# The Incredible Shrinking Context... in a Decompiler Near You

SIFIS LAGOUVARDOS, University of Athens, Greece and Dedaub, Greece
YANNIS BOLLANOS, Dedaub, Greece
NEVILLE GRECH, Dedaub, Malta
YANNIS SMARAGDAKIS, University of Athens, Greece and Dedaub, Greece

Decompilation of binary code has arisen as a highly-important application in the space of Ethereum VM (EVM) smart contracts. Major new decompilers appear nearly every year and attain popularity, for a multitude of reverse-engineering or tool-building purposes. Technically, the problem is fundamental: it consists of recovering high-level control flow from a highly-optimized continuation-passing-style (CPS) representation. Architecturally, decompilers can be built using either static analysis or symbolic execution techniques.

We present Shrnkr, a static-analysis-based decompiler succeeding the state-of-the-art Elipmoc decompiler. Shrnkr manages to achieve drastic improvements relative to the state of the art, in all significant dimensions: scalability, completeness, precision. Chief among the techniques employed is a new variant of static analysis context: *shrinking context sensitivity*. Shrinking context sensitivity performs deep cuts in the static analysis context, eagerly "forgetting" control-flow history, in order to leave room for further precise reasoning.

We compare Shrnkr to state-of-the-art decompilers, both static-analysis- and symbolic-execution-based. In a standard benchmark set, Shrnkr scales to over 99.5% of contracts (compared to ∼95% for Elipmoc), covers (i.e., reaches and manages to decompile) 67% more code than Heimdall-rs, and reduces key imprecision metrics by over 65%, compared again to Elipmoc.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → *General programming languages*; • **Security and privacy** → **Software and application security**.

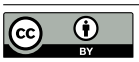
Additional Key Words and Phrases: Program Analysis, Smart Contracts, Decompilation, Datalog, Ethereum

**ACM Reference Format:**
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 (July 2025), 24 pages. https://doi.org/10.1145/3728935

## 1 Introduction

*Decompilation* or *lifting* from low-level binary code to a structured, high-level representation is a problem with a substantial history and practical significance in a variety of settings [13, 26, 27, 38]. In the context of programmable blockchains, decompilation has found a new application domain, with difficult technological considerations but intense demand. *Smart contracts* (the colloquial name for programs on a programmable blockchain) are deployed publicly and executed by-consensus of the entire network. Decompiling smart contracts is in high demand for several applications: building automated analyses over a uniform representation (regardless of the existence or not of source code for the smart contract); reverse-engineering security attacks (where source code is

Authors' Contact Information: Sifis Lagouvardos, University of Athens, Athens, Greece and Dedaub, Athens, Greece, sifis.lag@di.uoa.gr; Yannis Bollanos, Dedaub, Athens, Greece, ybollanos@dedaub.com; Neville Grech, Dedaub, Msida, Malta, me@nevillegrech.com; Yannis Smaragdakis, University of Athens, Athens, Greece and Dedaub, Athens, Greece, smaragd@di.uoa.gr.




ACM 2994-970X/2025/7-ARTISSTA060
https://doi.org/10.1145/3728935

unavailable); understanding competitive trading strategies by trading bots (where source code is unavailable); and much more.

The dominant binary platform for smart contracts is the Ethereum VM (EVM). It is the execution layer for most programmable blockchains, such as Ethereum, BSC, Arbitrum, Polygon, Optimism, Fantom, Base, Avalanche, and more. Accordingly, the problem of decompiling EVM bytecode has received significant attention [5, 8, 17, 20, 36, 47] and new entrants constantly vie for adoption—e.g., with the recent Heimdall-rs repo [5] rapidly reaching 1,000 stars and 100 forks.

From a technical standpoint, the problem of EVM decompilation is especially challenging. The EVM bytecode language is extremely low-level with respect to control flow, replacing all execution-control constructs with jump instructions to an address popped from the execution stack. That is, all control-flow statements (e.g., conditionals, loops, function calls, function returns) are translated into jumps to an address that is a run-time value of the low-level program. The challenge of EVM decompilation, thus, is to derive a higher-level representation, including functions, calls, returns, and structured control flow, from EVM bytecode. As a program analysis challenge, it has been the domain for applying several techniques. The primary distinction is between *symbolic execution* approaches [5, 36] and *static analysis* [17, 20] approaches. Symbolic-execution-based decompilers are easier to develop, naturally produce partial results (i.e., can always produce *something*, rather than failing on a smart contract in its entirety), yet are often vastly incomplete, failing to even discover a significant portion of the code. In contrast, static-analysis-based decompilers typically require much heavier development effort, cover (nearly) all code, but can fail to scale or can produce imprecise output.

In this work, we present a static-analysis-based decompiler that significantly advances the state of the art, on all quality dimensions (precision, completeness, scalability). Compared to Elipmoc [20], its predecessor and the leading static-analysis-based decompiler, our tool, Shrnkr, achieves much greater scalability (up to 99.7% on Elipmoc's evaluation dataset, compared to Elipmoc's 95.3%), while substantially improving precision and completeness—virtually nullifying imprecision or incompleteness for most metrics. Compared to the modern, most-adopted symbolic-execution-based decompiler, Heimdall-rs, Shrnkr exhibits a large advantage, decompiling up to 67% more binary statements. In essence, for complex contracts, Shrnkr succeeds in decompiling much of the interesting logic, while Heimdall-rs simply fails to find values to even cover the deepest statements via *one* path, let alone via all the different paths that can lead to such statements.

The technical essence of Shrnkr lies in several improvements over past static-analysis-based approaches: more precise and scalable static modeling, control-flow normalization via cloning, and pre-analysis-guided elimination of spurious calls. One key novelty is responsible for the lion's share of the benefit: the fundamental static model is improved, by use of a new kind of static *context* kept inside the decompiler. That is, the decompiler maintains as its current control-flow history (i.e., how the execution got to the currently-analyzed statement) a list of basic blocks that is updated under a different algorithm. The new logic, dubbed *shrinking context sensitivity*, aggressively *shrinks* the context when a likely matching call-return or chained-call pattern is observed.

In overview, the key contributions of this work consist of:

- a new algorithmic specification of context sensitivity, shrinking context sensitivity, suitable for the domain of EVM smart contract decompilation;
- an array of other techniques (block cloning, incomplete global pre-analysis to prepare the main analysis) that contribute to precision, completeness, and scalability;
- an experimental evaluation demonstrating substantial improvement over past decompilers in all interesting axes.

## 2 Background

We next introduce the setting of this work: smart contracts, decompilation, context sensitivity.

### 2.1 EVM Smart Contracts

Smart contracts are small programs (typically up to around 1,000 lines of high-level code, limited to 24KB in binary form in Ethereum) stored on a persistent blockchain as part of its state. They are typically written in a high-level programming language, with Solidity being by far the most widely used. Solidity, which is the setting of our work, dominates over all other languages in terms of adoption. At the high-level, a smart contract defines a set of external/public functions, which are its public entry points through which Externally-Owned Accounts (*EOA*s) or other smart contracts can interact with the contract, and a set of persistent *storage* variables which are part of the contract's state on the blockchain. Code reuse is facilitated through the use of *internal* (a.k.a. *private*) functions, inheritance, and library contracts. Solidity is a statically-typed language supporting operations on a number of value types (signed and unsigned integers, bytes, boolean), dynamic-length arrays, and associative mappings, as well as complex types combining the above.

The execution setting of smart contracts exhibits several intricacies, many of which are relevant to our discussion of bytecode analysis and decompilation. Performing transactions on the EVM requires a *gas* fee, paid in the chain's native token. This cost is (intended to be) analogous to the effort the blockchain's nodes need to perform, and I/O-heavy tasks (e.g., random access to blockchain state) are much more costly. As a result, a smart-contract compiler will typically optimize for two things: decreasing the executable bytecode's size, and reducing the runtime gas cost of its transactions. As a virtual machine, the EVM is powerful but simple and very low-level. It is a stack-based machine that supports arithmetic and logic operations over 256-bit (32-byte) words, has an implicitly persistent heap area (called *storage*), and a transient heap-like area (called *memory*). Types, objects, functions, closures, arrays, records, and any other high-level concepts are all translated away into word-level operations at the EVM level. This means that operations for most data types will require additional low-level code performing bit shifting or masking.

In the EVM, basic blocks are explicitly delineated, via `JUMPDEST` and `JUMP`/`JUMPI` (collectively: *jump*) instructions. The flow between blocks, however, is far from clear. The EVM's jump statements are inherently dynamic, reading the value of the target block from the stack.[1] Although most jump targets can be *resolved locally* (i.e., by looking at each basic block in isolation), the existence of *locally unresolved* dynamic jumps makes the computation of the control-flow-graph (CFG) an involved task. Each transaction involving a contract begins at statement `0x0` and goes on until a statement that halts execution is reached. The EVM offers no primitives for defining and calling functions, requiring the use of low-level code patterns to support public and private functions.

In addition, compiler version and settings greatly affect the produced bytecode. The release of Solidity v0.8.0 [65] introduced checked arithmetic and employed v2 of the ABI encoder, greatly increasing the number of internal functions. Since Solidity v0.8.13 [66], a new compilation pipeline became stable, with plans to make it the default in a future release [67]. This new *Yul/viaIR* pipeline involves Yul: a standardized, exportable intermediate language/representation (IR), which is also integrated into the Solidity language as inline assembly [11]. The Yul/viaIR pipeline enables deeper optimizations and more auditable code generation than the currently default "legacy" pipeline.

