents of a variable, and that variable points to a pre-existing index.

The second rule ("ARRAY ACCESS") will infer that if a variable holding an ArrD_I is added to the result of the multiplication of a variable and a constant, the variable defined by the addition

$$\text{(A1)}\ \frac{\mathsf{v} \xrightarrow{S} \mathsf{si} \quad (\_ := \mathsf{LOAD}(\mathsf{v}) \vee \mathsf{STORE}(\mathsf{v}) := \_)}{\Downarrow \mathsf{si}}$$

$$\text{(A2)}\ \frac{\mathsf{sv} \xrightarrow{S} \mathsf{si} \quad \mathsf{lv} := \phi^*(\mathsf{sv}) \quad (\_ := \mathsf{LOAD}(\mathsf{lv}) \vee \mathsf{STORE}(\mathsf{lv}) := \_)}{\Downarrow \mathsf{si}}$$

$$\text{(A3)}\ \frac{\Downarrow \mathsf{si} \qquad \mathsf{si} : \mathit{ArrA}_I \mid \mathit{ArrD}_I \mid \mathit{Map}_I \mid \mathit{Offs}_I}{\Downarrow \mathsf{si.par}}$$

Figure 8: Inference Rules for recognizing actual (used) storage indexes

operation will point to a new $ArrA_I$ value, inheriting the parent index of the $ArrD_I$ value and using the multiplied variable as its access/indexing value.

It is worth asking whether the above code patterns are *always* indicating a dynamic array, or could arise for random code. If the compiled code has been produced via compilation, these patterns are very unlikely to arise for non-array structures. There is no other data structure with both contiguous (indicated via addition) and regular (indicated via multiplication with a constant) storage location access. Furthermore, the presence of a hashed value, via a 1-argument hash operation, adds even more confidence to the inference. Finally, the potential over-approximation of storage indexes will be, in the next step of the analysis, checked against the use of the index as a proper array index. All these elements contribute to a very high-fidelity inference.

Finally, the last two rules create $Offs_I$ values that are used to model struct accesses in EVM storage. A struct is being accessed by addition of constant field offsets to a base storage index. The base storage index is that of a mapping or array. (If it is a mere constant index, then there is no way to distinguish the struct from just an explicit listing of its fields.)

The first rule will create a new $Offs_I$ when a small integer is added to a variable pointing to a Map_I or $ArrA_I$ value, while the second one recursively creates new $Offs_I$ values for additions of existing *OffsetIndex* values and small integers.

5.1.2 Filtering Out Non-Realized Indexes. The next step for computing a smart contract's storage layout is to identify the subset of indexes computed in the overapproximating $\xrightarrow{S}$ relation that are *actually* used in storage operations. Relation $\Downarrow$ is used to compute these storage indexes as shown in the rules of Figure 8.

The first two rules are inferring the end-level storage indexes when they are used in `LOAD/STORE` operations either directly or through PHI operations via the ϕ^* relation, the transitive closure of the ϕ relation of our input. The last rule is introduced to transitively infer that all parent indexes of "actual" storage indexes are considered "actual" indexes as well.

This seemingly very simple logic hides an important subtlety. This concerns the treatment of PHI (ϕ) instructions: the data-flow merge instructions in a static-single-assignment (SSA) representation. PHI instructions are merging the values for the same higher-level variable that arrive, via different program paths, to a control-flow merge point. For instance, if a high-level program variable `x` is set in two different branches ("then" or "else") of an `if` statement, then `x` is produced by a PHI whose arguments are the `x`-versions in the branches that merge.

Note that, in the rules we have seen (Figure 7), the left-hand-side variable of a PHI instruction does *not* become the first part of a $\xrightarrow{S}$ entry, even if the right-hand-side variables (one or multiple) *are* in it. Doing so would result in analysis non-termination. A PHI may be merging different potential indexes, all captured at run-time by the same variable. Then if the variable cyclically feeds into itself (as in the case of code with a loop), we would end up with an unbounded number of potential storage index inferences.

This is the importance of the "A2" rule, handling PHI instructions. Although PHI instructions do not yield more storage indexes, recognized storage indexing patterns propagate through the transitive closure of PHI instructions. In this way, we can get the confidence of recognizing actual storage indexes, without attempting to fully track them at every point in the program.

5.1.3 Storage index analysis results on our example. Now that we have presented how the storage indexes are computed it is interesting to see the results of the actual index ($\Downarrow$) relation for our example in Figure 1:

```
ConstI(0x1)
ConstI(0x0)
ConstI(0x2)
ConstI(0x3)
ConstI(0x4)
ArrayAI(ConstI(0x2), 0x14d)
MapI(ConstI(0x3), 0x11e)
MapI(ConstI(0x4), 0x80)
MapI(MapI(ConstI(0x4), 0x80), 0x8e)
OffsI(MapI(MapI(ConstI(0x4), 0x80), 0x8e), 1)
```

As can be seen, the computed actual storage indexes are constant indexes `0x0` (containing the 256-bit `supply` variable), `0x1` (containing variables `owner` and `isPaused`), and composite indexes to access array `supplies` at index `0x2` and mappings admins and complex at indexes `0x3` and `0x4`, along with their parent indexes.

5.2 Inferring Storage Constructs from Storage Index Values

Figure 9 presents our definition of the *SCons* (storage construct) Algebraic Data Type, used to describe all data structures that can be found in Solidity smart contracts. The constructor cases of *SCons* cover the different *SInd* types, while also introducing *Var* as an option. Instances of *Var* express a value-typed fundamental unit of data at the end of our nesting chain. This can either be a top-level value-typed variable, the element of an array, the key to a mapping, or a struct member. Finally, the *PVar* type is used to express a "packed" variable: a construct that takes up part of a 32-byte storage word.

```
type SCons = Const (c: C)
           | Arr   (par: SCons)
           | Map   (par: SCons)
           | Offs  (par: SCons, of: Int)
           | Var   (par: SCons)
           | PVar  (par: SCons, b: (Int, Int))
```

Figure 9: Storage Construct Type

To define the algorithms that identify the program's high-level structures we need to introduce the following additional notation/-computed predicates.

$SCons(\text{si}) \mid \text{si} \in SInd$ *SCons* constructor, syntactically translating corresponding *SInd* cases.
$\downarrow \text{sc} \mid \text{sc} \in SCons$ Relation containing all storage constructs in a program.
$I \mapsto \text{sv} \mid I \in S, \text{sv} \in Var$ Relation mapping storage `LOAD/STORE` instructions to the storage variable they operate on.

