Elipmoc: Advanced Decompilation of Ethereum Smart Contracts

NEVILLE GRECH, University of Malta, Malta and Dedaub Ltd
SIFIS LAGOUVARDOS, University of Athens, Greece and Dedaub Ltd
ILIAS TSATIRIS, University of Athens, Greece and Dedaub Ltd
YANNIS SMARAGDAKIS, University of Athens, Greece and Dedaub Ltd

Smart contracts on the Ethereum blockchain greatly benefit from cutting-edge analysis techniques and pose significant challenges. A primary challenge is the extremely low-level representation of deployed contracts. We present Elipmoc, a decompiler for the next generation of smart contract analyses. Elipmoc is an evolution of Gigahorse, the top research decompiler, dramatically improving over it and over other state-of-the-art tools, by employing several high-precision techniques and making them scalable. Among these techniques are a new kind of context sensitivity (termed "transactional sensitivity") that provides a more effective static abstraction of distinct dynamic executions; a path-sensitive (yet scalable, through path merging) algorithm for inference of function arguments and returns; and a fully context sensitive private function reconstruction process. As a result, smart contract security analyses and reverse-engineering tools built on top of Elipmoc achieve high scalability, precision and completeness.

Elipmoc improves over all notable past decompilers, including its predecessor, Gigahorse, and the state-of-the-art industrial tool, Panoramix, integrated into the primary Ethereum blockchain explorer, Etherscan. Elipmoc produces decompiled contracts with fully resolved operands at a rate of 99.5% (compared to 62.8% for Gigahorse), and achieves much higher completeness in code decompilation than Panoramix—e.g., up to 67% more coverage of external call statements—while being over 5x faster. Elipmoc has been the enabler for recent (independent) discoveries of several exploitable vulnerabilities on popular protocols, over funds in the many millions of dollars.

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, Security, Ethereum, Blockchain

ACM Reference Format:
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 (April 2022), 27 pages. https://doi.org/10.1145/3527321

1 INTRODUCTION

Decentralized financial applications, built using smart contracts running on programmable blockchains (most notably Ethereum), are starting to rival traditional financial systems. Therefore coding or logical errors in smart contracts can have large financial implications. This makes smart contracts primary targets for automated analysis and verification tasks.

Authors' addresses: Neville Grech, University of Malta, Malta and Dedaub Ltd, me@nevillegrech.com; Sifis Lagouvardos, University of Athens, Greece and Dedaub Ltd, sifis.lag@di.uoa.gr; Ilias Tsatiris, University of Athens, Greece and Dedaub Ltd, i.tsatiris@di.uoa.gr; Yannis Smaragdakis, University of Athens, Greece and Dedaub Ltd, smaragd@di.uoa.gr.

2475-1421/2022/4-ART77
https://doi.org/10.1145/3527321

More specifically, the analysis of smart contracts *as-deployed*, i.e., by taking their binary form as input, is attractive for several reasons. First, it offers significant generality: many deployed smart contracts have no publicly released source code. Security analysts routinely need to inspect such contracts (e.g., bots) to determine their functionality. Second, analyzing contracts at the bytecode level offers a uniform platform for analysis, not distinguishing between source languages and source language versions. This is a major factor, given that (the dominant Ethereum language) Solidity does not maintain source compatibility across versions. Source-level tools very quickly become obsolete. Bytecode decompilation offers by far the most general substrate for automated smart contract analyses.

Decompilation of the Ethereum Virtual Machine (EVM) bytecode is an open problem for the security community. Existing approaches lack in completeness, precision, or scalability. For instance, the most-used decompiler in practice, *etherscan.io*'s Panoramix [Kolinko and Palkeo 2020], decompiles complex contracts only partially, with much of the low-level code often not reflected in the decompiled version. Past academic research tools, such as the Gigahorse decompiler [Grech et al. 2019a], sacrifice precision and occasionally even scalabily. Lower precision results in low-quality decompilation, for instance, operations with unknown operands, or jumps with many infeasible targets. To illustrate such shortcomings, our experiments show that these leading past decompilers manage to decompile under 20% (Panoramix: 18.2%, Gigahorse: 19.8%) of Ethereum contracts of large size (over 20KB).

Technically, the low-level nature of the EVM poses a challenge for any decompilation effort. The EVM uses a single stack for calls and local operations, without any guarantees about its well-formedness. The only control flow operators are unstructured jumps. All calls, returns, loops, and conditionals are implemented as jumps to whichever address is currently at the top of the stack. Compilers producing EVM bytecode often perform aggressive optimization, since instructions have a monetary cost to execute. A major obfuscating factor, for instance, is the aggressive merging of identical instruction sequences belonging to different internal, a.k.a. *private*, functions. Control and data flow are, as a result, very hard to discern without sophisticated modeling techniques.

In this paper, we introduce Elipmoc ("compile", backwards), a decompiler for Ethereum smart contracts. Elipmoc advances the state of the art in EVM bytecode decompilation, offering significantly increased precision and completeness. Elipmoc is the substrate of a successful analysis framework that has flagged numerous exploitable vulnerabilities on contracts with or without source (but with millions of dollars in locked value). All analyses operate over the three-address code exported by Elipmoc. We have made several vulnerability disclosures, some of which resulted in major rescue efforts [Dedaub 2021b,c,d,f; Michales, Jonah 2021; Primitive Finance 2021]. In addition, the Elipmoc team has been commissioned to perform three separate studies for the Ethereum Foundation, for the assessment of the impact of Ethereum Improvement Proposals EIP-1884, EIP-3074, and a future EIP that proposes a rearchitecting of storage gas cost metering. The Elipmoc decompiler allows answering questions such as "what will be the impact of *change-X* on the entire set of currently deployed contracts?" and is becoming a crucial tool for the evolution of the Ethereum blockchain and many of its applications.

In technical terms, Elipmoc makes the following contributions:

- **Transactional Context Sensitivity.** Elipmoc proposes an effective form of context sensitivity for smart contract execution. Context sensitivity offers a way to condense actual executions into short static signatures. Devising good signatures is crucial for both precision and scalability throughout the decompiler. Elipmoc's transactional context sensitivity is based on insights about EVM low-level control flow (esp. continuation-based optimizations) and overall execution.