[1] In this paper, the term *block* refers to a *basic block*, as in standard compilers literature, i.e., a maximal sequence of low-level instructions always executed from start to finish. This has no connection to the "block" in "blockchain". Since our work does not involve distributed systems considerations, we never need to refer to the latter.

### 2.2 Context Sensitivity

Context sensitivity [15, 49, 58, 59, 61] has a long history in static program analysis, with work in over 3 decades. Every static analysis that computes the flow of abstract values through the program is trying to approximate the solution to an undecidable problem. As a result, it faces challenges with respect to both scalability and precision. Addressing such challenges requires careful design decisions and context-sensitivity has offered a good way to balance these needs.

Context-sensitivity associates program variables (and sometimes heap objects) with context information and distinguishes the values not just on the basis of variables but on the basis of variable+context combinations. Analysis inferences for multiple executions that result in the same context will be merged, but stay differentiated from inferences associated with different contexts. Call-site-sensitivity, i.e. using one or more previous call-sites as context information, has seen success in analyzing functional languages [59] and low-level imperative languages[15]. For object-oriented languages, the use of the receiver object(s) as context information has been the state-of-the-art context sensitivity abstraction since its introduction. Section 7 offers more detailed pointers and comparison with past work.

### 2.3 EVM Decompilation

We define the problem of EVM bytecode decompilation (a.k.a. *binary lifting*) as the derivation of high-level control-flow constructs and program structure from EVM bytecode. One can view the problem as the attempt to reconstruct a high-level program[2] from a low-level, stack-based intermediate representation (IR), where all control flow is represented in a *continuation-passing style (CPS)* form. For instance, a function call is done by pushing a continuation on the stack (the address of the basic block to return to), then pushing the function's entry block address, and jumping. All control-flow patterns, such as in-function branching, tail calls, calls in-sequence, passing a return value of a call as an argument to another, etc., are represented as complex sequences of pushing continuations and eventually jumping to the first.

In the setting of EVM decompilation, the dynamic nature of the `JUMP` operations creates the need for whole-program reasoning, in order to compute a program's control-flow graph. The Gigahorse/Elipmoc framework [17, 20] has addressed this need by introducing a global context-sensitive control-flow graph/points-to analysis as the backbone of its decompiler.

This context-sensitive global control-flow graph is then used by the Elipmoc framework [20], which we extend, to identify potential call-sites, which are in turn used to compute function boundaries. Lastly, after the function boundaries are computed, their number of arguments and return arguments are inferred.

The Gigahorse lifter employed a N-call-site (or jump-site) context-sensitivity algorithm, while Elipmoc introduced a composite approach that included the identity of the public entry point and the 8 last call-sites that are likely private function calls or returns. The evaluation of the Elipmoc publication highlighted the key importance the context-sensitivity algorithm plays in the decompiler's scalability and precision.

## 3 Motivation: Solidity to EVM by example

We next showcase various elements of binary-level EVM smart contracts as produced by the Solidity compiler. These motivate and provide important context for our later discussion.

[2]Notably, none of the EVM decompilers produce code that can be re-compiled. This does not diminish the value of EVM decompilation: the output of state-of-the-art decompilers is typically excellent both for human consumption and for writing automated program processing tools (e.g., static analyzers [62], symbolic-execution tools [22, 55], or program verification engines [3, 4, 24]).

### 3.1 Compiler Translation

To glimpse the low-level complexity of compiled smart contracts, we consider a simple example. The program of Figure 1 contains two external functions that accept various parameters and perform fund-transfer operations. As we discuss later, the expression `amt - defaultFee - feeA - feeB` on line 5 actually performs three function calls to a private function used to check subtraction for underflow.

```
1 interface IERC20 { function transfer(address to, uint256 value) external returns (bool); }
2 contract DecomptTest {
3   uint256 defaultFee;
4   function transWFee(address tok, address to, uint256 amt, uint256 feeA, uint256 feeB) external
5   { IERC20(tok).transfer(to, amt - defaultFee - feeA - feeB); /* 3 private function calls */ }
6   function simpleTransfer(address tok, address to, uint256 amt) external
7   { IERC20(tok).transfer(to, amt); }
8 }
```

Fig. 1. Simple smart contract, used as running example.

From the perspective of decompilation, the Solidity compiler is two different compilers, because of the aforementioned Yul/viaIR pipeline. The compiler effectively has two entirely separate code generation back-ends, which produce vastly different binary code patterns. Different optimization levels also greatly affect the binary program. Observable high-level metrics, such as the bytecode size, or internal metrics, such as the number of private/internal functions (which are not apparent in the final binary but are a key concept in the intermediate compiler representations) vary greatly, as shown for an example contract in the table below.

| Compiler Configuration | Bytecode Size | Number of Internal functions |
|---|---|---|
| legacy, no optimizer | 1,000 | 20 |
| viaIR, no optimizer | 1,195 | 43 |
| legacy, optimizer level 200 | 667 | 8 |
| viaIR, optimizer level 200 | 542 | 6 |

### 3.2 Public Function Patterns

As public functions are not inherent in the EVM, high-level languages adhere to the contract Application Binary Interface (ABI) [64], which specifies how input and output data are to be encoded when interacting with a smart contract. Per the ABI, the first 4 bytes provided in a smart contract invocation are the *function selector*, used to identify the public function being called.

Figure 2 shows the bytecode implementing the function selector logic for our example. These compiler-produced patterns implementing the function-selector logic have been used by past tools [2, 17, 20] to identify public function entries. However even detecting such simple patterns can have challenges. Looking at the previous code segment it is trivial for a local analysis to deduce that the `EQ` statement `0x24` operates on the function selector, since the input variable is loaded in the same block. `EQ` statement `0x2f` also checks the function selector, however an inter-block analysis is required to be able to deduce this. The state-of-the-art Elipmoc binary lifter requires computing the public function entries before performing its global control-flow graph analysis [20, Figure 4]. In order to achieve this, it uses an approximation based on a local-only analysis that does not verify that the selector is actually used.

In Section 5.2 we propose a 2-phase global analysis that, among others, tackles this problem.

```
0x1a: CALLDATALOAD
0x1b: PUSH1     0xe0
0x1d: SHR                   // function selector === calldataload(0) >> 28
0x1e: DUP1
0x1f: PUSH4     0x12e49406
0x24: EQ
0x25: PUSH2     0x38        // push function address of transWFee() public function
0x28: JUMPI
0x29: DUP1
0x2a: PUSH4     0x87d7a5f4
0x2f: EQ
0x30: PUSH2     0x54        // push function address of simpleTransfer() public function
0x33: JUMPI     ...
```

Fig. 2. Function selector logic for our example in Figure 1.

### 3.3 Private Function Patterns

The absence of native support for internal/private functions for the EVM has led to the emergence of low-level patterns to support code reuse.

```
0x12a: PUSH2  0x132  // push continuation address
0x12d: DUP3          // position argument in stack
0x12e: PUSH2  0x109  // push function address
0x131: JUMP
0x132: JUMPDEST      // continuation address
...
0x109: JUMPDEST      // cleanup_t_uint160 function
       address
0x10b: PUSH20  0xffffffffffffff...ffffffffffffffff
0x121: AND           // masks arg's upper 12 bytes off
0x124: SWAP2         // shuffles stack
0x127: JUMP          // jumps to continuation
```

(a) Simple Private Function Call

```
0x58: JUMPDEST
0x5a: PUSH2  0x77   // pushes final continuation
0x5d: PUSH1  0x84
0x5f: CALLDATALOAD
0x60: PUSH2  0x72   // pushes cont. for 3rd call
0x63: PUSH1  0x64
0x65: CALLDATALOAD
0x66: PUSH2  0x72   // pushes cont. for 2nd call
0x69: PUSH0
0x6a: SLOAD
0x6b: PUSH1  0x44
0x6d: CALLDATALOAD
0x6e: PUSH2  0x1c7  // push safeSub func address
0x71: JUMP
0x72: JUMPDEST      // cont. address for 2nd, 3rd call
0x73: PUSH2  0x1c7  // push safeSub func address
0x76: JUMP
0x77: JUMPDEST      ...
```

(b) Optimized Chained Private Function Calls

Fig. 3. Private Function Call Patterns

The basic pattern for private function calls, presented in Figure 3a, has been identified in past literature [17, 20, 29]. A basic block makes a call to internal function `cleanup_t_uint160` at offset `0x109`, having first pushed the bytecode offset at which it wants to return after the called function's execution completes. The return block of a function is a *locally unresolved* block that jumps back to the continuation block pushed by its caller.

In case of optimized code, a block performing an internal function call can also push the continuations of future calls. This pattern is typically produced in cases of calls that can be chained together (such as complex arithmetic expressions). As an example, consider the optimized compilation of expression '`amt - defaultFee - feeA - feeB`' from our example program. The subtraction operations are actually function calls, to a checked-subtraction function, so the expression should be thought of as '`safeSub(safeSub(safeSub(amt,defaultFee),feeA),feeB)`'.

The resulting bytecode can be seen in Figure 3b.

In this optimized case, block `0x58` will set the stack so that the 3 checked sub-operations are chained. Earlier work on the Elipmoc decompiler includes a function reconstruction algorithm [20,

Figure 6] that recursively infers these chained function calls. It assumes that each low-level block will map to one high-level function call. However, in optimized code, the same low-level block can be used to perform more than one high-level function call. This can be seen in the above example, which pushes the address of block `0x72` on the stack twice, implementing in this way the last two checked-subtraction operations. Effectively, if we think of our high-level code as '$\texttt{safeSub}_1\texttt{(safeSub}_2\texttt{(safeSub}_3\texttt{(amt,defaultFee),feeA),feeB)}$' then $\texttt{safeSub}_2$ and $\texttt{safeSub}_3$ are a *single* low-level instruction, the same for both calls.