Following the computation of the $\Downarrow$ "used storage index" relation we can populate the $\downarrow$ relation with all program structures, as shown in Figure 10.

$$(\textsc{Base})\frac{\Downarrow \text{si}}{\downarrow SCons(\text{si})}$$

$$(\textsc{Var})\frac{\Downarrow \text{si} \quad \nexists \text{si}' : (\Downarrow \text{si}' \quad SCons(\text{si}') = Arr(SCons(\text{si})) \vee SCons(\text{si}') = Map(SCons(\text{si})))}{\downarrow Var(SCons(\text{si}))}$$

Figure 10: Using the "used storage index" inferences to compute a program's storage constructs.

The first rule considers all constructs that were translated from storage indexes. The second one introduces new *Var* instances for every translated construct that is never used as a parent index to a more complex construct.

Finally, in Figure 11, we map the `LOAD` and `STORE` statements to the instance of *Var* they operate on.

5.2.1 (Packed) Variable Partitioning. After computing the contract's storage construct instances we need to identify instances of multiple variables packed together into the same 32-byte storage word. These instances of *Var* are encoded as *PVar* ("packed variable").

This analysis step, presented in Appendix A, models all reads and writes of each *Var* instance through low-level bit-masking and shifting operations. If all read and write operations write to non-conflicting sub-word segments, *PVar* instances are introduced, replacing the pre-existing *Var* inferences.

$$\frac{\downarrow Var(SCons(\text{si})) \quad \text{v} \xrightarrow{S} \text{si} \quad (I : \_ := \text{LOAD}(\text{v}) \quad \vee \quad I : \text{STORE}(\text{v}) := \_)}{I \mapsto Var(SCons(\text{si}))}$$

$$\frac{\downarrow Var(SCons(\text{si})) \quad \text{v} \xrightarrow{S} \text{si} \quad \text{u} := \phi^*(\text{v}) \quad (I : \_ := \text{LOAD}(\text{u}) \quad \vee \quad I : \text{STORE}(\text{u}) := \_)}{I \mapsto Var(SCons(\text{si}))}$$

Figure 11: Mapping `LOAD` and `STORE` statements to the storage variables they operate on.

Table 1: Kinds of operations supported by each value type with the corresponding EVM instructions implementing them

ops	bytesX	uintX	intX	address	bool
equal	`EQ, SUB`	`EQ, SUB`	`EQ, SUB`	`EQ, SUB`	`EQ, SUB`
logical	✗	✗	✗	✗	`ISZERO`
comp	`LT, GT`	`LT, GT`	`SLT, SGT`	`LT, GT`	✗
bitwise	`AND, OR, XOR, NOT`	`AND, OR, XOR, NOT`	`AND, OR, XOR, NOT`	✗	✗
shifts	`SHL, SHR, MUL, DIV`	`SHL, SHR, MUL, DIV`	`SHL, SAR`	✗	✗
arithm	✗	`ADD, SUB, MUL, DIV, MOD, EXP, ADDMOD MULMOD`	`ADD, SUB, MUL, EXP, SMOD, SDIV`	✗	✗
byte ind	`BYTE`	✗	✗	✗	✗

5.2.2 Storage construct Var and PVar inferences on our example. At this point in our analysis pipeline the following *Var* inferences will be produced for our example in Figure 1, each corresponding to a (potentially packed) top-level variable, array element, mapping value, or struct member:

```
Var(Const(0x0))            // uint256 supply
PVar(Const(0x1), 0, 19)    // address owner
PVar(Const(0x1), 20, 20)   // bool isPaused
Var(Array(Const(0x2)))     // uint256[] supplies
Var(Map(Const(0x3)))       // mapping admins
// the 2 fields of struct value of nested mapping complex
Var(Map(Map(Const(0x4))))
Var(Offs(Map(Map(Const(0x4))), 1))
```

6 Value Type Inference

The second part of the DYELS analysis is to identify the types of data structure entries, i.e., the types of the *Var* (and *PVar*) instances identified in the structure recognition of the previous section.

Once *Var* and *PVar* instances have been identified, type inference over them is primarily an instance of inferring monomorphic types by process of elimination, based on compatible operations.

The value-types supported by Solidity are the following:

- `uintX` with `X in range(8, 256, 8)` (all numbers from 8 to 256, for each increment of 8): Unsigned integers, left-padded
- `intX` with `X in range(8, 256, 8)`: Signed integers, left-padded
- `address`: Address type, 20 bytes in width, left-padded
- `bool`: Boolean, left padded
- `bytesX` with `X in range(1, 32, 1)`: Fixed width bytearrays, right-padded

Table 1 captures the DYELS systematic encoding of the different high-level operations Solidity supports for its value types, along with the low-level EVM instructions that implement them. In addition to the table, `bool` typed variables support the high-level short-circuiting `&&` and `||` operators, supported via control-flow patterns (i.e., with no single corresponding low-level EVM instruction).

In most cases, a simple analysis can identify a storage variable's type, given that the packed variable partitioning analysis of the previous subsection will give us its width. If a tightly packed variable is then moved to the leftmost bytes of a variable (i.e., is right-padded) we identify its type as `bytesX`.

For packed variables that are moved to the stack as left-padded variables, we can easily distinguish signed- and unsigned-integer-typed variables as the former will be used in signed arithmetic operations, after getting the variable's length extended to 256 bits via the `SIGNEXTEND` operation.

The cases that remain ambiguous require further analysis to correctly infer the variable type. These cases of ambiguity are:

- `bool` vs. `uint8`
- `address` vs. `uint160`
- `uint256` vs. `int256` vs. `bytes32`

The first two cases are treated by initially assigning variables to the most restricted type (`bool`, `address`) and replacing it with the respective `uint` inference if the storage variable ends up being used in integer arithmetic.

The last case is the most challenging as the 3 possible types have many common supported operations, as can be seen in Table 1. In addition we can't take advantage of syntactic information such as variable alignment or length extension operations to get type clues.

We handle this case by first assigning an `any32` type to all 32-byte width variables, and replacing that by any other inference based on the type constraints propagated to them. If no other type constraints are propagated to the storage variable when our analysis reaches its fixpoint, we replace the `any32` inference with `uint256`.

7 Evaluation

The analysis of DYELS, presented as recursive inference rules, is implemented as a set of recursive Datalog rules on top of the Gigahorse/Elipmoc framework.

We evaluate DYELS over a diverse set of unique smart contracts deployed on the Ethereum mainnet. To make the evaluation systematic, we take advantage of the `storageLayout` json field output by the Solidity compiler since version] 0.5.13 [39], released in 2019. This compiler output provides the *ground truth* for our evaluation.