- **Scalable, context & path-sensitive function reconstruction.** Since compilers tend to reuse EVM bytecode sequences for the benefit of multiple functions, a direct inference of private functions is not straightforward. Additionally, high-level control flow is compiled away to the point of near-irreversibility, using continuation-passing-style patterns. We address such issues with a static analysis maintaining the current contents of a continuation stack, and a path-sensitive analysis to determine with high precision the maximum number of stack elements consumed, i.e., the likely number of function arguments. To make such algorithms scalable, we employ various techniques to reduce complexity, such as merging paths into unordered sets of basic blocks.
- **Experimental benefit over state-of-the-art.** We thoroughly compare Elipmoc to its predecessor, the leading EVM research decompiler, Gigahorse [Grech et al. 2019a] and the best-known, most-used industrial decompiler, Panoramix [Kolinko and Palkeo 2020]. Elipmoc exhibits much higher precision than Gigahorse—e.g., dropping the percentage of contracts with unresolved operands for some operation from 37.4% to 0.5%. It is also much more complete than Panoramix—e.g., mapping 40% more `CALL` instructions in the input into decompiled statements—as well as significantly more scalable—with a 5x speedup and timeouts for under 5% of contracts, to Panoramix's 17.8%.

Generally, Elipmoc showcases how core programming language implementation techniques contribute to a cutting-edge domain, outperforming the leading past research and industrial tools.

Brief Illustration. It is useful to start by defining the purpose of *decompilation* (a.k.a. *binary lifting*) in our intended use of the term. In our setting, ***decompilation is defined as the recovery of a structured intermediate representation (three-address code) from very low-level bytecode***.

This representation enables advanced automatic processing of the contract code, by downstream tools. An easy-to-appreciate tool (but not the primary goal of our decompilation) is a *source unparser*, producing a human-readable version of the code. Figure 1 showcases improvements of Elipmoc's source-like output over that of Gigahorse—the leading past research decompiler.

The decompiled code is for the `transfer` function of an ERC20 token contract. Differences include the more complete inference of high-level calls (Gigahorse fails to recognize the private, compiled-away `_SafeAdd`) and the much less noisy function argument inference (Gigahorse infers a function call to a non-existing 6 argument function). These techniques are described in detail in Sections 5.2 and 5.3. The structure (both control- and data-flow) of the Elipmoc output is much closer to that of a human-written source program. One could claim that the intent of the code is understandable in the Elipmoc version, yet obscured due to noise (decompilation imprecision errors) or omissions (decompilation incompleteness) in the Gigahorse version. Arguably, an important factor is the introduction of human-readable names (e.g., "`_SafeAdd`"), which is orthogonal to the decompilation itself, yet this orthogonal inference is not even possible if the decompilation does not correctly recover the limits and arguments of functions.

2 BACKGROUND

We proceed to describe the context within which Elipmoc exists—analysis of Ethereum smart contracts.

2.1 Smart Contracts

Ethereum is the largest programmable blockchain, on virtually any metric. The programming platform at the core of Ethereum, the EVM, is also used in other smart contract platforms, such as Binance Smart Chain (BSC), Polygon (previously Matic), JP Morgan Chase's Quorum [Chase 2020], Tron, Solana, Fantom and Cardano (via KEVM). The EVM is therefore the defacto standard smart contracts platform, akin to the JVM for enterprise applications. The semantics of the EVM are

```
function transfer(address varg0, uint256 varg1) public {
  require(!msg.value);
  require(!(0xff & _addEther >> 160));
  require(varg0 != 0);
  require(varg1 <= _balanceOf[msg.sender]);
  if (varg0 != address(_rubusOrangeAddress)) {
    v0 = _SafeSub(varg1, _balanceOf[msg.sender]);
    _balanceOf[msg.sender] = v0;
    v1 = _SafeAdd(varg1, _balanceOf[varg0]);
    _balanceOf[varg0] = v1;
    emit Transfer(msg.sender, varg0, varg1);
    emit MoreData(0, _ethPerToken);
    v2 = 1;
  } else { ... } ... }
```

```
function transfer(address varg0, uint256 varg1) public {
 require(!msg.value);
 require(!(0xff & stor___fun_selector__ >> 160));
 require(varg0 != 0x0);
 v232a = msg.sender;
 v232f = 0x20;
 require((varg1 <= owner_2[v232a][0]));
 if (varg0 != address(storage_3 >> 0))) {
  v265a = msg.sender;
  v265f = 0x20;
  v2676_0 = 0xdcf(varg1, owner_2[v265a][0]);
  v26a8 = msg.sender;
  v26ad = 0x20;
  owner_2[v26a8] = v2676_0;
  v26ef = varg0;
  v26f4 = 0x20;
  0xdb1(varg1, owner_2[v26ef][0], 0x270c, 0x0, 0x0, varg1);
 } else { ... } ... }
```

Fig. 1. Elipmoc (top) vs. Gigahorse (bottom) output. Slight simplification (names, casts) for space.

detailed in the Ethereum yellowpaper [Wood 2014] and the rest of the paper assumes knowledge of this execution model, including knowledge of the EVM stack, control flow instructions, etc.

Smart contracts are predominantly written using the high-level language Solidity [Various 2018]. Solidity is used in over 99% of the deployed contracts that publish source. (The second most-used language is Vyper [Various 2017], which statistically lags far behind in adoption yet has some prominent uses.) Contracts are compiled to a low-level bytecode representation that is executed on the EVM. This bytecode is deployed to the blockchain where it is persisted at an address. It is then publicly accessible for anyone to interact with. A deployed contract instance is identified with its address, which also holds persistent storage (for maintaining the contract's internal state) and cryptocurrencies. The native cryptocurrency of Ethereum, Ether, is used as a store of value, as a means of payment for executing smart contracts on the blockchain, and as a reward for the miners that maintain the security of the Ethereum network.

The execution of a smart contract is transactional in nature. The execution starts at the first instruction until an instruction is encountered that halts or reverts the execution. Ethereum smart contracts follow a standard application binary interface (ABI), where each external call to a contract needs to select an entry. Therefore each transaction has a single entry point into the contract: a public function.[1] The selection of this entry point is explicitly relayed by the caller by prepending the function signature to the payload message. This payload message is parsed by the smart contract and the appropriate code is executed. This public entry point will become an important component of our approach to context sensitivity in Section 4.

[1]Recursive calls are possible, but each would live within its own "internal transaction", thus in a separate EVM instance.

The mechanisms employed for transfering control between functions is what sets it apart from other low level bytecode languages. In EVM bytecode, all control-flow transfer mechanisms are dynamic. This is not the case of low-level languages such as JVM bytecode. This dynamicity is further challenged by the fact that the EVM does not enforce statically balanced stacks so the depth of the stack at any point in the program can be different according to which path the program has taken. It is noteworthy that even x86 assembly may be easier to analyze in this context—unlike x86, the EVM has no method invocation and return instructions. Instead, private function calls within a contract are translated to dynamic jumps. Hence, all functions in a contract are fused into one stream of instructions, with dynamic jumps as the only means to transfer control.