Such complex patterns evade past function reconstruction algorithms. In Section 5.1 we propose a cloning-based technique that identifies such low-level blocks with different high-level uses, and clones them, recovering precision of decompilation output.

### 3.4 Our Context

Shrnkr is the third iteration of the Gigahorse/Elipmoc [17, 20] lifter framework, building on the foundation of the state-of-the-art Elipmoc [20] tool. Thus, Shrnkr is available as an open source tool on the public repository of the Gigahorse framework.[3]

Elipmoc's combination of scalability, precision, and completeness, paired with its expressive IR, have established it as a dominant lifter for EVM bytecode. Research tools for diverse program analysis applications have been implemented on top of Elipmoc. The applications include static-analysis [7, 19, 37, 39, 42, 46, 50, 62, 75, 76, 79], symbolic execution [22, 55], and deep learning [78].

Shrnkr's novel techniques mainly improve the decompiler's context-sensitive global control-flow graph with the introduction of the *shrinking context sensitivity* variant described in Section 4 and its tuning via incompleteness in Section 5.2. Section 5.1 describes the introduction of a block cloning transformation step, which is performed before the global analyses and helps Shrnkr produce normalized decompilation output.

Apart from the above, Shrnkr inherits Elipmoc's architecture, design decisions, and implementation with the exception of shallow fixes. The main components inherited from Elipmoc are its componentized local analyses, function reconstruction algorithms, and IR generation pipeline.

## 4 Shrinking Context-Sensitivity

We next present the main algorithmic techniques that help our tool, Shrnkr, drastically improve over the state of the art in EVM decompilation. Chief among them is *shrinking context sensitivity*, a new analysis context abstraction.

Past work [17, 20] has established a context-sensitive global control-flow-graph analysis as the backbone of a decompiler. That is, the decompiler abstractly simulates all possible executions of the decompiled program, but in a finite space: instead of keeping a full, unbounded execution stack, the decompiler collapses the stack into a finite *context* structure. That is, both the dynamic execution stack and the static context can be thought of as sequences of basic blocks, with the static context being a bounded sequence. The essence of the context-sensitivity algorithm is to decide *which* elements of the execution stack to keep at every point of modification, i.e., at every jump instruction. Different dynamic executions that have the same context (because their differing elements have been dropped by the context-sensitivity algorithm) will be treated the same, with the analysis computing all possible values for a variable, instead of just a single value.

As demonstrated by the Elipmoc work [20] the choice of context sensitivity algorithm greatly affects a decompiler's scalability and output quality. In contrast to the N-call-site sensitivity employed by Gigahorse [17], Elipmoc proposed a *transactional context-sensitivity* variant consisting of two parts: a sticky public function component, and a private function context including the $N$ latest

[3] https://github.com/nevillegrech/gigahorse-toolchain/tree/sub24

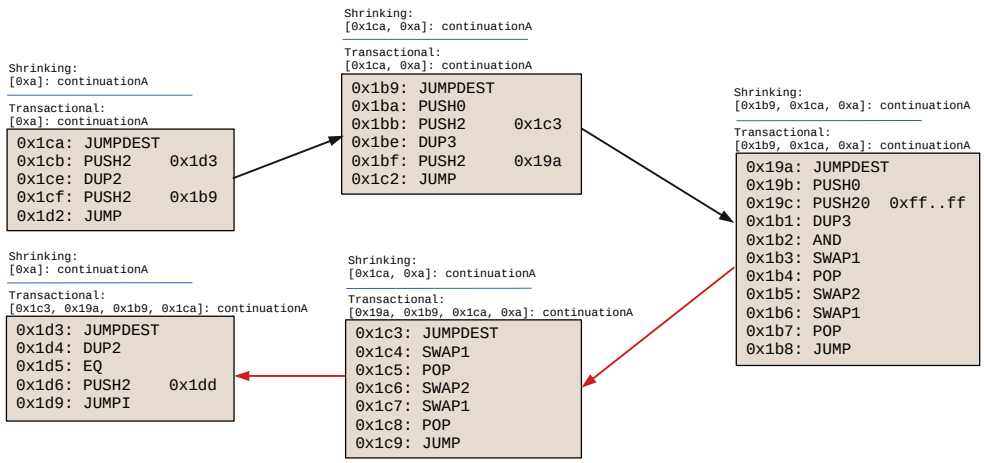



Fig. 4. Example: *Shrinking* context sensitivity contrasted (at each analyzed block) to the *Transactional* context sensitivity of past work. Both context sensitivity algorithms have a maximum context depth of 4. The public function components of both algorithms are omitted because they remain unchanged in the transitions shown. Arrows to the right are calls, arrows to the left returns. The analysis has initial information that should be kept precisely through the analyzed sub-graph: `continuationA` is applicable (e.g., it is kept in a certain stack location) if we reach the first analyzed block (`0x1ca`) with context `0xa`. Transactional context sensitivity forgets this information by the time it analyzes the last block: the context is merely the blocks shown in the figure, with no trace of how the analysis got to them. In contrast, shrinking context sensitivity maintains the information: the context shown at the last block captures how we got to the first block.

likely private function calls or returns. The publication's evaluation confirmed that both of the components of *transactional context-sensitivity* had a positive impact on scalability and precision.

Our approach, dubbed *shrinking context sensitivity*, retains the two-part approach with a key distinction: the private function context can *shrink* (much more drastically than merely discarding the oldest element), disregarding the context elements that are related to a likely private call, after that call returns to the first continuation pushed by its caller.

The example of Figure 4 helps explain the intuition behind *shrinking context sensitivity*. The example illustrates the effects of *shrinking context sensitivity*, contrasted with Elipmoc's *transactional context-sensitivity*, on the private function context, for a series of block transitions. Blocks `0x1ca` and `0x1b9` likely perform function calls as, following their execution, they leave the stack with continuations (`0x1d3` and `0x1c3`, respectively) pushed to it. For *shrinking context sensitivity*, when these continuations are reached, the most recent blocks, up to the block that pushed the continuation on the stack, are dropped. This leaves room to maintain other information, within the same maximum context depth: *shrinking context sensitivity* reaches block `0x1d3` with the same calling context as the initial block `0x1ca`, retaining crucial information about how the latter was reached. In contrast, *transactional context-sensitivity* reaches its maximum context depth and has to remove the oldest element from the stack, when it analyzes block `0x1d3`. This will mean that if block `0x1ca` was reachable under more than one calling context, upon reaching block `0x1d3`, *transactional context-sensitivity* is unable to differentiate these contexts, merging them all into one.

Figure 5 presents the definition of shrinking context sensitivity, in compact form. (A description in English follows shortly, and the reader may choose to consult it before referring to the formal definition.) The context-sensitivity definition is given in the form of the **Merge** context constructor.

$B$: set of basic blocks
$PC$: set of private contexts, $PC \cong B^n$
$C$: set of contexts, $C \cong B \times PC$, as labeled record [**pub:** $B$, **pri:** $PC$]

Initially, $ctx$ = [**pub:** Null, **pri:** []]

$$\textbf{Merge}([\textbf{pub: } u, \textbf{pri: } p], cur, next) = \begin{cases} [\textbf{pub: } next, \textbf{pri: } p], & \text{if } \texttt{PublicCall}(cur, next) \\ [\textbf{pub: } u, \textbf{pri: } [cur, \texttt{First}_{n-1}(p)]], & \\ \quad \text{if } \texttt{PrivateCallAndContinuation}(cur, *) & \\ \quad \text{or } (\texttt{PrivateReturn}(cur) & \\ \quad\quad \text{and } (\nexists c \in p\text{: } \texttt{PrivateCallAndContinuation}(c, next))) & \\ [\textbf{pub: } u, \textbf{pri: } \textbf{CutTo}(p, c)], & \\ \quad \text{if } \texttt{PrivateReturn}(cur) & \\ \quad\quad \text{and } (\exists c \in p\text{: } \texttt{PrivateCallAndContinuation}(c, next)) & \\ [\textbf{pub: } u, \textbf{pri: } p], \text{otherwise} & \end{cases}$$

Fig. 5. Context constructor for shrinking context sensitivity. For ease of exposition, we use labeled records to distinguish the public part of the context (single element) from the private part (of $n$ elements), instead of merging both in a flat tuple of $n + 1$ elements.

| | |
|---|---|
| `PublicCall`(*cur: B, next: B*) | Block transition is likely an entry to a public function. |
| `PrivateCallAndContinuation`(*caller: B, cont: B*) | The *caller* block likely makes a private function call after having pushed block *cont* as a continuation. |
| `PrivateReturn`(*cur: B*) | The current block likely returns from a private function call. |
| **CutTo**($p$: *PC*, $b$: *B*) = $p'$ | Truncating private context $p$ until encountering $b$ yields $p'$. |

Fig. 6. Auxiliary relations.

Namely, the value **Merge**([**pub:** $u$, **pri:** $p$], $cur$, $next$) gives the analysis context for basic block $next$ when the analysis finds an edge (i.e., a possible jump) from basic block $cur$ to $next$ and the current analysis context for block $cur$ is [**pub:** $u$, **pri:** $p$].