To evaluate our approach we gather all contracts deployed on the Ethereum mainnet up to block 22,800,000 (proposed on the 28th of June 2025) and deduplicate them, considering two contracts to be duplicates if they contain the same bytecode modulo constant values. This query returned 903,805 distinct contracts corresponding to 73,729,326 smart contract deployments.[3] We filter the dataset, removing 80,580 distinct contracts that implement "minimal proxy" patterns. These contracts are deployed an enormous number of times and add nothing but noise to an evaluation like ours, since they have no storage variables. Following this collection we identified the contracts that have published, verified source code and use the appropriate compiler version in each case, for ground truth extraction. This results in our *ground truth* dataset of 377,132 distinct contracts, corresponding to 1,944,788 deployments on the Ethereum mainnet.

[3]Our initial query returned 903,806 distinct contracts corresponding to 78,621,489 smart contract deployments. However 4,892,163 of these deployments corresponded to the empty bytecode. This happens for contracts that are created and self-destructed within a single transaction. Since these are never stored on the blockchain's state we cannot retrieve their bytecode and we have to disregard them.

Table 2: Analysis execution statistics for DYELS on the ground truth dataset.

	DYELS
Analysis Terminated	374,959 (**99.42%**)
Timeouts	2,173 (**0.58%**)
Errors	0
Total	377,132

Table 3: DYELS execution breakdown. Reported decompilation/inline/analysis time excludes the 2,173 contracts that timed out.

DYELS analysis stage	Time (secs)	Timeouts
Decompilation	589,757	2,146
Inline	581,755	0
DYELS analysis	186,999	27
Total	1,358,511	2,173

We evaluate DYELS against the state-of-the-art VarLifter tool [29]. As VarLifter's storage layout output is not compatible with the `storageLayout` output of the Solidity compiler, we parse its textual output and produce `solc`-compatible layouts. Additionally, the comparison to the ground truth for VarLifter is more relaxed than that for DYELS, as VarLifter's output lacks some crucial information:

- For storage variables packed into a single slot, no information regarding the offset of each variable is produced.
- In the case of struct types that serve as values to mappings, VarLifter does not produce any information about the layout of the struct members.

We conducted our experimental evaluation on an idle Ubuntu 24.04 machine with 2 Intel Xeon Gold 6426Y 16 core CPUs and 512G of RAM. We compile our Datalog analysis using Souffle [21, 22, 36] version 2.4.1, with 32-bit integer arithmetic and openmp disabled. An execution cutoff of 300s is used for both tools. DYELS runs 30 single-threaded analysis jobs in parallel, taking advantage of the native parallelization of the Gigahorse/Elipmoc framework. (This is only a disadvantage for the timings of DYELS, since it may introduce minor contention.) VarLifter, lacking such support, is executed sequentially.

Our evaluation examines analysis performance on the axes of scalability, precision, and completeness (recall). The metrics have standard definitions, given the ground truth (i.e., the compiler's output): precision is the fraction *#Success/#Reports*, while recall is the fraction *#Success/#GroundTruth*. Therefore a missing inference lowers recall, while a wrong inference lowers both precision and recall.

7.1 Scalability

Table 2 shows the execution statistics of DYELS for the full ground truth dataset of 377,132 distinct contracts. DYELS is able to analyze 99.42% of contracts in the dataset, under the given 300s execution cutoff.

Table 3 gives more insights into DYELS's performance. Based on the time summary for the 374,959 contracts analyzed, DYELS takes

Table 4: Analysis execution statistics for DYELS and VarLifter on the trimmed dataset.

	DYELS	VarLifter
Analysis Terminated	3748 (**99.36%**)	1906 (**50.53%**)
Timeouts	24 (**0.64%**)	1064 (**28.20%**)
Errors	0	802 (**21.26%**)
Total	3772	3772

an average of 3.62 seconds per contract. It is important to note that DYELS's analysis execution is only accounting for 13.76% (0.5s on average) of the total execution time—the rest is spent on the underlying framework's decompilation and inlining stages. Additionally, the storage analysis of DYELS introduces very few additional timeouts, with the vast majority of timeouts coming from the underlying Gigahorse/Elipmoc decompilation stage.

For comparing against VarLifter we soon realized that, due to scalability limitations, we would not be able to run VarLifter on our full ground truth dataset. To overcome this we introduce a *trimmed* version of our ground truth dataset containing 1% of its original contracts, sampled randomly. Table 4 shows the execution statistics of DYELS and VarLifter for the *trimmed* version of the ground truth dataset, consisting of 3772 contracts. DYELS is able to successfully analyze nearly all contracts in this dataset. On the other hand VarLifter is able to successfully analyze just over half of the contracts. This is informative, since the VarLifter publication [29] does not include any statistics on the tool's timeouts and errors.[4]

7.2 Precision

To measure analysis precision we consider an analysis inference to be successful if it is able to infer all the variables in a storage slot and their exact types. It should be noted that achieving a successful inference becomes more difficult as the nestedness and complexity of the defined variables increase.

For example, the nested mapping defined at slot 0x4 in Figure 2 requires:

- Identifying that it is a 2-nested mapping
- Recovering both key-types based on the success criteria
- Recovering the value's struct type based on the success criteria

Table 5: Analysis results for DYELS on the 374,959 deduplicated contracts it analyzed.

	Result
Ground Truth	$\leq$ 5,329,677
DYELS Reports	5,044,373
DYELS Success	4,827,641 (Precision **95.70%**, Recall $\geq$ **90.58%**)

Table 5 contains the analysis results of DYELS for the 374,959 distinct contracts it successfully analyzes. We are focusing on the Precision numbers, i.e., the percentage of DYELS inferences that also appear in the ground truth. We can see that DYELS is a very precise analysis, able to infer a variable's exact type 95.7% of the time.

Next, we consider how DYELS compares against the state-of-the-art VarLifter. Since VarLifter times out for nearly 50% of contracts, we perform the precision comparison over the 1896 contracts that both tools managed to analyze. Table 6 shows the results.

Table 6: Results for DYELS and VarLifter on the 1896 contracts (subset of the trimmed dataset) analyzed by both tools.