The challenge of EVM bytecode analysis lies in the low-level nature and unconventionality of its design. We shall see several aspects of the EVM in Sections 4 and 5. Here we only note that EVM bytecode is, in many ways, a lower-level language than even typical microprocessor (e.g., x86) assembly—it has no calls, only dynamic jumps, and a single mixed stack for instruction addresses, function arguments, return values, and local operands. The proliferation of high-value smart contracts compiled for the Ethereum Virtual Machine (EVM) virtually guarantees that this computing platform will remain relevant.

2.2 Smart Contract Analysis

Analysis and verification of smart contracts is now commonplace and a standard part of the software engineering process. The importance of this process is usually appreciated by all the system's stakeholders as software defects in smart contracts can cripple entire crypto-economic systems. Smart contract security audits are generally conducted by external organizations, with access to smart contract analysis and verification tools.

Analysis or verification of smart contracts can be conducted either at the source code or at the bytecode level. Analysis at the bytecode level requires a lifter, like Elipmoc. Most major Ethereum smart contract protocols—e.g., *Decentranized Finance (DeFi)* applications—publish sources. The majority of deployed contracts, however, do not: the deployers are not inviting interaction with parties outside their trusted user base. In this highly security- and privacy-conscious space, being able to inspect and automatically analyze all contracts is invaluable.

Even when source code is available, automated analysis at the bytecode level is beneficial, or even essential. Most contract sources also incorporate inline assembly for critical functionality. At submission time, 78% of substantial contracts (over 1000 LoC) published on Etherescan.io contain inline assembly. Assembly use is rife in popular libraries—e.g., [Various 2018a]. Typical routines that use assembly are used to interact with external contracts in a dynamic manner or for cryptographic functions. Decompilation is therefore essential for recovering the high-level semantics of modern smart contracts, uniformly integrating both their high-level and assembly-level parts.

More importantly, bytecode-level analysis guarantees that a tool stays relevant across different source languages and source-language versions. Source-level tools in the Ethereum space are highly fragile and language-version-dependent, since the Solidity language changes very frequently in syntactically incompatible ways. Many automated analysis tools that work at the source level (i.e., eschew decompilation from bytecode) quickly become irrelevant in practice because of the rapid advancement of (incompatible) Solidity versions. For instance, a recent study [Brent et al. 2020] reports that the Securify2 tool, which operates at the source code level applies to under 3% of deployed contracts. Additionally, although source analysis tools are generally easier to engineer and use, bytecode analysis tools are more faithful to the run-time semantics.

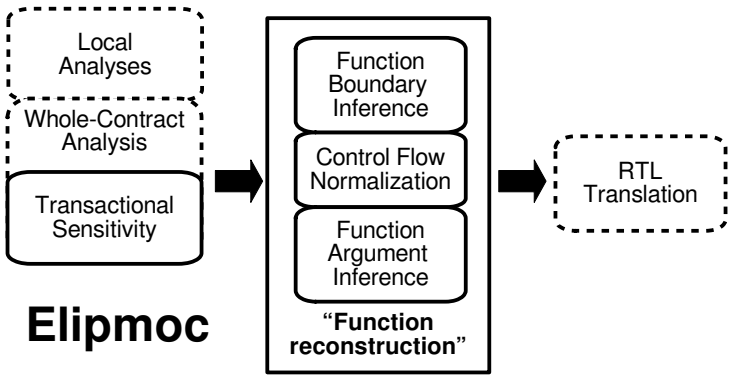

Fig. 2. High-level architecture of Elipmoc, noting the new elements (solid lines). Dashed elements are reused from the Gigahorse framework [Grech et al. 2019a] that Elipmoc builds on.

3 STRUCTURE OF ELIPMOC

A bird's-eye view of Elipmoc is shown in Figure 2. This figure depicts the various technical contributions in solid-line boxes, together with how these fit within existing technologies.

After the bytecode is parsed, a straightforward process, local analyses on this bytecode summarize the effects of each basic block. These local analyses also compute direct edges between basic blocks.

Similar to past decompilers, the transformed code is now analyzed with the whole contract as a single monolithic function, with dynamic jumps to transfer control flow between internal function callers and callees, and vice-versa. Unfortunately, an analysis of this kind can quickly lead to imprecision, as the same snippet of code can be used in many different parts of the program under different contexts. Elipmoc addresses this issue using an appropriate novel form of context sensitivity called *transactional context sensitivity*, which is detailed in Section 4. Transactional context sensitivity can retain precision, while maintaining scalability. The global analysis yields a global control-flow graph, with all data flows between stack locations for each point of the program resolved. This information takes a highly-precise form—e.g., "the value at location l of the stack upon entry to block B may have been pushed at instruction i". The precision is essential for recovering the high-level form of the code.

The information gleaned from the whole-contract analysis is used to produce a new intermediate representation, tailored to subsequent analysis phases. Subsequently, the *private function reconstruction* starts. This starts by identifying the function boundaries (Section 5.2). The algorithms here are concerned with assigning each basic block to a single function. Normally, this also requires *control-flow normalization*. Briefly, a new normalized intermediate representation is produced. In this IR, basic blocks that are reused between different functions are inlined (i.e., also cloned as necessary), reversing global code-reuse optimizations performed by the high-level language compiler. This greatly facilitates the final step: *function argument inference*. In this step, function arguments are inferred using another precise (and potentially very costly) path-sensitive algorithm. The algorithm is optimized to retain scalability, merging paths into sets of basic blocks, instead of sequences of (artificially bounded) length. This algorithm is presented in Section 5.3.

Finally, a register-transfer-language representation can be emitted, which is used by subsequent security analyses or applications, including the higher-level decompilation.

After function reconstruction, the output of the decompiler is suitable for performing client analyses. However, the same cannot be said about its human readability. The reverse-engineering

client in the Elipmoc toolchain addresses this issue, outputting source-like code, intended for human consumption. This performs various high-level feature reconstruction analyses such as detecting structured programming constructs. Internally, the reverse-engineering client constructs an AST, by inlining expressions whenever possible, up to a depth threshold. A series of AST transformations are then performed that aim to optimize human readability, before finally serializing the resulting AST into text form. Source-level idioms (e.g., `require` and `assert` statements for dynamic checks, `emit` statements for events) are introduced. Human-readable signatures for functions and events are shown when available.