Figure 6 gives definitions for auxiliary relations that we refer to both in the context-sensitivity definition and in later logical specifications. It is important to note that function-inference predicates such as `PrivateCallAndContinuation` and `PrivateReturn` are only *likely* true to their name. The analysis cannot know for sure when a control-flow transition corresponds to a high-level function call. At this stage, the analysis can only, at best, grossly *over-approximate* what *might* be the possible calls and returns. (This over-approximation will contain many more edges than what will be eventually deemed to be function calls and returns.) However, the naming reflects the intuition: we want shrinking context sensitivity to attempt to match function calls and returns, and hopefully achieve both precision and scalability even with this incomplete information.

As a reminder, in predicate `PrivateCallAndContinuation`(*caller: B, cont: B*), the continuation does not have to be the block to return to after the call performed by block *caller* (as it would be in a straightforward, unoptimized compilation of a simple call). It can instead be the return block for the caller's caller (in case of tail calls), or the entry block of another called function (in case of

chained calls), or any other block determined by complex optimizing compilation patterns.
In English, the definition of Figure 5 states:

Upon a block transition,

- if a public function entry is found, enter it as the public part of the context; otherwise
- if a likely private function entry is found, push the caller block in the private part of the context, simultaneously dropping the oldest block in the context. Do the same if the transition is a likely function return that cannot be matched with a call in the context. Matching is done by comparing the continuation (that the earlier likely call has pushed) with the one the return is going to;
- if the block transition is a likely function return and can be matched with a call in the context, then drop all top-most private context elements until reaching the matching call;
- in all other cases the context is propagated as it is.

The intuition behind shrinking context sensitivity is deceivingly simple: static context abstracts away the dynamic execution stack of the EVM. It then stands to reason that when the dynamic execution returns from a function call, no record of the function entry should remain on the static context, much like in the dynamic stack. The analogy is not perfect, however. First, as discussed, call and return block transitions are far from certain. Second, in the dynamic execution stack, it is not the caller block of a function that is kept during the call, but only the continuation (i.e., the code where the function will return).

These two differences play into each other. The static analysis defensively keeps limited information to deal with natural uncertainty. (This uncertainty is due to not being certain about function call/return transitions but also due to the static loss of precision relative to dynamic execution, because of truncating state to a bounded size.) But it can drop information when it develops higher confidence: when the static analysis sees a *likely* call, it cannot be confident enough that it *is* a call and will eventually return, thus it keeps the called block in the context. When a matching return is found, however, the analysis confidence increases enough to remove not only the (likely) function call block but also all other blocks pushed on the context between the function entry and the return: these blocks are very likely intra-function control flow.

Truncating the context enables much greater precision later, since the context depth is finite (and would otherwise need to "forget" potentially valuable prior state about previous blocks that led to the current one). Additionally, the truncation logic offers a natural self-healing mechanism for the analysis abstraction of execution context: even if some inference (i.e., determining that a block may be a call and should thus be kept in the context) turns out to be noisy, it will likely be pruned when an enclosing function returns.

## 5 Other Enhancements

Shrnkr also integrates some secondary enhancements compared to the Elipmoc decompiler.

### 5.1 Control Flow Normalization via Cloning

Shrnkr performs aggressive cloning of blocks that are *locally* determined to be used in inconsistent ways. (*Local* inspection refers to inspection that does not require the full power of the decompiler's static analysis, i.e., the shrinking context of Section 4.)

The motivation for cloning has already been discussed with the private function reconstruction example of Section 3.3. (A detailed example and explanation can be found in the extended version of the paper [41], in Appendix A.) Effectively, the cloning transformation discovers blocks that are used as continuations in more than one case (i.e., by more than one push statement). These continuation blocks are often used to perform chained calls at different points in a contract's execution (as in the

example in Figure 3b). We encode our block-cloning instances as tuples of [pushStmt, blockToClone] and generate a new low-level block for each tuple. To reduce implementation complexity we only allow the cloning of low-level blocks that end with `JUMP` statements, having no fallthrough block that would need to be cloned as well.

### 5.2 Incomplete Global Pre-Analysis

Shrnkr leverages a *global pre-analysis* before the main decompilation step. The pre-analysis is a best-effort, *incomplete* version of the full context-sensitive control-flow graph analysis and we use it to configure the subsequent complete analysis in the following ways:

- We remove spurious `PublicCall` inferences by ensuring that the blocks identified by the detection of the local patterns presented in Figure 2 actually operate on a stack location holding the function selector bytes.
- We filter out spurious `PrivateCallAndContinuation` inferences by making sure that the locally identified *likely* private calls push a continuation block that is actually used as a target in a subsequent `JUMP` operation.
- We identify block transitions that lead to imprecision in the global control-flow graph analysis.

The first two cases help our analysis stay more precise and scalable by ensuring precious space in the public or private context components is not wasted on false inferences whose inclusion in the contexts give no additional precision benefits. The final case also helps reduce imprecision by identifying important edges not captured by our local heuristic rules.

Appendix B [41] provides additional details on the incomplete global pre-analysis.

## 6 Evaluation

The evaluation of Shrnkr intends to answer three distinct research questions:
*RQ1: Comparison with static-analysis-based decompilers* How does Shrnkr compare against the closest comparable state-of-the-art static-analysis-based decompiler?
*RQ2: Comparison with symbolic-execution-based decompilers* How does Shrnkr compare against the most popular symbolic-execution-based decompiler?
*RQ3: Design Decisions* How do the various technical components of Shrnkr (Sections 4, 5.1, 5.2) affect its results?

### 6.1 Experimental Setup

We perform the evaluation of Shrnkr using 2 experimental datasets:

*Standard Dataset.* The first dataset is that used in the publication and artifact for the state-of-the-art Elipmoc binary lifter. The dataset consists of [20]: *5,000 unique contracts, first deployed on the main Ethereum network between blocks 12,300,000 (April 24, 2021) and 13,300,000 (September 26, 2021).*

*Yul Dataset.* To investigate how the different tools do on the recently-released Yul/IR pipeline we introduce a new dataset consisting of 3,000 unique contracts compiled using the Yul/IR pipeline, deployed on the Ethereum mainnet until block 18,750,000 (Dec 09, 2023).

Although the Yul/IR pipeline is still used for a small minority of deployed smart contracts, it is likely to become more dominant in the future, especially after it becomes default. (Although becoming default does not immediately signify adoption: developers in this space are particularly sensitive to compilation settings and routinely override the defaults for deployment.) Furthermore, the Yul/IR pipeline is explicitly much harder for decompilers.[4]

[4]Cf. recent comments of Solidity lead developers: "*Decompilation is more complicated, yes*" and "*For decompilers it could be a problem*", https://youtu.be/3ljewa1__UM?t=921 .

Experimental runs are performed on a machine with 2 Intel Xeon Gold 6136 12 core CPUs and 754G of RAM. An execution cutoff of 200s was used for all tools. (This is over an order of magnitude higher than the average decompilation time of a contract. That is, if the decompiler does not finish in 200s, it is unlikely to ever finish, due to exponential explosion in the number of contexts, expressing failure to maintain precision.) For Shrnkr and Elipmoc we performed the experiments using 24 concurrent jobs, taking advantage of their out-of-the-box support for the parallel analysis of a set of contracts. We performed the Heimdall-rs runs sequentially as it lacks such support.

When performing the evaluation, we noticed that Heimdall-rs was often spending most of its execution time querying an online database to resolve the signatures of public methods via their function selector values. This results in a cosmetic-only improvement in the output, by showing high-level identifiers. Thus, in order to avoid disadvantaging Heimdall-rs, we used the `--skip-resolving` flag when invoking it. To match, we deleted all entries on the files Shrnkr and Elipmoc use for the resolution of public function signatures.

Our configuration of Shrnkr sets the maximum context depth of the *shrinking context sensitivity* to 20. In addition when we refer to Elipmoc's *transactional context sensitivity* we use a maximum depth of 8, as set in the Elipmoc publication. Elipmoc is largely unscalable with deeper context. Generally, these parameters are chosen as defaults by the respective tool authors because they are close to "experimentally optimal", so to speak. One can change them to improve some metric (e.g., higher values will improve precision), at the expense of others (incurring more timeouts).

### 6.2 Comparison with Elipmoc

Elipmoc [20] is the state-of-the-art research decompiler for EVM smart contracts. It has also seen industrial success by being the core of the infrastructure of the Dedaub Contract Library and Security Suite, available at https://app.dedaub.com/. The shared core of Shrnkr and Elipmoc allows us to perform an in-depth comparison. In all numbers shown in this section, **lower is better**. That is, precision, completeness, and scalability are evaluated via metrics of *imprecision*, *incompleteness*, and *lack of scalability*, respectively.

*6.2.1 Scalability.* Perhaps the topmost quality axis for a static-analysis-based decompiler is how often its static model scales well. (Without sacrificing precision, as confirmed later.) We compare the scalability of the two tools in Table 1.

Table 1. Timeouts of Shrnkr and Elipmoc. **Left table**: Standard Dataset, **Right table**: Yul Dataset.

| | Timeouts |
|---|---|
| Shrnkr | 13 (**0.26%**) |
| Elipmoc | 235 (**4.7%**) |
| Total | 5000 |

| | Timeouts |
|---|---|
| Shrnkr | 94 (**3.13%**) |
| Elipmoc | 379 (**12.63%**) |
| Total | 3000 |

Shrnkr vastly outscales Elipmoc in both datasets: For the Standard dataset, it manages to decompile nearly all contracts, with a timeout rate of just 0.26% versus Elipmoc's 4.7%. For the Yul dataset, the difference is again very significant with Shrnkr achieving 3 times fewer timeouts, at a timeout rate of 3.13%, compared to Elipmoc's 12.63%.