	Result
Ground Truth	$\leq$ 25963
DYELS Reports	24844
DYELS Success	23365 (Precision **97.70%**, Recall $\geq$ **93.55%**)
VarLifter Reports	17485
VarLifter Success	13898 (Precision **83.30%**, Recall $\geq$ **55.65%**)

DYELS manages to perform even better for this subset of contracts, successfully identifying the exact types in 97.70% of reported variables. On the other hand, VarLifter is significantly less precise, at 83.30%. Notably, VarLifter's output can be self-evidently imprecise, reporting colliding types for the same slot.

7.3 Completeness

We evaluate the completeness of DYELS by examining its ability to recover the ground truth, i.e., the *recall* of the analysis: the percentage of variables in the ground truth that DYELS recovers. Table 5 shows the recall of the DYELS to be 90.58% for the 374,959 contracts of the ground truth dataset it was able to analyze.

However, *this number is only a lower bound.*

The reason is that real-world smart contracts often need to declare unused variables. These variables are available to the compiler's ground truth (since the compiler has access to the source code) but cannot be detected by any bytecode-level analysis. (Inferring these unused variables is a no-op for all practical purposes.)

The principal case of contracts that declare unused variables is upgradable proxy contracts. Upgradable proxy contracts need to maintain backwards compatibility of their storage layouts throughout their upgrades. (Failure to do this can result in storage collisions, a well-recognized problem, also studied in past literature [35].) The need to maintain compatible storage layouts makes developers continue to declare variables that are no longer used. Additionally, to avoid storage collisions, developers (and standard upgradability libraries) preemptively declare unused static arrays in storage, in order to keep a distance between the variables of a Solidity contract and those of the sub-contracts that inherit from it, so that future versions of the super-contract can add more variables.

One can observe from Table 5 that the majority of the incompleteness is reflected in the many fewer instances of variables inferred by DYELS relative to the ground truth: the ground truth contains 285,304 (5.35%) more variables, numerically.

We manually inspected 50 randomly-selected instances of such variables missed by DYELS, and all of them turned out to be unused variables. Our sampling (of 50 out of 5,329,677) has a margin of error of 13.86% for a confidence level of 95%. [5] That is, with 95%

[4] We have also confirmed the high timeout and error rates on the VarLifter publication's own dataset and artifact. We have publicly reported this discrepancy along with other issues we identified in the publication's evaluation via github issues https://github.com/wsong-nj/VarLifter/issues/1 and https://github.com/wsong-nj/VarLifter/issues/2.

[5] One can verify with a standard margin-of-error calculator.

confidence DYELS misses at most 39,543 variables for the contracts in our dataset, with the rest of the 245,761 reported missing variables being unused ones. Based on the above, the ground truth includes 5,083,916 variables instead of the 5,329,677 reported by `solc`. Thus, with 95% confidence, the real recall of DYELS is *at least* 94.96%.

Yet another way to appreciate the completeness of DYELS is by comparing the analysis recall to that of VarLifter. Table 6 shows the recall results for both tools, on the contracts in the *trimmed* dataset that are analyzed by both tools. DYELS has a significantly higher recall than VarLifter, successfully identifying (at least) 93.55% of declared contract variables and their exact types compared to VarLifter's 55.65%.

The incompleteness of VarLifter is due to its incomplete path extraction algorithm that will not attempt to visit all code paths. In contrast, the DYELS analysis is recursive to arbitrary depth, yet fully scalable—e.g., avoiding non-termination issues via the subtle treatment described in Section 5.1.2.

Furthermore, VarLifter's implementation is heavily limited in terms of supported structures and their composability:

- Nested mappings can only have a maximum depth of 2.
- Nested static arrays can only have at most 2 dimensions.
- Only value type and string arrays are supported.
- Only structs with members of value types are supported for mapping values. (Whereas storage slots of struct types may also contain strings.)

In contrast, the DYELS inference algorithm of Section 5 is capable of detecting arbitrarily-nested storage structures.

7.4 Practical Value

To demonstrate the practical value of our storage modeling analysis we show how DYELS can benefit downstream applications and analyses, deployed on the whole chain:

(1) DYELS can recover hundreds of thousands of variables missed by the compiler.
(2) DYELS is invaluable for clients analyses, such as the identification of reentrancy guards—a key component of reentrancy analyses.

As context, it is worth noting that DYELS applies to all deployed contracts on the Ethereum blockchain (and not just the ground truth dataset of our evaluation, which consisted of contracts with available source code) and it successfully analyzes **99.50%** of them (899,278 out of 903,805 distinct bytecodes), corresponding to 73,721,169 deployed contracts.

7.4.1 Incompleteness in the compiler-produced metadata. As discussed in Section 3, common low-level storage patterns are not included in the compiler-produced `storageLayout` json. Therefore, DYELS often retrieves *more* storage variables than the compiler itself. Of course, the compiler misses these variables because of the use of inline assembly. However, the inline assembly information is still available to the compiler, and certainly in much more accessible form than that available to a bytecode-only analyzer.

To quantify the impact of the storage variables missed by the compiler-produced metadata we identified 7 cases of low-level storage slots used in either EIP/ERC standard proposals or popular libraries. Each of these code patterns contains variables that the compiler misses, yet DYELS (overwhelmingly) detects.[6] We identify these patterns in all deployed contracts and present statistics on their frequency in Table 7.

Table 7: On-chain usage of various standard patterns using low-level storage variables.

Pattern	# Distinct	# Deployments
Diamond Proxy	503	1,147
1967 Proxy	15,970	257,180
1967 Beacon	207	93,820
1822 Proxy	359	23,331
OZ Proxy	121	2,836
OZ Initializable	8,213	10,928
OZ ReentrancyGuardUpgr	2,409	3,392

```
modifier nonReentrant() {
  require(_status == NOT_ENTERED);
  _status = ENTERED;
  _; // function code goes here
  _status = NOT_ENTERED;
}
```

Figure 12: Standard reentrancy guard pattern

Thus, DYELS is able to recover hundreds of thousands of instances of low-level storage variables deployed in the Ethereum mainnet.

7.4.2 Client: Reentrancy Guards. As an example of a client analysis that clearly relies on DYELS, we consider a detector of reentrancy guards. The typical reentrancy guard pattern, shown in Figure 12, treats a special storage variable as a mutex, using it to prevent reentry to the contract's other protected functions.

The identification of reentrancy guards is instrumental for increasing the precision of static reentrancy analyses by considering calls between the two guard-setting statements as reentrancy-safe.

We implement a detector for reentrancy guards in 70 lines of Datalog code, combining the storage analysis of DYELS with value-flow, control-flow, and data-flow predicates and components provided by the Gigahorse framework [16].

The checker identifies reentrancy guards in 187,633 distinct contract bytecodes, corresponding to 811,639 contracts deployed on the Ethereum mainnet.