4 TRANSACTIONAL CONTEXT SENSITIVITY

Elipmoc is built on the declarative (Datalog) framework introduced by the Gigahorse decompiler [Grech et al. 2019a], inheriting the vocabulary and input/output form and mechanisms, but replacing nearly all of the complex decompilation core. Gigahorse uses as its backbone a *context-sensitive analysis* of possible stack contents per program point. Due to the design of later stages in the decompilation process, this stage in Elipmoc can be an order of magnitude more efficient, by only modelling stack locations that may contain possible jump targets. This added efficiency encourages the development of more detailed context sensitivity. The primary relation modeling the global stack is of the form `StackContents(ctx, insn, index, var)`, denoting that, under abstract execution context `ctx`, at instruction `insn`, stack position `index` holds the value originally assigned to variable `var`. (The intermediate representation is in static single assignment (SSA) form, so the variable has a unique assignment.)

Modeling the contents of the stack context-sensitively, in turn, yields a context-sensitive low-level control-flow graph, since all control flow is just jumps to the address held in the topmost element of the stack. In general, all inferences of the decompiler, even at a much higher level, depend crucially on the precision of the context-sensitivity model. For instance, in Figure 6 (Section 5) we will see the function reconstruction algorithm appeal to the context-sensitivity model to test if a path between a likely call site and a call continuation is realizable.

It is, therefore, crucial, for both precision and scalability, to define an appropriate context-sensitivity model. Maintaining full precision (e.g., all actual dynamic context of an execution—a super-exponential state space) is prohibitive for static analysis. Context-sensitive static analysis aims to define a compact abstract context that summarizes dynamic execution conditions in a way that executions with the same information (e.g., stack contents) have the same context (to avoid replication of effort), yet executions with interesting differences are not confused.

The context policy of Elipmoc, which we dub *transactional context sensitivity*, is a major element of its increased precision. Transactional context sensitivity is designed for the execution model of smart contracts and differs significantly from context sensitivities designed for functional [Shivers 1991], or imperative and object-oriented programs [Milanova et al. 2005]. Transactional context sensitivity is unique in that it retains part of the transaction payload (the public function signature) while analyzing the smart contract, and also adjusts local context heuristically. Before discussing transactional context sensitivity in detail (Section 4.2) we describe a useful pre-analysis.

4.1 Classifying Jump Instructions

The low-level execution flow of smart contracts is characterized by a series of low-level jump instructions. All these jumps are dynamic, i.e., to targets not directly apparent in the (bytecode) program text—the jump target is merely the address held at the top of the stack. Precision is essential for recovering valid code sequences: a basic block may be preceded by many and followed by many others, but very few combinations of predecessors and successors are realizable.

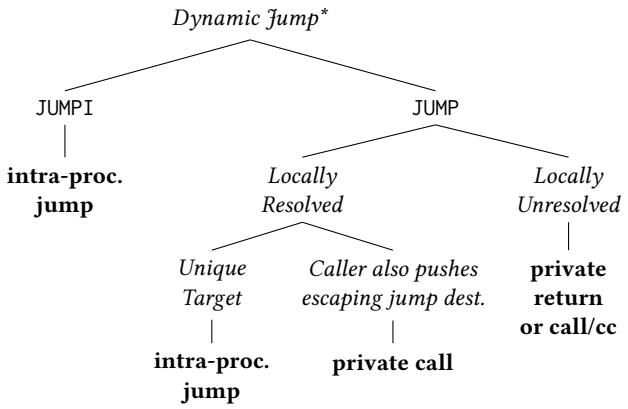

Fig. 3. How jumps in a smart contract are classified in terms of the kinds of control flow changes these are normally associated with.

The input form can be viewed as a continuation-passing-style (CPS) representation: function returns and calls are both treated as mere jumps, i.e., can be thought of as forward calls. Providing a return address at a function call site (i.e., at a jump instruction that implements a high-level call) is no different from providing an extra argument: the continuation, to be jumped-to at the end of the called code. However, in addition to calls and returns, the same jumps also implement all *local, intra-function* control flow, such as `if` statements or loops.

Past approaches [Brent et al. 2018; Grech et al. 2019a] have utilized various kinds of context sensitivity to recover precise control flow. Traditional call-site sensitivity[2] over CPS input [Shivers 1991] can be precise for very deep contexts: paths of length over 10 jump sites (or basic blocks). Unfortunately this does not scale well. Many of the dynamic jumps (around 90%), however, can be statically resolved by a fast local pre-analysis, which can yield some structure to the program at an early stage. Such an analysis is very scalable as it is local, however it also needs to perform constant folding.[3]

The output of this analysis, in conjunction with other transformations to standardize the control flow patterns (e.g. translating a conditional public function jump into simpler control flow operations), can be used to classify jumps as depicted in Figure 3.

The `JUMP` instruction can be used for many applications. The fact that a `JUMP` instruction's address can be locally resolved (i.e., the address was pushed to the stack in the same basic block as the instruction), however implies that the control-flow transfer was known to the caller. Therefore, we can immediately conclude that the instruction is either an intra-procedural jump or a direct call to another function. On the other hand, if the target of `JUMP` instructions is unresolved, it is likely a return from a private function, or a call to a continuation passed by the caller. These targets would have been passed on the stack multiple basic blocks prior in the execution.

[2]In this case, call-site sensitivity might be better named "jump-site sensitivity", but we choose to keep the conventional name.

[3]Note that jump targets may even be calculated at run-time (i.e., can be computed results of arithmetic and not merely different constants pushed along different execution paths). This aspect is mostly a function of the compiler used and its level of optimization. Note also that constant folding needs to implement the EVM's 256-bit arithmetic operations.

$call \in I$: set of instructions (jump sites)
PC: set of private contexts, $PC \cong I^n$
$ctx \in C$: set of contexts, $C \cong I \times PC$

Initially, ctx = [Null, [Null, Null]]

$$\textbf{Merge}(ctx, call) = \begin{cases} [pub, \textsc{Second}(ctx)], \\ \quad \text{if } \textsc{PublicCall}(call, pub) \\ [pub, [call, \textsc{First}_{n-1}(\textsc{Second}(ctx))]], \\ \quad \text{if } \textsc{PrivateCall}(call) \\ \quad \text{or } \textsc{PrivateReturn}(call) \\ ctx, \text{otherwise} \end{cases}$$

Fig. 4. Context constructor for transactional context sensitivity. Returns the context for the callee function given the current context, *ctx*, of the caller and the specific *call* instruction in the caller. For ease of exposition of the spirit of the definition, we use nested tuples 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.