This gives us an initial confirmation on the difference of the two datasets and their underlying code generation pipelines. The newer, more powerful Yul/IR pipeline provides a significantly increased challenge to decompilers which we will also see confirmed later in this section.

Table 2 breaks down this performance by size class. As can be seen, for the largest contracts (15KB and above), Elipmoc very often fails. Shrnkr drops the timeout rates by a factor of 3 or more in all size classes.

Table 2. Shrnkr and Elipmoc's timeouts for each contract size class.
**Top table**: Standard Dataset. **Bottom table**: Yul Dataset.

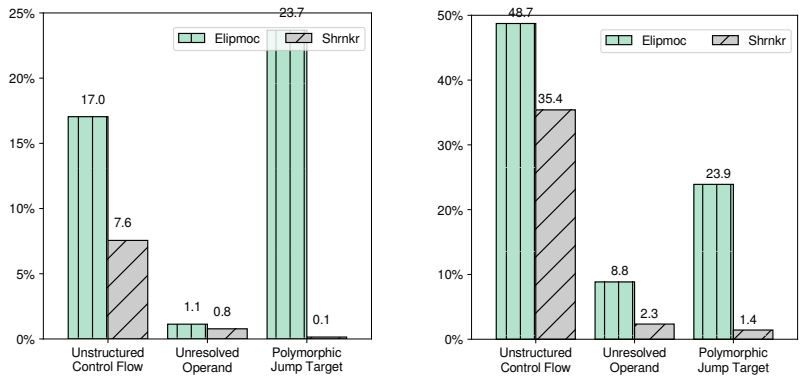
| Bytecode Size | [0,5KB) | [5KB,10KB) | [10KB,15KB) | [15KB,20KB) | [20KB,max) |
|---|---|---|---|---|---|
| Shrnkr | 2 (**0.08%**) | 4 (**0.38%**) | 2 (**0.31%**) | 1 (**0.34%**) | 4 (**0.91%**) |
| Elipmoc | 5 (**0.2%**) | 40 (**3.76%**) | 111 (**17.1%**) | 39 (**13.22%**) | 40 (**9.11%**) |
| Contracts in size class | 2552 | 1065 | 649 | 295 | 439 |

| Bytecode Size | [0,5KB) | [5KB,10KB) | [10KB,15KB) | [15KB,20KB) | [20KB,max) |
|---|---|---|---|---|---|
| Shrnkr | 2 (**0.17%**) | 12 (**1.66%**) | 14 (**3.35%**) | 29 (**8.68%**) | 37 (**10.22%**) |
| Elipmoc | 17 (**1.46%**) | 54 (**7.49%**) | 62 (**14.83%**) | 114 (**34.13%**) | 132 (**36.46%**) |
| Contracts in size class | 1165 | 721 | 418 | 334 | 362 |

*6.2.2 Precision.* To compare the precision of the two tools we employ the following precision metrics [20]:

Unresolved Operand: Missing operands in the output.
Unstructured Control Flow: High-level control flow in the output that is not expressible using structured programming constructs (e.g., high-level loops or conditionals).
Polymorphic Jump Target: (intra-procedural) Jump instructions with targets not uniquely resolved under the same context.

The percentages of contracts which exhibit these imprecision artifacts for the subset of contracts analyzed by both Shrnkr and Elipmoc are available in Figure 7. For all metrics, Shrnkr presents a clear improvement over Elipmoc. Inspecting the results of the "Polymorphic Jump Target" metric one can clearly notice that imprecision of the global control-flow-graph analysis has been nearly eliminated with 0.1% of the contracts having some imprecision compared to Elipmoc's 23.7% for the Standard dataset, with 1.4% and 23.9% respectively for the Yul dataset.




(a) Standard Dataset (b) Yul Dataset

Fig. 7. Precision Metrics in comparison with Elipmoc. All metrics show the % of contracts over the common contracts the 2 tools manage to decompile that exhibit the behavior measured—lower is better.

Notably, Figure 7 *downplays* the precision impact. If one takes the improvement in the cumulative value of each metric (instead of just the percentage of contacts that exhibit any non-zero amount

of imprecision) the effect is magnified. Table 3 presents these absolute numbers. For example, for the "Unstructured Control Flow" metric on the Yul dataset, Figure 7 shows a 27% decrease in the number of contracts with imprecision (48.7% to 35.4%). However, considering the absolute numbers, the total reduction of imprecision instances is nearly 59% (7,410 to 3,057).

Table 3. Analysis metrics for a comparison of Shrnkr and Elipmoc. The table unifies both precision and completeness metrics. **Top table**: Standard Dataset. **Bottom table**: Yul Dataset.

| | Polymorphic Jump Target | Missing Control Flow | Missing IR Block | Unresolved Operand | Unstructured Control Flow |
|---|---|---|---|---|---|
| Elipmoc | 4118 | 9411 | 1712 | 202 | 2253 |
| Shrnkr | 19 | 424 | 3 | 108 | 667 |

| | Polymorphic Jump Target | Missing Control Flow | Missing IR Block | Unresolved Operand | Unstructured Control Flow |
|---|---|---|---|---|---|
| Elipmoc | 2288 | 1221 | 2443 | 2116 | 7410 |
| Shrnkr | 74 | 145 | 1161 | 196 | 3057 |

*6.2.3 Completeness.* Static-analysis-based decompilers are nominally complete, i.e., cover all code. However, this is not a full guarantee, for two reasons. First, the decompiler will likely have a bound in the amount of work it performs, in order to minimize timeouts. Second, although each statement may be covered, not all execution paths may be covered.

To compare the completeness of the two tools, we use two incompleteness metrics:

Missing IR Block: Blocks that are reachable in the global CFG analysis but do not have any corresponding blocks in the three-address IR (TAC) output.

Missing Control Flow: Blocks in the TAC output that do not have the required number of outgoing edges (1 for non-return blocks, 2 for conditional jumps).

Both of these kinds of incompleteness artifacts arise due to the decompiler's inability to process the input context-sensitive control-flow graph, to produce a normalized decompilation output.

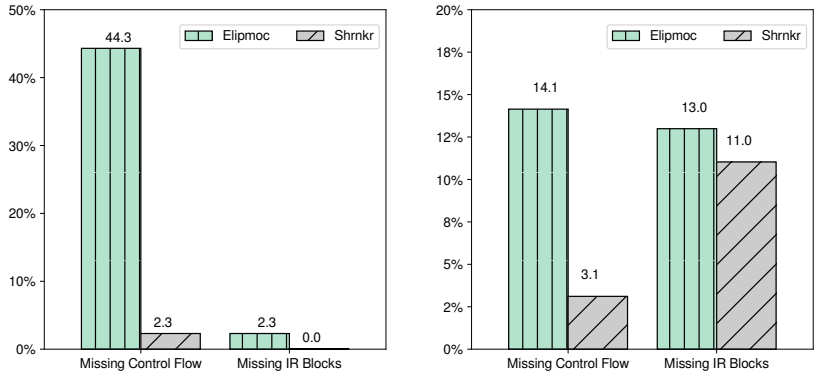
The percentages of contracts that exhibit these incompleteness artifacts are plotted in Figure 8 and the absolute counts are shown in Table 3.

As can be seen, Shrnkr significantly lowers incompleteness. The only metric that still exhibits non-negligible incompleteness artifacts is "Missing IR Blocks" and, although 11% of decompiled contracts in the Yul dataset have at least one such block, the absolute number of such missing blocks is tiny: just 0.11% of total recovered basic blocks.

### 6.3 Comparison with Heimdall-rs

Heimdall-rs [5] is an increasingly-popular symbolic-execution-based decompiler. It has received significant attention in the past year, and its GitHub repository has surpassed 100 forks and 1,000 stars in a brief time. The primary objective of Heimdall-rs is to serve as a precise and performant decompilation toolkit.

With symbolic execution being the backbone of Heimdall-rs, the decompilation leverages the symbolic representation of a program's execution traces in order to produce a higher-level program representation. This approach enables the decompiler to reason over actual execution paths, resulting in highly-precise decompilation results. However, since it is only feasible to utilize a limited number of execution sequences, symbolic-execution-based methods typically yield incomplete results, capturing fewer program behaviors overall.




(a) Standard Dataset (b) Yul Dataset

Fig. 8. Completeness Metrics in comparison with Elipmoc. All metrics show the % of contracts over the common contracts the 2 tools manage to decompile that exhibit the behavior measured—lower is better.

Due to the fundamentally different architecture of Shrnkr and Heimdall-rs, we cannot directly compare detailed internal metrics for each tool, as in the comparison with Elipmoc. However, we can compare user-level, end-to-end metrics. Specifically, we compare the number of *unique* CALL and LOGx *signatures* in the decompiled code. These are code elements that should undisputedly exist in a correct decompilation: they are the static *signatures* of functions called on *external* contracts (encoded in the bytecode as the 4-byte hash of the function name and argument types—e.g., `0x0001e862` for "`balanceOfAt(uint256,address)`") and of events emitted for consumption by external, off-chain code (encoded as a 32-byte hash of a similar signature). Capturing (in decompiled code) these unique signatures is a completeness/coverage metric over possible contract behaviors with regards to external calls and events. Although simple, the metric has the property of being indifferent to different decompilation styles (esp. inference of private functions by Shrnkr vs. inlining of all code/logic by Heimdall-rs).