8 Related Work

Reasoning about the usage of the EVM's storage has been instrumental for analysis tools and decompilers. However, no past tools (other than VarLifter—extensively compared earlier) attempt to statically fully recover storage structures as-if in the source program. For instance, past analyses may have inferred "this is an access to the `balances` mapping" but not the width of an entry, the full nested structure of the mapping, or the type of elements. Such work, discussed next, can potentially benefit from our techniques.

[6] Although there is no ground truth for variables that the compiler misses, the performance of DYELS for them is expected to be similar to that for variables that the compiler does know about. Patterns such as those of Figure 3 are not a problem for a bytecode-level analyzer, like DYELS. In manual sampling of contracts with inline assembly, we have found no evidence of different DYELS precision or recall for these variables.

Most early frameworks [3, 41] would only precisely reason about storage loading / storing statements with indexes resolved to constant values, sacrificing precision or completeness in low-level code treating dynamic data structures. Madmax [17] was the first work to propose an analysis that inferred high-level structures (arrays and mappings) from EVM bytecode. This analysis enabled MadMax to detect storage-related vulnerabilities focusing on griefing and DoS. Ethainter [6] also made use of the storage modeling introduced in [17] to detect guarding patterns and track the propagation of taint through storage. An implicit modeling of storage was also achieved (as `keccak256` expressions with free variables) in [38].

In addition, as DYELS is publicly available in an open-source repository, the results of previous snapshots of our codebase have been incorporated into independently published research tools. One such example is BlockWatchdog [43], which leverages DYELS's storage modeling to identify storage variables that hold addresses of external contracts.

The recent CRUSH tool [35] implements a storage collision vulnerability analysis for upgradable proxy contracts. Part of this work involves modeling storage, including a modeling of mappings, arrays, and byte-ranges of constant-offset storage slots to discriminate between storage variables packed into a single slot. Unfortunately, the work lacks support for arbitrarily-nested data structures. These same techniques have been applied (for the same security application) to the Proxion tool [9]. Another recent tool [2] analyzes storage access patterns to precisely compute the gas bounds of contracts via a Max-SMT based approach.

Other work has focused on analyzing the usage patterns of the EVM's various "memory" stores. [26] proposes techniques to infer high-level facts from EVM bytecode. These inferences include high-level uses of operations reading from memory (hashing operations, external calls) memory arrays and their uses. DYELS relies on these inferences for the modeling of the storage index values, and the propagation of type constraints through memory. The Certora prover employs a memory splitting transformation [20] after modeling the allocations of high-level arrays and structs of EVM contracts and the aliasing between different allocations. This transformation allows the tool to consider disjoint memory locations separately, speeding up SMT queries by up to 120x. In other work [1] the uses of memory are analyzed to identify optimization opportunities.

Several other end-to-end applications rely on storage modeling. Storage modeling is also used in blockchain explorers and can be done dynamically. The now-discontinued storage explorer tool, evm.storage, used such a dynamic analysis, by examining the use of hash pre-images derived from past executions of the contract. Analysis of confused deputy attack contracts has employed both static and dynamic storage modeling techniques [19]. A smart contract policy enforcer, EVM-SHIELD, utilizes storage modeling to pinpoint functions that perform state updates and adds pre- and post-conditions within the smart contract itself to prevent malicious transactions on-chain [46].

Related to our work, past tools [11, 47] have been proposed to infer the ABI interfaces of unknown contracts by inferring the structures and types of public function arguments. Such tools can benefit from our work by taking our inferences into account in their type recovery efforts. As an example, SigRec [11], lacking a model of the EVM's storage, considers any variable read from or written to storage to be of type `uint256`.

Outside the domain of smart contracts, a number of techniques on variable recognition and type inference of binaries are relevant to our work [4, 7, 10, 13, 14, 27, 28, 31, 37]. Static-analysis-based approaches [4, 31, 34] have historically seen widespread adoption in this setting. Recent learning-based tools [30, 33, 40] have also been successful in recovering type information from binaries. More closely related to our approach, OOAnalyzer [37] uses Prolog to infer C++ classes from binaries.

9 Conclusion

We presented DYELS, a static-analysis-based lifter for storage variables from the binaries of Ethereum smart contracts. DYELS, powered by an analysis of low-level storage indexes, is able to resolve arbitrarily-nested high-level data structures from the low-level bytecode. Compared against the state-of-the-art in a diverse dataset of real-world contracts, DYELS manages to excel in all evaluation dimensions: scalability, precision, and completeness. Analyzing all deployed contracts, DYELS identifies variables missed by the Solidity compiler in hundreds of thousands of deployed contracts.

Acknowledgments

We gratefully acknowledge funding by ERC Advanced Grant PINDESYM (101095951).

Data-Availability Statement

The implementation of DYELS is available in the https://github.com/nevillegrech/gigahorse-toolchain open source repository. The paper's artifact [24] is available on Zenodo.

References

[1] 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.

[2] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2024. Synthesis of Sound and Precise Storage Cost Bounds via Unsound Resource Analysis and Max-SMT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1186–1197. doi:10.1145/3650212.3680352

[3] 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.

[4] Gogul Balakrishnan and Thomas Reps. 2007. DIVINE: DIscovering Variables IN Executables. In *Verification, Model Checking, and Abstract Interpretation*, Byron Cook and Andreas Podelski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–28.

[5] Jonathan Becker. 2023. Heimdall is an advanced EVM smart contract toolkit specializing in bytecode analysis and extracting information from unverified contracts. https://github.com/Jon-Becker/heimdall-rs

[6] 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. doi:10.1145/3385412.3385990

[7] Juan Caballero and Zhiqiang Lin. 2016. Type Inference on Executables. *ACM Comput. Surv.* 48, 4, Article 65 (May 2016), 35 pages. doi:10.1145/2896499

[8] Stefanos Chaliasos, Arthur Gervais, and Benjamin Livshits. 2022. A study of inline assembly in solidity smart contracts. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 165 (Oct. 2022), 27 pages. doi:10.1145/3563328

[9] Cheng-Kang Chen, Wen-Yi Chu, Muoi Tran, Laurent Vanbever, and Hsu-Chun Hsiao. 2024. Proxion: Uncovering Hidden Proxy Smart Contracts for Finding

Collision Vulnerabilities in Ethereum. arXiv:2409.13563 [cs.CR] https://arxiv.org/abs/2409.13563

[10] Ligeng Chen, Zhongling He, and Bing Mao. 2020. CATI: Context-Assisted Type Inference from Stripped Binaries. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 88–98. doi:10.1109/DSN48063.2020.00028