Elipmoc can distinguish between simple intraprocedural jumps and direct function calls in an overapproximate, manner by finding (potentially folded) constants that are pushed onto the stack that are (i) valid jump targets that, (ii) *escape* (in dataflow terms) from that basic block. When these two conditions hold, that basic block is most likely calling a private function. The intuition behind this algorithm is that when compiling function calls, return label addresses need to be pushed prior to the call of a private function.

4.2 Context Sensitivity

A defining feature of transactional context sensitivity is that it retains the public function (i.e., the transaction's entry point) throughout the analysis. This means that code reachable from the same public end-points (exported by the contract) will be analyzed separately per such end-point. To form the full analysis context, this "sticky" context information is combined with a transient context part, which also benefits from the classification of jump instructions of the previous section.

Per past work [Jeong et al. 2017; Smaragdakis et al. 2011; Thiessen and Lhoták 2017], context sensitivity can be defined precisely by a constructor **Merge**, which produces a new context for a callee taking as input the information (context and call instruction) at the call site. The definition of **Merge** for transactional context sensitivity is shown in Figure 4. The context is made up of two components: the public function context, and the private function context. The public function context keeps the function signature of the currently executing public function, which is initially Null. The private function context keeps context information about the currently executing private functions. This part of the context retains the n most recent *likely* private function call sites or return sites found while jump instructions are being processed. A jump instruction that does not correspond to a call or return does not change the context. Figure 5 illustrates: Two execution paths are shown and their static abstraction is highlighted. The bold arrows show the essence of transactional context sensitivity: the execution context modeled statically only retains the source block of the very first edge of an execution path (i.e., the public function entry point) and the source block of the most recent "important" edges. All other, intermediate execution points are not modeled statically, i.e., paths that only differ in those will be staticaly treated as one.

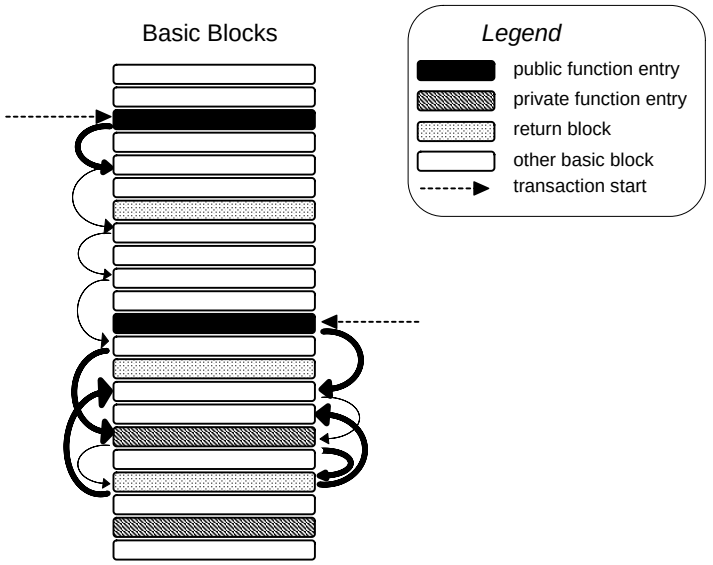

Fig. 5. Two sample execution paths (left and right of basic blocks) and their static abstractions (consisting of the blocks from which bold arrows originate), under transactional context sensitivity of depth 2 for the private context. The two examples illustrate transactional context sensitivity: execution paths are distinguished by keeping the source of the very first edge (the public entry point) and that of the most recent few edges (2 in this case) that are likely to be private function calls or returns.

Keeping the top-level (i.e., public function) transaction entry point is one part of the novelty of transactional context sensitivity. The other part concerns the private context. By definition, call-site sensitivity over CPS input [Shivers 1991] treats every call site (in our case, jump instruction) as a point of context update. Transactional context sensitivity updates the context only at a subset of these jumps: those earlier classified as *likely* private function calls or returns.

Retaining a return-site context distinguishes return targets (i.e., continuations) depending on which previous block reaches them. This decision is beneficial for precision given the *tail-call* and other continuation-passing optimizations (see Section 5) employed by the high-level language compilers: "return" sites may actually not return to the caller but to the caller's caller or the next function being called.

These subtle insights contribute to the unique performance characteristics of transactional context sensitivity. In general, the combination of jump instruction classification and the transactional context sensitivity policy provides an excellent heuristic for high-value context elements to maintain for both scalability and precision.

5 (PRIVATE) FUNCTION RECONSTRUCTION

Function reconstruction is a major part of Elipmoc. This is a hard problem, only solvable in empirical terms. However, better function reconstruction directly affects the bottom line of a decompiler: the quality/precision of decompilation, along multiple metrics, improves significantly.

5.1 The Private Function Reconstruction Problem

To detect (private) functions, Elipmoc uses computationally-demanding algorithms that require the global dataflow information, together with the global CFG, constructed during the whole-contract analysis phase. Although all (or most) stack locations have been resolved to global definitions throughout the smart contract, the resulting program does not contain functions: these have been compiled away in EVM bytecode.

The *function reconstruction problem* concerns identifying *private* high-level functions (likely present in the original source program) from a compiled smart contract in EVM bytecode.[4]

For a systematic definition, function reconstruction aims to identify private functions consisting of groups of basic blocks, used in a function-like manner:

- the function is reached via jumps from multiple call sites;
- the data- and control-flow patterns at these jumps match high-level calls:
  - (a consistent, across call sites, number of) arguments are passed on the stack before the jump that purportedly implements the function call;
  - a return address is similarly passed to the function;
  - the called function returns (a consistent number of) return values—the EVM allows functions with multiple return values;
  - the called function returns by jumping to the return address pushed by the caller.

The above properties seem easily checkable via static analysis. However, the compilation process introduces many exceptions to high-level function patterns, mostly because of global optimizations performed during compilation to EVM bytecode. Even worse, most of the properties that one typically associates with high-level functions are violated.

- Basic blocks are aggressively merged, so that a coincidentally identical basic block is shared across several different functions. With the EVM being a stack machine (without named temporaries, only stack locations) identical basic blocks (i.e., sequences of 3-8 instructions) are very common even accidentally. Furthermore, many of these common blocks implement standard snippets of code (e.g., bounds checking routines, memcopy routines, etc.).
- Functions do not always return to their high-level callers: the supplied return address could well be the continuation of the call, which could be the caller's caller (i.e., the call is a tail call) or the caller's next callee (i.e., the call is chained with another).
- The return block could be pushed on the stack long before the actual call. (E.g., before an intervening call that computes one of the second call's arguments.) The compiler only has an obligation to keep the stack values in the right order, and is motivated to avoid superfluous instructions to reorder values on the stack.
- Functions do not always return: they can also abort or successfully exit the transaction, in which case the stack can be left in any state. Arguments on the stack could be left unconsumed and return values may be partially pushed.
- Basic blocks may be calling a function (or returning from a function) under one path of execution, and do something completely different under another.