Table 4 shows the number of CALL and LOGx signatures that are identified by each tool. (In these completeness numbers, **higher is better**.) Shrnkr manages to discover 67% more calls (13,998 calls compared to Heimdall-rs's 8,381) in the standard dataset and 38% more calls (13,600 calls against 9,841 for Heimdall-rs) in the Yul dataset. Since the numbers provide an estimate of how much more code is decompiled by Shrnkr when compared against Heimdall-rs, the results demonstrate the large advantage of Shrnkr over Heimdall-rs in terms of completeness. A similar conclusion may be drawn by looking at the events metrics.

Table 5 breaks down these results by contract size. As can be seen, the completeness benefit is very substantial in large contracts, leading nearly to a doubling of event and function signatures observed in the output code. It is reasonable to expect that larger contracts have a higher need for automatic analysis: they are both harder to analyze manually and involve more sophisticated code patterns. Therefore, any verifiable advantage in completeness holds large practical value.

Table 4 also shows average execution time and repeats the Shrnkr timeout rate. It is apparent that, in terms of scalability, Heimdall-rs has no hurdle to overcome, as expected in a symbolic execution tool, which covers the program only to the extent that it can execute it precisely. Heimdall-rs has no timeouts and is extremely fast on average. The Heimdall-rs average execution time can be more than 10 times smaller than Shrnkr—although the average times are low for both tools, at under 7s for the slowest dataset.

Table 4. Total number of identified CALL and LOGx signatures between Shrnkr and Heimdall-rs.
**Top table**: Standard Dataset. **Bottom table**: Yul Dataset.

| | Unique External Calls | Unique Events | Avg. Time | Timeouts |
|---|---|---|---|---|
| Shrnkr | 13998 | 12725 | 1.87s | 13 |
| Heimdall-rs | 8381 | 9345 | 0.88s | 0 |

| | Unique External Calls | Unique Events | Avg. Time | Timeouts |
|---|---|---|---|---|
| Shrnkr | 13600 | 13661 | 6.76s | 94 |
| Heimdall-rs | 9841 | 9505 | 0.59s | 0 |

Table 5. Shrnkr and Heimdall-rs's sigs for each contract size class. The number of contracts per size class is slightly smaller than in Table 2 because timeouts are excluded.
**Top table**: Standard Dataset. **Bottom table**: Yul Dataset

| Bytecode Size | [0,5KB) | [5KB,10KB) | [10KB,15KB) | [15KB,20KB) | [20KB,max) |
|---|---|---|---|---|---|
| Shrnkr function sigs | 2343**(+37%)** | 3245**(+58%)** | 3127**(+89%)** | 2259**(+65%)** | 3024**(+88%)** |
| Heimdall-rs function sigs | 1699 | 2051 | 1654 | 1371 | 1606 |
| Shrnkr event sigs | 1995**(+15%)** | 3050**(+19%)** | 2629**(+31%)** | 1479**(+36%)** | 3572**(+80%)** |
| Heimdall-rs event sigs | 1729 | 2554 | 1994 | 1084 | 1984 |
| Contracts in size class | 2550 | 1061 | 647 | 294 | 435 |

| Bytecode Size | [0,5KB) | [5KB,10KB) | [10KB,15KB) | [15KB,20KB) | [20KB,max) |
|---|---|---|---|---|---|
| Shrnkr function sigs | 2254**(+19%)** | 3197**(+33%)** | 2384**(+39%)** | 2296**(+40%)** | 3469**(+57%)** |
| Heimdall-rs function sigs | 1883 | 2397 | 1712 | 1643 | 2206 |
| Shrnkr event sigs | 1451**(+35%)** | 3078**(+27%)** | 2727**(+31%)** | 2340**(+57%)** | 4065**(+65%)** |
| Heimdall-rs event sigs | 1068 | 2410 | 2077 | 1487 | 2463 |
| Contracts in size class | 1163 | 709 | 404 | 305 | 325 |

Overall, the results are indicative of the completeness superiority of a static analysis when compared against symbolic execution as the underlying technique for decompilation. Arguably, the very essence of a decompiler is to lift as much low-level code as possible. Thus, *every* part of the program to be decompiled should be considered, and while symbolic execution makes an efficient implementation simpler from an engineering standpoint, the completeness offered by a deep static analysis appears to be unparalleled.

### 6.4 Human Study

To provide deeper insights in our comparison with Elipmoc and Heimdall-rs, we complement the quantitative evaluation of the two previous subsections with a small-scale human study, assessing the decompilation quality of the outputs of the 3 tools.

To produce source-like high-level output (instead of the usual TAC output of Shrnkr) we ported the source unparser of the Dedaub Security Suite to use Shrnkr.

We directed the study to expert participants working in the industry and/or academia in roles related to smart contract security. To incentivize participation, we offered participants a $100 reward. Overall, 8 participants completed the study, each given 3 randomly assigned decompilation tasks, resulting in a total of 24 data points.

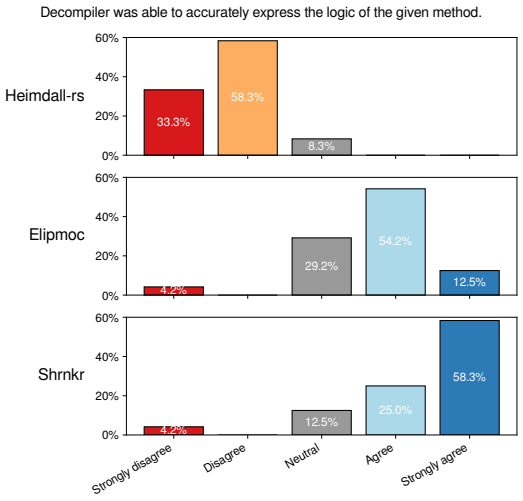



Fig. 9. Human Study Results: Participant agreement with statement *"Decompiler was able to to accurately express the logic of the given method."*

More specifically, for each decompilation task the participant was given the outputs of the three tools as well as the original source code and was tasked with evaluating the decompilers' ability to accurately recover the logic of the contract's largest public method (which was identified programmatically and provided to the participants).

To minimize participant bias we anonymized the outputs of the three tools and prompted the participants to *"attempt to ignore the differences in decompilation style such as the inference of private functions vs the inlining of all code, naming conventions, etc."*. The entirety of the study is available in Appendix C [41].

Figure 9 plots the results of the human study. Heimdall-rs is shown to perform the worst in the human study, with its incomplete algorithms failing to recover the logic of the given programs. While both Shrnkr and Elipmoc have a generally positive performance with the majority of participants rating their decompilation ability positively, Shrnkr's improvements manage to shift the participant concensus from "Agree" to "Strongly Agree", highlighting their usefulness.

### 6.5 Case Study: Decompilation of Hack Contracts

As a case study, we examine the ability of Shrnkr, Elipmoc, and Heimdall-rs to decompile adversarial contracts, used in past security attacks ("hacks"). The goal is to confirm that there is no obvious negative bias in this subset of contracts, at least with respect to scalability. The case study is performed on the `malicious_smart_contracts` dataset [5] from the labelled-datasets repository of OpenZeppelin's Forta Network. While the original dataset contains 753 smart contract addresses, it contains many duplicate bytecodes which we removed, ending up with a dataset of 592 unique contracts.

Table 6. Runtime statistics for the adversarial dataset.

| | Avg. Time | Timeouts |
|---|---|---|
| Shrnkr | 0.79s | 0 |
| Elipmoc | 0.67s | 10 (**1.69%**) |
| Heimdall-rs | 0.44s | 0 |

Table 6 presents the runtime statistics for the adversarial contracts dataset. As can be seen, these contracts represent a smaller challenge than our other two datasets, with Shrnkr and Heimdall-rs managing to decompile all 592 contracts and Elipmoc timing out at only 1.69% percent of contracts. This result can be explained by the relative simplicity of hack contracts: regular contracts, intended to be used heavily, pack as much code as possible and employ heavy optimization, whereas adversarial (hack) contracts are both smaller (i.e., do not struggle to fit in the EVM 24KB limit) and less mature. Accordingly, in terms of decompilation completeness, in the comparison between Shrnkr and Heimdall-rs, the story of Table 4 stands: Shrnkr recovers 60.5% more external call signatures (1106 vs 689) and 9.4% more event signatures (1049 vs 959).

[5] https://github.com/forta-network/labelled-datasets/blob/main/labels/1/malicious_smart_contracts.csv

### 6.6 Design Decisions

In order to understand how each of the features of Shrnkr affect its scalability, precision, and completeness, we decompiled our datasets using 3 modified configurations of Shrnkr, in addition to its default one:

- No Shrinking ctx: Shrnkr replacing its *shrinking context sensitivity* with the *transactional context sensitivity* of Elipmoc.
- No Cloning: Shrnkr with the cloning transformation we presented in Section 5.1 disabled.
- No Pre-Analysis: Shrnkr with the incomplete global pre-analysis we presented in Section 5.2 disabled.

***Scalability.*** Table 7 shows the timeouts for the various different configurations of Shrnkr. (In the metrics of this section, **lower is better**.)