[11] Ting Chen, Zihao Li, Xiapu Luo, Xiaofeng Wang, Ting Wang, Zheyuan He, Kezhao Fang, Yufei Zhang, Hang Zhu, Hongwei Li, Yan Cheng, and Xiao-song Zhang. 2021. SigRec: Automatic Recovery of Function Signatures in Smart Contracts. *IEEE Transactions on Software Engineering* (2021), 1–1. doi:10.1109/TSE.2021.3078342

[12] Zhiyang Chen, Ye Liu, Sidi Mohamed Beillahi, Yi Li, and Fan Long. 2024. Demystifying Invariant Effectiveness for Securing Smart Contracts. *Proc. ACM Softw. Eng.* 1, FSE, Article 79 (July 2024), 24 pages. doi:10.1145/3660786

[13] E. N. Dolgova and A. V. Chernov. 2009. Automatic reconstruction of data types in the decompilation problem. *Program. Comput. Softw.* 35, 2 (March 2009), 105–119. doi:10.1134/S0361768809020066

[14] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. Scalable variable and data type detection in a binary rewriter. *SIGPLAN Not.* 48, 6 (June 2013), 51–60. doi:10.1145/2499370.2462165

[15] etherscan.io. 2017. Etherscan Blockchain explorer. https://etherscan.io

[16] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 1176–1186. doi:10.1109/ICSE.2019.00120

[17] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. *Proc. ACM Programming Languages* 2, OOPSLA (Nov. 2018). doi:10.1145/3276486

[18] 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. doi:10.1145/3527321

[19] Fabio Gritti, Nicola Ruaro, Robert McLaughlin, Priyanka Bose, Dipanjan Das, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. 2023. Confusum Contractum: Confused Deputy Vulnerabilities in Ethereum Smart Contracts. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1793–1810. https://www.usenix.org/conference/usenixsecurity23/presentation/gritti

[20] 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 (Oct. 2024), 32 pages. doi:10.1145/3689796

[21] Xiaowen Hu, David Zhao, Herbert Jordan, and Bernhard Scholz. 2021. An efficient interpreter for Datalog by de-specializing relations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 681–695. doi:10.1145/3453483.3454070

[22] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430.

[23] Tomasz Kolinko and Palkeo. 2020. Panoramix – Decompiler at the heart of eveem.org. https://github.com/palkeo/panoramix

[24] Sifis Lagouvardos, Yannis Bollanos, Michael Debono, Neville Grech, and Yannis Smaragdakis. 2025. *Precise Static Identification of Ethereum Storage Variables (Artifact)*. doi:10.5281/zenodo.17709265

[25] Sifis Lagouvardos, Yannis Bollanos, Neville Grech, and Yannis Smaragdakis. 2025. The Incredible Shrinking Context... in a Decompiler Near You. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA060 (June 2025), 24 pages. doi:10.1145/3728935

[26] 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. doi:10.1145/3428258

[27] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. In *18th Annual Network and Distributed System Security Symposium (NDSS'11)*.

[28] Daniel Lehmann and Michael Pradel. 2022. Finding the Dwarf: Recovering Precise Types from WebAssembly Binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 410–425. doi:10.1145/3519939.3523449

[29] Yichuan Li, Wei Song, and Jeff Huang. 2024. VarLifter: Recovering Variables and Types from Bytecode of Solidity Smart Contracts. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 271 (Oct. 2024), 29 pages. doi:10.1145/3689711

[30] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. 2019. TypeMiner: Recovering Types in Binary Programs Using Machine Learning. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren (Eds.). Springer International Publishing, Cham, 288–308.

[31] Alan Mycroft. 1999. Type-Based Decompilation (or Program Reconstruction via Type Reconstruction). In *Programming Languages and Systems*, S. Doaitse Swierstra (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 208–223.

[32] Santiago Palladino, Francisco Giordano, and Hadrien Croubois. 2019. ERC-1967: Proxy Storage Slots. https://eips.ethereum.org/EIPS/eip-1967

[33] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. StateFormer: fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 690–702. doi:10.1145/3468264.3468607

[34] G. Ramalingam, John Field, and Frank Tip. 1999. Aggregate structure identification and its application to program analysis. 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, 119–132. doi:10.1145/292540.292553

[35] Nicola Ruaro, Fabio Gritti, Robert McLaughlin, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. 2024. Not your Type! Detecting Storage Collision Vulnerabilities in Ethereum Smart Contracts. (2024).

[36] Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 196–206. doi:10.1145/2892208.2892226

[37] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. 2018. Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables. 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, 426–441. doi:10.1145/3243734.3243793

[38] 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. doi:10.1145/3485540

[39] Solidity Team. 2019. Solidity 0.5.13 Release Announcement. https://soliditylang.org/blog/2019/11/14/solidity-0.5.13-release-announcement/

[40] Zirui Song, YuTong Zhou, Shuaike Dong, Ke Zhang, and Kehuan Zhang. 2024. TypeFSL: Type Prediction from Binaries via Inter-procedural Data-flow Analysis and Few-shot Learning. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1269–1281. doi:10.1145/3691620.3695502

[41] 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)*. ACM, New York, NY, USA, 67–82. doi:10.1145/3243734.3243780

[42] Sam Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William Knottenbelt. 2023. SoK: Decentralized Finance (DeFi). In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies* (Cambridge, MA, USA) *(AFT '22)*. Association for Computing Machinery, New York, NY, USA, 30–46. doi:10.1145/3558535.3559780

[43] Shuo Yang, Jiachi Chen, Mingyuan Huang, Zibin Zheng, and Yuan Huang. 2024. Uncover the Premeditated Attacks: Detecting Exploitable Reentrancy Vulnerabilities by Identifying Attacker Contracts. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 128, 12 pages. doi:10.1145/3597503.3639153

[44] Shuo Yang, Xingwei Lin, Jiachi Chen, Qingyuan Zhong, Lei Xiao, Renke Huang, Yanlin Wang, and Zibin Zheng. 2024. Hyperion: Unveiling DApp Inconsistencies using LLM and Dataflow-Guided Symbolic Execution. arXiv:2408.06037 [cs.SE] https://arxiv.org/abs/2408.06037

[45] Mengya Zhang, Preksha Shukla, Wuqi Zhang, Zhuo Zhang, Pranav Agrawal, Zhiqiang Lin, Xiangyu Zhang, and Xiaokuan Zhang. 2025. An Empirical Study of Proxy Contracts at the Ethereum Ecosystem Scale . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 2996–3008. doi:10.1109/ICSE55347.2025.00083