The above points constitute significant challenges for tools attempting to recover private functions. Failing to recover function calls and their boundaries can result in having the "merged" basic blocks used in many different ways inside the same caller-function. This can be observed in Figure 1, where failing to recognize calls results in unstructed control-flow, greatly affecting the precision

[4]In contrast to private functions, *public* functions are relatively easy to infer: they have well-known entry points and often published signatures. In fact, most existing decompilers [eth 2018; Brent et al. 2018; Grech et al. 2019a; Kolinko and Palkeo 2020] can identify public functions. To our knowledge the Gigahorse decompiler [Grech et al. 2019a] is the only other to try to infer *private* functions.

of any tools running on top of the decompiler. In addition, incorrectly inferring function boundaries or a function's number of arguments can affect the completeness of the decompilation output in two different ways. If the number of arguments of a private function is inferred incorrectly, a statement can attempt to use a stack variable that would correspond to an argument (set outside the function) and be unable to resolve it. In addition, if a block is falsely inferred to be part of a private function it may attempt to use stack variables that are neither pushed inside the function nor are arguments, resulting, again, in an unresolved operand.

To face these complexities, Elipmoc uses two separate clusters of algorithms, one for detecting functions and one for inferring their arguments and returns. The next sections examine them.

5.2 Function Boundary Inference

The only function reconstruction algorithm for EVM bytecode in past literature [Grech et al. 2019a, Section IV.C] is rather simplistic: a function needs to be called from multiple call sites, and a call site needs to push a return address and arguments. This approach works for simple cases, such as the one below, which calls function `foo` with argument `0xFF`.

```
     PUSH ret    // set return address
     PUSH 0xFF   // set argument
     PUSH foo    // set function address
     JUMP        // jump to (call) 'foo'
ret: ...
foo: ...
     JUMP        // jump to 'ret' (return)
```

This function pattern, however, is inadequate for the complications of optimized EVM bytecode mentioned earlier: it produces functions that share basic blocks, it does not detect return addresses pushed long before the call site, and it gets confused with inconsistent use of function arguments or returns (e.g., due to transaction termination).

Elipmoc offers a four-step approach yielding both better precision and completeness.

(1) Find instructions that push addresses and then jump to code that will eventually jump back to the pushed addresses. This is the root inference pattern, identifying likely return instructions. Although the return is identified with high confidence, the call is not identified. Every jump, from the original block that sets the return address, up until the return jump, is a potential matching call that leads to the return.
(2) For each such potential call site, find all addresses on the stack at the time of the jump, ranked by stack depth. The call site at which the return address is at rank 1 (i.e., at the top of the stack after the call) is a possible matching call site, provided context-sensitive reachability conditions are satisfied. This information is maintained recursively across calls, handling the pattern of pushing return addresses well before a call (due to other intervening calls).
(3) The above produces an over-approximation of possible call- and return-site pairings. These are filtered for well-formedness. The main filter ensures that a jump classified as a return is to a block that is not also a target of a jump classified as a call. Functions are inferred with high confidence when no conflicting call-return classification exists and the entry block (i.e., target of a call) is called multiple times.
(4) In all cases of blocks reachable from multiple function entries, inline them (by cloning) in all functions that can reach them. This *control-flow normalization* phase addresses the basic block merging that occurred during compilation. The invariant guaranteed by inlining is that a function "owns" all the basic blocks that it is made of. In other words, any basic block in one function should not be reachable from the body of another function.

All of the above steps reveal important design choices. However, the algorithm of Step 2 is the most involved, and also has an important effect, worth illustrating.

To motivate Step 2, consider the following (simplified) pattern in compiled EVM bytecode.[5]

```
     PUSH ret3
     PUSH 0x3
     PUSH ret2
     PUSH 0x2
     PUSH ret1
     PUSH 0x1
     PUSH foo
     JUMP // jump to (call) 'foo', returns to ret1
ret1:
     PUSH bar
     JUMP // jump to (call) 'bar', returns to ret2
ret2:
     PUSH baz
     JUMP // jump to (call) 'baz', returns to ret3
ret3: ...
```

This pattern implements the chained high-level calls `baz(bar(foo(0x1), 0x2), 0x3)`. The complication is that the return addresses of `bar` and `baz` are pushed on the stack well before the jump instructions that implement the respective calls, separated by any number of instructions, both statically, in the program text, and dynamically, in the execution.

Figure 6 illustrates the heart of the algorithm of Step 2, in pseudo-rules that closely capture the actual structure of the computation. Relation `ReturnAddressRank` has initial contents computed locally (not shown in Figure 6), i.e., from addresses pushed on the stack in the same basic block as `setter`. (The stack serves both as an activation and as an operand stack, so it will also have arbitrary other contents: arguments, local temporaries, etc. Only return addresses are kept and ranked in `ReturnAddressRank`.)

The contents of `ReturnAddressRank` are then enriched, as shown in the figure, in mutual recursion with the eventual output of Step 2, relation `FnCallReturn`. The latter relation identifies matching call-return pairs, based on the top-most address on the stack (rank 1) that the caller pushes and the return jumps to. Every such call-return pair matched allows the further propagation (from before the call to after the return) of other return addresses on the stack.

In our EVM code example, once the call-return pattern for `foo` is recognized (as a `FnCallReturn`), the `JUMP` instruction for `bar` acquires (via the second rule) a `ReturnAddressRank` of rank 1 for `retTo` being `ret2`. This, in turn, allows recognizing the call to `bar` and its return to `ret2` as a `FnCallReturn`. Finally, the same pattern applies to resolve the call to `baz`.

Using such precise modeling of stack contents and gradually combining it with function reconstruction, Elipmoc manages to revert the vast majority of control-flow tangling resulting from optimizing decompilation. The techniques are likely applicable beyond EVM bytecode, in other CPS settings, although this is just an informal assessment—a full claim can only be ascertained experimentally, in future work.