Table 7. Timeouts for Shrnkr configurations. **Left table**: Standard Dataset. **Right table**: Yul Dataset.

| | Timeouts |
|---|---|
| No Shrinking ctx | 559 (**11.18%**) |
| No Cloning | 16 (**0.32%**) |
| No Pre-Analysis | 19 (**0.38%**) |
| Shrnkr | 13 (**0.26%**) |
| Total contracts in dataset | 5000 |

| | Timeouts |
|---|---|
| No Shrinking ctx | 403 (**13.43%**) |
| No Cloning | 105 (**3.5%**) |
| No Pre-Analysis | 146 (**4.86%**) |
| Shrnkr | 94 (**3.13%**) |
| Total contracts in dataset | 3000 |

The table makes clear that the scalability of Shrnkr is due to the *shrinking context sensitivity*. Disabling it leads to 35x the timeouts for the Standard dataset and over 4x the timeouts on the Yul dataset. In addition, the more demanding Yul dataset allows us to observe that all 3 of the components of Shrnkr have a positive impact on scalability. Disabling the incomplete global pre-analysis leads to a 53% increase in timeouts while disabling the cloning transformation leads to a 10% increase.

***Precision and Completeness.*** Table 8 presents the absolute numbers for our precision and completeness metrics for the four configurations of Shrnkr.

It is easy to see that the replacement of the *shrinking context sensitivity* with Elipmoc's *transactional context sensitivity* produces a much less precise analysis. This imprecision first manifests

Table 8. Analysis metrics for various configurations of Shrnkr. The table unifies both precision and completeness metrics. **Top table**: Standard Dataset. **Bottom table**: Yul Dataset

| | Polymorphic Jump Target | Missing Control Flow | Missing IR Block | Unresolved Operand | Unstructured Control Flow |
|---|---|---|---|---|---|
| No Shrinking ctx | 3406 | 459 | 751 | 128 | 611 |
| No Cloning | 18 | 728 | 153 | 140 | 1042 |
| No Pre-Analysis | 40 | 421 | 3 | 62 | 542 |
| Shrnkr | 19 | 421 | 3 | 62 | 542 |

| | Polymorphic Jump Target | Missing Control Flow | Missing IR Block | Unresolved Operand | Unstructured Control Flow |
|---|---|---|---|---|---|
| No Shrinking ctx | 3381 | 253 | 1474 | 333 | 3624 |
| No Cloning | 34 | 168 | 1602 | 504 | 6568 |
| No Pre-Analysis | 101 | 121 | 956 | 184 | 2927 |
| Shrnkr | 65 | 120 | 956 | 184 | 2927 |

itself at the global context-sensitive control-flow graph level and also results in decompilation artifacts at the three-address-code output.

Disabling the cloning component has the biggest negative impact to the precision of Shrnkr's decompilation output with 92% more "Unstructured Control Flow" inferences for the Standard dataset and 124% for the Yul dataset.

Lastly, disabling the incomplete global pre-analysis results in imprecision for the global control-flow graph analysis, without affecting the precision of three-address-code output.

Inspecting the completeness metrics, we can deduce that the *shrinking context sensitivity* and block cloning techniques have the largest impact on the completeness of Shrnkr.

## 7 Related Work

Multiple EVM decompilers have been proposed over the years [8, 17, 36, 68]. However, continuous technical advancements are needed to keep up with the complexity of modern smart contracts, therefore many decompilers (even past leaders) have been not been maintained [8, 36, 68], delivering very poor results by more modern standards [17, 20]. Most relevant to our work, Gigahorse [17], Elipmoc [20], and Heimdall-rs have been discussed extensively throughout the paper. Other decompilers used by practitioners include EtherVM [1], and, indirectly, the decompiler in Certora [25, 56]. More recent decompilers used for static analysis clients include Ethersolve [14]. However, [14] only raises the abstraction level to a global CFG, which requires a small subset of the techniques developed within Shrnkr. For instance, Ethersolve does not produce a register-based IR (retaining the original stack-altering instructions in its output) nor does it discover private functions. These limitations inhibit its ability to support high-level client analyses.

A number of other EVM toolchains are used today, and several studies [10, 54, 77] have examined their usefulness and real-world impact. Among them, popular for finding vulnerabilities, are fuzzing frameworks [12, 28, 34, 72], which identify vulnerabilities by analyzing bytecode directly. Notable examples include ContractFuzzer [34], Harvey [72], Echidna [21, 23], sFuzz [51], and the recent Ityfuzz [60], which leverages a faster interpreter (RETH) for improved performance. Additionally, are a number of tools, which are meant to analyze the 3-address IR output that Shrnkr provides, including MadMax [18], Ethainter [7], Greed [22, 55], DeepInfer [78], and Todler [50].

Outside of the smart contract domain, a number of tools and techniques are relevant. Context sensitivity has been employed in many static analysis settings before, and is well-known for improving precision for value-flow analysis in languages with dynamic dispatch [49, 53, 58, 61]. Selective context sensitivity approaches [30–33, 43, 48, 52, 63, 69, 70] have been proposed to overcome the scalability and precision obstacles faced when applying traditional context sensitivity [49, 59, 61] variants to large, real-world programs. Much past selective context sensitivity research [43–45, 52, 63] has relied on the results of a pre-analysis to create context sensitivity variants that achieve balance between scalability and precision. Such work [43–45, 63, 70] often makes use of an imprecise context-insensitive pre-analysis, which is not always ideal when attempting to approximate the behavior of a context-sensitive analysis. *Shrinking context sensitivity* does not base its decisions on such a less-precise pre-analysis.

In some of the aforementioned work [30–33], selective context sensitivity has also been fruitfully combined with machine learning approaches. In [31] authors introduced the technique of *context tunneling* to create context sensitivity variants that, upon a transition, in some cases choose to update the calling context and in others to simply propagate it. Context tunneling has shown great promise in the analysis of Java applications, having been used [32] to almost completely simulate object sensitivity via call-site sensitivity. Our *shrinking context sensitivity* (as well as Elipmoc's *transactional context sensitivity*) also employs similar logic to just propagate (instead of updating) the calling contexts in most transitions.

An important distinction relative to Shrnkr is that all such past work (in selective context sensitivity) limits the context by *avoiding* to include context elements, in advance. For instance, the description of novelty of the Bean technique [70] reads: *The novelty lies in identifying redundant context elements [...] based on a pre-analysis (e.g., a context-insensitive Andersen's analysis) performed initially on a program, and then avoid them in the subsequent k-object-sensitive analysis.* All tunneling work follows a similar pattern.

Instead, the distinguishing feature of *shrinking context sensitivity* is that it has a temporal character: it first includes context elements, while they are useful, but later eliminates them eagerly, i.e., before the maximum context depth is reached. No past algorithm has this feature.

It is not entirely surprising that past context-sensitive algorithms have not explored this direction. Past context-sensitive analyses have mainly worked in the setting of points-to analysis of large Java programs. The context depth employed in such a setting is much shorter and does not lend itself to more adaptive algorithms. For instance, the typical context depth in points-to analysis work [43, 52, 63, 69, 70] is just 2. Shrinking context sensitivity is applied with contexts of depth around 20. This showcases that decompilation is a much higher-precision setting (but for very specific kinds of information). This large context depth is a key enabler of shrinking context sensitivity: it means the context includes elements all the way from a function's call to its return, even if the function itself makes many other nested calls.

Binary disassembly [6, 16, 38] and decompilation [9, 13, 35, 71, 73, 74] techniques have seen use in several domains. Numerous foundational techniques had been established by the mid-1990s [13], with particular emphasis on the x86 architecture. This architecture offers a somewhat simplified path to decompilation, given a dependable disassembly process. The delineation of function boundaries and the deduction of arguments are facilitated by the adherence to standard calling conventions, the Instruction Set Architecture's (ISA) support for function calls and returns, and a uniform call stack architecture. More closely aligned with the techniques of our work, the Ddisasm tool [16] uses Datalog to provide a disassembler for x64 binaries, while the OOAnalyzer system [57] employs a logic programming approach with Prolog to infer C++ class structures from compiled binaries.

## 8 Conclusion

We presented Shrnkr, a static-analysis-based decompiler for EVM bytecode. Shrnkr achieves a significant improvement over the state-of-the-art using a new variant of context sensitivity, *shrinking context sensitivity*, additionally tuned via an incomplete global pre-analysis, and a cloning transformation to better normalize decompilation output. These three advancements enable Shrnkr to vastly outscale the state-of-the-art decompiler, while also seeing significant improvements in both precision and completeness. Shrnkr was also compared against the most popular alternative-technology decompiler displaying superior coverage of program behaviors. We perform our evaluation on datasets of contracts using the two pipelines of the Solidity compiler: the currently default "legacy" pipeline, and the new Yul pipeline. The latter had not been considered in the evaluations of previous publications, and we experimentally show it to provide a greater challenge to decompilers.

## Acknowledgments

We thank the study participants for taking the time to complete the study. We gratefully acknowledge funding by ERC Advanced Grant PINDESYM (101095951).

## Data-Availability Statement

Shrnkr is part of the gigahorse-toolchain open source repository. The paper's artifact is also publicly available [40] and can be used to reproduce all experiments in the paper's evaluation except for the human study in Section 6.4, which uses closed-source code.

## References


[1] 2018. Online Solidity Decompiler. http://ethervm.io/decompile

[2] 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 (ATVA)*. Springer.

[3] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2020. Taming callbacks for smart contract modularity. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 209 (nov 2020), 30 pages. https://doi.org/10.1145/3428277

[4] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2023. Relaxed Effective Callback Freedom: A Parametric Correctness Condition for Sequential Modules With Callbacks. *IEEE Transactions on Dependable and Secure Computing* 20, 3 (2023), 2256–2273. https://doi.org/10.1109/TDSC.2022.3178836

[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] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. 2016. Speculative Disassembly of Binary Code. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (Pittsburgh, Pennsylvania) *(CASES '16)*. Association for Computing Machinery, New York, NY, USA, Article 16, 10 pages. https://doi.org/10.1145/2968455.2968505

[7] 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

[8] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. arXiv:1809.03981 [cs.PL]

[9] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 353–368. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/schwartz

[10] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. 2024. Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners?. 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 60, 13 pages. https://doi.org/10.1145/3597503.3623302

[11] 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. https://doi.org/10.1145/3563328

[12] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 227–239. https://doi.org/10.1109/ASE51524.2021.9678888

[13] Cristina Cifuentes. 1994. *Reverse compilation techniques.* Ph. D. Dissertation. Queensland University of Technology. https://eprints.qut.edu.au/36820/ Presented to the School of Computing Science, Queensland University of Technology..