[46] Xiaoli Zhang, Wenxiang Sun, Zhicheng Xu, Hongbing Cheng, Chengjun Cai, Helei Cui, and Qi Li. 2024. EVM-Shield: In-Contract State Access Control for Fast Vulnerability Detection and Prevention. *IEEE Transactions on Information Forensics and Security* 19 (2024), 2517–2532. doi:10.1109/TIFS.2024.3349852

[47] Kunsong Zhao, Zihao Li, Jianfeng Li, He Ye, Xiapu Luo, and Ting Chen. 2023. DeepInfer: Deep Type Inference from Smart Contract Bytecode. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 745–757. doi:10.1145/3611643.3616343

f ⇝ t | f ∈ V, t ∈ V Limited data flows analysis: f flows to t through low-level shifting and masking operations.
r := LowBytesMask(u, w) | r ∈ V, u ∈ V, w ∈ *Int* Masking operation (w bytes of mask width) used in casting u to value types `uintX`, `intX`, `address`, `bool`
r := HighBytesMask(u, w) | r ∈ V, u ∈ V, w ∈ *Int* Masking operation (w bytes of mask width) used in casting u to the `bytesX` value types
r := LShift/RShift(u, n) | r ∈ V, u ∈ V, n ∈ *Int* Variable u is shifted to the left/right by n bytes.
r := BooleanCast(u) | r ∈ V, u ∈ V Low-level cast to boolean using two consecutive `ISZERO` operations.
HighLevelOpUse(v) | v ∈ V Variable v is used in high-level operation (e.g., non-storage-address computation, calls).

Figure 13: Additional input relation definitions

A (Packed) Variable Partitioning Analysis

To identify instances of multiple high-level variables taking up part of and sharing the same EVM storage word, we need to model their uses in high-level operations as well as their definitions.

Figure 13 contains additional input relations needed to identify instances of these packed variables. Predicates handling masking and shifting are convenience wrappers for low-level arithmetic and bitwise operations of variables and constants: `LowBytesMask` and `HighBytesMask` correspond to operation patterns that use `AND` instructions; `LShift` to patterns with `MUL` and `SHL` (a left-shift); `RShift` to patterns with `DIV` and `SHR` (right-shift).

The following relations capture the inferences of the analysis.

I : v := sv[l : h] | I ∈ S, v ∈ V, sv ∈ *Var*, l ∈ *Int*, h ∈ *Int* **Partial Read**: Variable v holding byte range [l : h] of storage variable sv, loaded via I, is used in high-level operation or is not cast further.
I_S, I_L : sv[l : h] := v | I_S ∈ S, I_L ∈ S, sv ∈ *Var*, l ∈ *Int*, h ∈ *Int*, v ∈ V **Partial Write**: Variable v is written to byte range [l : h] of storage variable sv in statement I_S. All other contents of sv are retained as loaded in statement I_L.
I: sv[l : h] | I ∈ S, sv ∈ *Var*, l ∈ *Int*, h ∈ *Int* Aggregation of the two previous relations.
↯ sv | sv ∈ *Var* **Partitioning Analysis Failure**: Packed variable analysis failed to partition sv into multiple *PVar* instances.
I: v[l′ : h′] ⟶ sv[l : h] | I ∈ S, sv ∈ *Var*, l′ ∈ *Int*, h′ ∈ *Int*, v ∈ V, l ∈ *Int*, h ∈ *Int* **Intermediate Partial Read**: Bytes [l′ : h′] of variable v hold bytes [l : h] of storage variable sv, loaded via I.

At a first approximation, the analysis merely tracks constant-offset additions and constant-mask boolean operations that the compiler outputs. The rules of this appendix are to some extent just tedious "work". However, we explicitly list the rules/patterns recognized for technical concreteness and completeness, especially for detail-oriented readers who may question what pattern recognition can reliably yield the results reported in the experiments in Section 7.

$$(\textsc{Base})\ \frac{I_L \mapsto \text{sv} \qquad I_L : [\text{v} := \text{LOAD}(\_)]}{I_L\text{: v}[0:31] \longrightarrow \text{sv}[0:31]}$$

$$(\textsc{RShift})\ \frac{I_L\text{: pv}[\text{s}:31] \longrightarrow \text{sv}[\text{l}:\text{h}] \qquad \text{v} := \text{RShift}(\text{pv}, \text{n})}{I_L\text{: v}[\max(\text{s}-\text{n}, 0):31] \longrightarrow \text{sv}[\text{l}+\text{n}-\text{s}:\text{h}]}$$

$$(\textsc{LShift1})\ \frac{\begin{array}{c} I_L\text{: pv}[\text{s}:31] \longrightarrow \text{sv}[\text{l}:\text{h}] \\ \text{v} := \text{LShift}(\text{pv}, \text{n}) \qquad \text{w} := 1+\text{h}-\text{l} \qquad \text{s}+\text{n}+\text{w} \le 32 \end{array}}{I_L\text{: v}[\text{s}+\text{n}:31] \longrightarrow \text{sv}[\text{l}:\text{h}]}$$

$$(\textsc{LShift2})\ \frac{\begin{array}{c} I_L\text{: pv}[\text{s}:31] \longrightarrow \text{sv}[\text{l}:\text{h}] \\ \text{v} := \text{LShift}(\text{pv}, \text{n}) \qquad \text{w} := 1+\text{h}-\text{l} \qquad \text{s}+\text{n}+\text{w} > 32 \end{array}}{I_L\text{: v}[\text{s}+\text{n}:31] \longrightarrow \text{sv}[\text{l}:\text{h}-(\text{s}+\text{n}+\text{w}-32)]}$$

$$(\textsc{LBMask})\ \frac{\begin{array}{c} I_L\text{: pv}[\text{s}:31] \longrightarrow \text{sv}[\text{l}:\text{h}] \\ \text{v} := \text{LowBytesMask}(\text{pv}, \text{m}) \qquad \text{s} < \text{m} \end{array}}{I_L\text{: v}[\text{s}:31] \longrightarrow \text{sv}[\text{l}:\min(\text{s}+\text{h}, \text{s}+\text{l}+\text{m}-1)-\text{s}]}$$

$$(\textsc{HBMask})\ \frac{I_L\text{: pv}[\text{s}:31] \longrightarrow \text{sv}[\text{l}:\text{h}] \quad \text{v} := \text{HighBytesMask}(\text{pv}, \text{m})}{\begin{array}{c} I_L\text{: v}[\max(32-\text{m}, \text{s}):31] \longrightarrow \\ \text{sv}[\max(\text{s}+\text{l}, \text{s}+\text{h}-\text{m}+1)-\text{s}:\text{h}] \end{array}}$$

$$(\textsc{BoolCast})\ \frac{I_L\text{: pv}[0:31] \longrightarrow \text{sv}[\text{b}:\text{b}] \qquad \text{v} := \text{BooleanCast}(\text{pv})}{I_L\text{: v}[0:31] \longrightarrow \text{sv}[\text{b}:\text{b}]}$$