Adding context sensitivity. In order to simplify presentation, the algorithms in this section have been described in a context-insensitive manner. It should be noted however that the Elipmoc implementation of function reconstruction is fully context-sensitive: the relations shown are

[5]Although the pattern is not uncommon, for a concrete example one can see the Rubus token contract, `0x876a11639ce3d2bba1712fc9f47bd6faee575ad4`, basic block `0x1cc`, using any contract code explorer (e.g., *etherscan.io*).

ReturnAddressRank(setter,retFrom,retTo,rank)
`setter` pushes the address `retTo` on the stack, and it is the n-th such address it pushes (n dubbed the"rank"). `retFrom` eventually jumps to the `retTo` address.
FnCallReturn(caller, retFrom, retTo)
`caller` is inferred to call a function with return block `retFrom`, returning to `retTo`.

```
// If the callee function returns to the very first address (rank 1) on the stack
// that the caller supplied, this is a likely return from the callee to the caller
FnCallReturn(caller, retFrom, retTo) :-
  ReturnAddressRank(caller, retFrom, retTo, 1),
  FeasiblePath([caller, retFrom, retTo]).

// We have found a possible call-return pattern, propagate to the return block
// the extra pushed return values of the earlier caller, with rank
// adjusted based on the return block's other pushed addresses.
ReturnAddressRank(caller, retFrom, retTo, n') :-
  FnCallReturn(setter, _, caller),
  ReturnAddressRank(setter, retFrom, retTo, n),
  offset = max( rank: ReturnAddressRank(caller, _, _, rank) ),
  FeasiblePath([setter, caller, retFrom, retTo]),
  n > 1, n' = n+offset-1.
```

Fig. 6. Recursive inference of potential call-return pairs (`FnCallReturn`) and rank of return address among others on the stack (`ReturnAddressRank`). The presentation uses syntactic shorthands for computing the maximum matching entry in a relation and for context-sensitive reachability through any number of intermediate nodes.

enhanced with context elements so that infeasible combinations (e.g., values of `retFrom` and `retTo` in `ReturnAddressRank` that will never arise in the same execution) are pruned away.

5.3 Function Argument Inference

The final step in the function reconstruction process is the inference of function arguments. Similar to other problems, notions of function arguments have been erased from the program. Instead, arguments are passed over the stack by the caller and the callee returns its values over the stack. The main intuition behind function argument inference is that the number of arguments of a function is the high-watermark number of elements that the function will pop from the stack. The number of return arguments is the difference in number of elements that will have been pushed on the stack at function exit, after popping the arguments. This computation is far from straightforward because the number of arguments that are popped from the stack may change depending on runtime conditions.

For instance, if the function `foo`, shown in Figure 7, follows the path of the error checking logic, it will not pop the actual arguments from the stack. This means that in order to correctly determine the number of arguments, all possible paths need to be followed. Unfortunately, the EVM does not enforce stack balancing across branches, which means that different stack heights can be achieved depending on the path taken to reach a basic block. This can have even worse consequences if cycles emerge in the CFG, either explicitly, or due to imprecision.

```
foo: va = SLOAD 0x5
     vc = LT 0x1 va
     JUMPI vc <error>
     ... // use two arguments
error:
     REVERT
```

Fig. 7. Simple error handing logic (shown in Elipmoc's three-address IR) asserting that the value held in storage location 0x5 is greater than 0x1.

PotentialCycleEntry(block)
Basic block "block" may be the entry of a cycle in a path.
FnPopAndΔ(func, to, path, maxPop, Δ) A path, from function entry block func to node to has in its course a maximum number of elements (high watermark) maxPop popped from the stack and leaves the stack at difference Δ relative to its level at function entry.

```
PotentialCycleEntry(block) :-
  LocalBlockEdge(prev, block),
  LocalBlockEdge(prev2, block),
  prev ≠ prev2.

FnPopAndΔ(entry, entry, entry, 0, 0) :-
  FnEntry(entry).

FnPopAndΔ(func, to, path', maxPop', Δ') :-
  FnPopAndΔ(func, from, path, maxPop, Δ),
  LocalBlockEdge(from, to), to ∉ path,
  BlockMaxPop(from, blockMaxPop),
  BlockΔ(from, blockΔ),
  maxPop' = max(blockMaxPop - Δ, maxPop),
  Δ' = blockΔ + Δ,
  (PotentialCycleEntry(to), path'= path ∪ {to}
    ∨
   ¬PotentialCycleEntry(to), path' = path).
```

Fig. 8. Path-Sensitive Algorithm computing the number of stack elements popped and shifted by a function. For clarity, logical operators (¬, ∨, ≠) are used instead of their Datalog syntax counterparts. Set operators (∪, {}, ∉) are syntactic shorthands for calls to our Datalog-extending path functors.

The algorithm is selectively path-sensitive[6] one and is defined in Figure 8. This core algorithm finds all the possible number of elements that can be popped by the function under any acyclic path taken within the function.

In order to scale this algorithm, Elipmoc uses a number of optimizations:

[6] "Path-sensitive" is an overloaded term in static analysis. We use it in a "meet-over-all-paths" sense, to mean that different program paths, of length not bound *a priori*, are examined separately before their results are merged. The algorithm does not take into account the logical (branch) conditions needed for following a certain path.

(1) A new Datalog type `Path` and associated functors are developed. This allows efficient abstraction of paths within a single Datalog relation. The functionality is implemented as a C++ extension (functor) for the Souffle Datalog engine.
(2) Paths are maintained as unordered sets, not as ordered sequences. Since the stack effects of a basic block are independent of how it is reached, this does not affect precision.
(3) The possible number of stack elements that can be popped or shifted are calculated in tandem, within a single relation. This makes the implementation more efficient as the `Path` needs only to be manifested once.
(4) The `Path` element is an overapproximation of the true path taken by the program. Only blocks that could have potentially acted as a "gateway" to a new cycle (`PotentialCycleEntry`) in the execution are tracked. In a function with a single entry, these would be blocks with more than one predecessor.

The path-sensitive algorithm significantly enhances the precision of function argument inference and is key for raising the level of abstraction in decompiled code.

Comparison with Gigahorse. At a high-level, the description of the argument inference algorithm of Elipmoc is similar to that in Gigahorse. For instance, per the Gigahorse paper [Grech et al. 2019a][IV.D]

> *Gigahorse infers the number of function arguments by computing the number of caller-supplied elements that the entire function pops from the stack throughout its execution.*

That is, both decompilers try to compute the maximum number of elements popped from the stack (high watermark) during a function's execution. However, the crucial difference is in the precision of the path-sensitive approach of Elipmoc. Gigahorse employs a *polynomial* (non-path-sensitive) algorithm: the relevant program predicate computes `FnPopAndΔ` (using Figure 8's terminology) summarized *up to an instruction*. This collapses together all the paths that may be used to reach that instruction, by considering the maximum among predecessors.