[14] Filippo Contro, Marco Crosara, Mariano Ceccato, and Mila Dalla Preda. 2021. EtherSolve: Computing an Accurate Control-Flow Graph from Ethereum Bytecode. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. 127–137. https://doi.org/10.1109/ICPC52881.2021.00021

[15] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation* (Orlando, Florida, United States). 242–256.

[16] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog Disassembly. , 1075–1092 pages. https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya

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

[18] 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). https://doi.org/10.1145/3276486

[19] 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* (Nov. 2020).

[20] 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

[21] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 557–560. https://doi.org/10.1145/3395363.3404366

[22] 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

[23] Alex Groce and Gustavo Grieco. 2021. echidna-parade: a tool for diverse multicore smart contract fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 658–661. https://doi.org/10.1145/3460319.3469076

[24] 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 Programming Languages* 2, POPL, Article 48 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158136

[25] 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. https://doi.org/10.1145/3689796

[26] James Hamilton and Sebastian Danicic. 2009. An Evaluation of Current Java Bytecode Decompilers. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '09)*. IEEE Computer Society, Washington, DC, USA, 129–136. https://doi.org/10.1109/SCAM.2009.24

[27] Nicolas Harrand, C'esar Soto-Valero, Martin Monperrus, and Benoit Baudry. 2019. The Strengths and Behavioral Quirks of Java Bytecode Decompilers. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 92–102. https://arxiv.org/pdf/1908.06895.pdf

[28] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. ACM, New York, NY, USA, 531–548. https://doi.org/10.1145/3319535.3363230

[29] Jiahao He, Shuangyin Li, Xinming Wang, Shing-Chi Cheung, Gansen Zhao, and Jinji Yang. 2023. Neural-FEBI: Accurate function identification in Ethereum Virtual Machine bytecode. *Journal of Systems and Software* 199 (2023), 111627. https://doi.org/10.1016/j.jss.2023.111627

[30] Minseok Jeon, Sehun Jeong, Sungdeok Cha, and Hakjoo Oh. 2019. A Machine-Learning Algorithm with Disjunctive Model for Data-Driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 41, 2, Article 13 (jun 2019), 41 pages. https://doi.org/10.1145/3293607

[31] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and scalable points-to analysis via data-driven context tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 140 (oct 2018), 29 pages. https://doi.org/10.1145/3276510

[32] Minseok Jeon and Hakjoo Oh. 2022. Return of CFA: call-site sensitivity can be superior to object sensitivity even for object-oriented programs. *Proc. ACM Program. Lang.* 6, POPL, Article 58 (jan 2022), 29 pages. https://doi.org/10.1145/3498720

[33] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 100 (oct 2017), 28 pages. https://doi.org/10.1145/3133924

[34] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/3238147.3238177

[35] D. S. Katz, J. Ruchti, and E. Schulte. 2018. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 346–356.

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

[37] Queping Kong, Jiachi Chen, Yanlin Wang, Zigui Jiang, and Zibin Zheng. 2023. DeFiTainter: Detecting Price Manipulation Vulnerabilities in DeFi Protocols. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1144–1156. https://doi.org/10.1145/3597926.3598124

[38] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13* (San Diego, CA) *(SSYM'04)*. USENIX Association, USA, 18.

[39] Sifis Lagouvardos, Yannis Bollanos, Michael Debono, Neville Grech, and Yannis Smaragdakis. 2025. Precise Static Identification of Ethereum Storage Variables. arXiv:2503.20690 [cs.PL] https://arxiv.org/abs/2503.20690

[40] Sifis Lagouvardos, Yannis Bollanos, Neville Grech, and Yannis Smaragdakis. 2025. *The Incredible Shrinking Context... in a Decompiler Near You (Artifact)*. https://doi.org/10.5281/zenodo.15189969

[41] Sifis Lagouvardos, Yannis Bollanos, Neville Grech, and Yannis Smaragdakis. 2025. The Incredible Shrinking Context... in a Decompiler Near You (Extended Version). arXiv:2409.11157 [cs.PL] https://arxiv.org/abs/2409.11157

[42] 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

[43] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 141 (oct 2018), 29 pages. https://doi.org/10.1145/3276511

[44] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/3236024.3236041

[45] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 42, 2, Article 10 (may 2020), 40 pages. https://doi.org/10.1145/3381915

[46] Zeqin Liao, Zibin Zheng, Xiao Chen, and Yuhong Nan. 2022. SmartDagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 752–764. https://doi.org/10.1145/3533767.3534222

[47] Xia Liu, Baojian Hua, Yang Wang, and Zhizhong Pan. 2023. An Empirical Study of Smart Contract Decompilers. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 1–12. https://doi.org/10.1109/SANER56733.2023.00011

[48] Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 148 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360574

[49] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (2005), 1–41.

[50] Sundas Munir and Christoph Reichenbach. 2023. TODLER: A Transaction Ordering Dependency anaLyzER - for Ethereum Smart Contracts. In *2023 IEEE/ACM 6th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 9–16. https://doi.org/10.1109/WETSEB59161.2023.00007

[51] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. SFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 778–788. https://doi.org/10.1145/3377811.3380334

[52] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective context-sensitivity guided by impact pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 475–484. https://doi.org/10.1145/2594291.2594318

[53] Jihyeok Park, Seungmin An, and Sukyoung Ryu. 2022. Automatically deriving JavaScript static analyzers from specifications using Meta-level static analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1022–1034. https://doi.org/10.1145/3540250.3549097

[54] Daniel Perez and Benjamin Livshits. 2021. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1325–1341. https://www.usenix.org/conference/usenixsecurity21/presentation/perez

[55] 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).

[56] Mooly Sagiv. 2020. Invited Talk: Harnessing SMT Solvers for Verifying Low Level Programs.. In *SMT*. 2.

[57] 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. https://doi.org/10.1145/3243734.3243793

[58] Micha Sharir and Amir Pnueli. 1981. *Two Approaches to Interprocedural Data Flow Analysis.* Chapter 7, 189–233.

[59] Olin Grigsby Shivers. 1991. *Control-flow analysis of higher-order languages of taming lambda.* Ph. D. Dissertation. USA. UMI Order No. GAX91-26964.

[60] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 322–333. https://doi.org/10.1145/3597926.3598124

[61] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. *SIGPLAN Not.* 46, 1 (Jan. 2011), 17–30. https://doi.org/10.1145/1925844.1926390

[62] 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

[63] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 485–495. https://doi.org/10.1145/2594291.2594320

[64] Solidity. [n. d.]. Contract ABI Specification. https://docs.soliditylang.org/en/v0.8.25/abi-spec.html

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

[66] Solidity Team. 2022. Solidity 0.8.13 Release Announcement. https://soliditylang.org/blog/2022/03/16/solidity-0.8.13-release-announcement/

[67] Solidity Team. 2024. A Closer Look at Via-IR. https://soliditylang.org/blog/2024/07/12/a-closer-look-at-via-ir/

[68] Matt Suiche. 2017. Porosity: A Decompiler for Blockchain-Based Smart Contracts Bytecode. http://web.archive.org/web/20170915103422/https://www.comae.io/reports/dc25-msuiche-Porosity-Decompiling-Ethereum-Smart-Contracts-wp.pdf

[69] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 147 (oct 2021), 27 pages. https://doi.org/10.1145/3485524

[70] Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting. In *Static Analysis*, Xavier Rival (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 489–510.

[71] Michael Van Emmerik. 2007. *Static Single Assignment for Decompilation.* Ph. D. Dissertation.

[72] Valentin Wüstholz and Maria Christakis. 2020. Harvey: a greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1398–1409. https://doi.org/10.1145/3368089.3417064

[73] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. 2016. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. In *2016 IEEE Symposium on Security and Privacy (SP)*. 158–177. https://doi.org/10.1109/SP.2016.18

[74] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. https://doi.org/10.14722/ndss.2015.23185

[75] 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. https://doi.org/10.1145/3597503.3639153

[76] Mengya Zhang, Preksha Shukla, Wuqi Zhang, Zhuo Zhang, Pranav Agrawal, Zhiqiang Lin, Xiangyu Zhang, and Xiaokuan Zhang. 2025. An Empirical Study of Proxy Smart Contracts at the Ethereum Ecosystem Scale . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 620–620. https://doi.org/10.1109/ICSE55347.2025.00083

[77] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying Exploitable Bugs in Smart Contracts. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 615–627. https://doi.org/10.1109/ICSE48619.2023.00061

[78] 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. https://doi.org/10.1145/3611643.3616343

[79] Zhijie Zhong, Zibin Zheng, Hong-Ning Dai, Qing Xue, Junjia Chen, and Yuhong Nan. 2024. PrettySmart: Detecting Permission Re-delegation Vulnerability for Token Behaviors in Smart 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 168, 12 pages. https://doi.org/10.1145/3597503.3639140

Received 2024-10-31; accepted 2025-03-31