Figure 14: Tracking storage variables through casts and shifts

A.1 Uses of Packed Variables

Figure 14 shows how the contents of a storage variable are tracked through sequences of shifting and casting operations. All rules are recursive, with the base case of the recursion being that an *intermediate* partial read fact "I_L: v[0 : 31] ⟶ sv[0 : 31]" is produced for each `LOAD` statement loading an index corresponding to storage variable sv.

When this computation reaches fixpoint, intermediate inferences are promoted to full partial read inferences based on the criteria shown in Figure 15. Rule Use1 will infer a partial read when the variable holding an intermediate inference is not cast or shifted further, while also ensuring it does not flow to a STORE statement through low-level casting and shifting operations, eliminating LOAD statements used in partial *write* patterns. The Use2 rule validates intermediate inferences held by variables used in high-level operations, while the Use3 rule increases completeness by validating intermediate inferences of *Var* subregions that have been used in other partial read operations.

A.2 Stores of Packed Variables

The above rules inferred use of packed variables from *uses* of the variables, i.e., from `LOAD` statements and subsequent patterns. Similar

$$\text{(Use1)}\ \frac{\begin{array}{c} I_L\colon \mathsf{v}[\_:\_] \longrightarrow \mathsf{sv}[\mathsf{l}:\mathsf{h}] \\ \nexists I_S : \left(I_S \Longmapsto \mathsf{sv} \quad I_S : \mathsf{STORE}(\mathsf{u}) := \_ \quad \mathsf{v} \rightsquigarrow \mathsf{u}\right) \\ \neg \_ := \mathsf{LShift}(\mathsf{v}, \_) \quad \neg \_ := \mathsf{RShift}(\mathsf{v}, \_) \\ \neg \_ := \mathsf{LowBytesMask}(\mathsf{v}, \_) \quad \neg \_ := \mathsf{HighBytesMask}(\mathsf{v}, \_) \end{array}}{I_L : \mathsf{v} := \mathsf{sv}[\mathsf{l}:\mathsf{h}]}$$

$$\text{(Use2)}\ \frac{I_L\colon \mathsf{v}[\_:\_] \longrightarrow \mathsf{sv}[\mathsf{l}:\mathsf{h}] \qquad \mathsf{HighLevelOpUse}(\mathsf{v})}{I_L : \mathsf{v} := \mathsf{sv}[\mathsf{l}:\mathsf{h}]}$$

$$\text{(Use3)}\ \frac{I_L\colon \mathsf{v}[\_:\_] \longrightarrow \mathsf{sv}[\mathsf{l}:\mathsf{h}] \qquad \_:\_ := \mathsf{sv}[\mathsf{l}:\mathsf{h}]}{I_L : \mathsf{v} := \mathsf{sv}[\mathsf{l}:\mathsf{h}]}$$

Figure 15: Inferring partial storage read inferences.

logic applies to the *definitions* of the variables, i.e., code that writes to a storage word via a `STORE` statement.

The $I_S, I_L : \mathsf{sv}[\mathsf{l}:\mathsf{h}] := \mathsf{v}$ "partial write" (by analogy to the earlier "partial read") is the result of the analysis tracking the stores of packed variables. We do not show the tedious rules explicitly, but briefly they track variables through the following low-level patterns:

(1) The contents of a storage variable sv are loaded by the I_L statement.
(2) The $[\mathsf{l}:\mathsf{h}]$ range of bytes is masked off, disregarding their contents.
(3) The new value, held in variable v, is shifted into the correct byte offset, if it happens to not already be at the correct offset.
(4) The shifted variable and the contents of the other variables occupying the same slot are combined using a bitwise `OR` operation.
(5) The result of the previous step is stored to sv in statement I_S.

It should be noted that, in optimized code, multiple nearby writes will be grouped, resulting in multiple partial write inferences for the same (sv, I_S, I_L) but different byte ranges, corresponding to different packed variables.

A.3 Inference aggregation

After computing the reads and writes of possibly packed storage *Var* instances, we aggregate their results to ensure they do not contain conflicting inferences.

Figure 16 captures the cases when this inference *fails*, i.e., when the analysis infers conflicting offsets for the same variable, or does not manage to infer any offsets for a variable. The partitioning analysis failure ($\not\downarrow$) predicate of Figure 16 is used negatively: if it does not apply, the packed variable analysis has been successful (for the specific variable being considered)—there is an inference of an offset inside a storage word and the offset is unique.

$$\text{(Conflict1)}\ \frac{\begin{array}{c} I : \mathsf{sv}[\mathsf{l}:\mathsf{h}] \quad I' : \mathsf{sv}[\mathsf{l}':\mathsf{h}'] \\ (\mathsf{l},\mathsf{h}) \neq (\mathsf{l}',\mathsf{h}') \quad (\mathsf{l},\mathsf{h}) \cap (\mathsf{l}',\mathsf{h}') \neq \emptyset \end{array}}{\not\downarrow \mathsf{sv}}$$

$$\text{(Conflict2)}\ \frac{I : \mathsf{sv}[\mathsf{l}:\mathsf{h}] \quad I' : \mathsf{sv}[\mathsf{l}':\mathsf{h}] \quad \mathsf{l} \neq \mathsf{l}'}{\not\downarrow \mathsf{sv}}$$

$$\text{(Missing)}\ \frac{\begin{array}{c} I : \mathsf{sv} \qquad \neg\, I : \_ := \mathsf{sv}[\_:\_] \\ \neg\, I, \_ : \mathsf{sv}[\_:\_] := \_ \qquad \neg\, \_, I : \mathsf{sv}[\_:\_] := \_ \end{array}}{\not\downarrow \mathsf{sv}}$$

Figure 16: Logic identifying conflicting inferences of *Var* sharing the same slot. If the failed variable partitioning inference ($\not\downarrow$) is not produced for a given storage variable sv it is considered successfully merged and all inferred partial reads and writes are matched with their corresponding *PVar* instances.