In contrast, as seen in Figure 8, Elipmoc maintains the paths separately, resulting in a worst-case exponential algorithm. The use of an exponential algorithm, in turn, requires support for this computation outside the Datalog language (which captures the PTIME complexity class), using a C++ functor. To keep the complexity bounded, Elipmoc utilizes the approximations described earlier: paths are sets, not sequences, and are cut off at key points.

6 END-TO-END IMPACT

Although this does not constitute a systematic evaluation, Elipmoc has had significant end-to-end impact, as the substrate of a large analysis and security inspection infrastructure. Research tools built on top of it include the Ethainter [Brent et al. 2020] analyzer for composite taint vulnerabilities, an EVM "memory" modeling analysis [Lagouvardos et al. 2020], and a symbolic value-flow ("symvalic") analysis [Smaragdakis et al. 2021] combining concrete values and symbolic expressions. This analysis machinery has recently yielded seven high-profile, critical vulnerabilities, some of which resulted in major rescue efforts [Dedaub 2021b,c,d,f; Immunefi 2021; Michales, Jonah 2021; Primitive Finance 2021]. The vulnerable services include two of the largest DeFi services that collectively hold over $1.5B in assets as of the time of writing. Three of the vulnerabilities [Dedaub 2021c,d; Immunefi 2021] are over contracts with no source code publicly available, therefore decompilation was essential even for human inspection, and not just to produce input for analysis tools.

Elipmoc has been the basis of three separate studies [Dedaub 2019, 2021a,e], commissioned by the Ethereum Foundation, over the entire set of deployed contracts. These studies assessed the impact of Ethereum Improvement Proposals EIP-1884, EIP-3074, and of a new architecture

of gas cost metering. Elipmoc allows the uniform treatment of deployed contracts regardless of their provenance (i.e., which language and which version was used to produce them) and the presence of source code. In this way, we could answer questions such as "what will be the impact of *change-X* on the entire set of currently deployed contracts?" Furthermore, the largest insurer of DeFi protocols has commissioned a watchdog service running on top of Elipmoc to constantly monitor the security of protocols it underwrites. Elipmoc is available as an open source repository [7] on GitHub. (Elipmoc is essentially Gigahorse 2.0 and the public repo retains the name "Gigahorse". However, the "Elipmoc" code name is one we use internally and is highly useful for purposes of comparison in this paper.)

7 EVALUATION

We next compare Elipmoc in detail to the best-performing past research/applied tool (Gigahorse—Section 7.1) and the most-used state-of-the-art industrial decompiler (Panoramix—Section 7.2).[8] Further experiments that validate the design decisions of Elipmoc are presented in Section 7.3. Our experimental evaluation aims to answer the following research questions.

RQ1: Precision. Does the decompiled code exhibit high precision, enabling a good match with high-level semantics (e.g., structured control flow, split into functions, operators with the expected inferred operands)?

RQ2: Completeness. How well does the decompiled code cover the bytecode's full behavior?

RQ3: Scalability. Is Elipmoc and its design scalable, in terms of successfully decompiling realistic smart contracts?

To speed up experiments, we run all tools using 40 concurrent processes on an idle Ubuntu 20.04 machine with two Intel Xeon Gold 6136 CPUs @ 3.00GHz (each with 12 cores w/SMT, for a total of 48 hardware threads) and 128GB of RAM. We use a cutoff of 120 seconds for each contract. We use Souffle 2.0.2 and Python 3.8.2 where needed. The depth of the private context for transactional context sensitivity is fixed at 8.

The dataset used for the experiments is a uniform random sample of 5000 unique contracts, first deployed on the main Ethereum network between blocks 12300000 (April 24, 2021) and 13300000 (September 26, 2021). This dataset is obtained by syncing the official Go Ethereum client, dumping its database of contracts and removing duplicates. This results in under 70000 unique contracts. The 5000 contract sample is taken from this set. Note that the sample size is quite large, in relation to the total population (under 70K). Even the weakest possible statistical measurement still has a confidence interval of 1.76%, with 99% confidence.[9]

Decompilers for EVM bytecode may have different goals but generally aim to recover a structured representation of the code for analysis and human understanding purposes, not for purposes

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

[8] Other Ethereum decompilers, such as EtherVM [eth 2018], Porosity [Various 2018b], or Vandal [Brent et al. 2018], are currently decidedly inferior or explicitly unmaintained—EtherVM is the most up-to-date of them, yet its last non-trivial code commits are from Sep. 2018. We are not considering the JEB decompiler [Falliere 2019] and Trustlook SECaaS [TrustLook 2019] as they are closed-source tools that offer no way to perform large experimental evaluations in their free versions. The recent EtherSolve tool [Contro et al. 2021] is not considered as it does not satisfy our definition of "decompilation" from Section 1: it produces an IR containing low-level stack-altering opcodes (i.e. POP, DUP, SWAP), not a structured IR, such as three-address code over variables.

[9] Specifically, for a uniform random sample of 5000 over a population of 70K, the measurement with the least confidence is one at exactly 50%. (E.g., if we were to discover that exacty 50% of the contracts can be decompiled.) For that, there is a 1.76% confidence interval (i.e., the real number will be between 48.24% and 51.76%) with 99% confidence. An online sample-size calculator can be handy for confirmation—e.g., *https://www.surveysystem.com/sscalc.htm*.

of maintaining the decompiled code as a development artifact. Therefore the output may be significantly different from the original source language—for instance, Panoramix decompiles into Python-esque syntax, Gigahorse primarily produces 3-address code. Comparing across dissimilar representations is not trivial, therefore we try to introduce the right metrics per-case.

7.1 Comparison to Gigahorse

Gigahorse is the leading Ethereum decompiler in the research literature [Grech et al. 2019a], as well as a publicly deployed tool—used in the *contract-library.com* Ethereum code-browsing and analysis service. Gigahorse was extensively evaluated against other (now unmaintained) decompilers such as Vandal [Brent et al. 2018] and Porosity [Various 2018b]. We compare Elipmoc directly to the Gigahorse public artifact [Grech et al. 2019b], which is also the base upon which Elipmoc has been built. Both decompilers share a similar architecture but have greatly dissimilar algorithms, which yield different levels of precision, completeness, and scalability. Sharing the same technical infrastructure allows us to normalize our experiments better and delve into deeper, fundamental metrics. Both decompilers are designed primarily to enable precise and complete security analyses on top of their IR.

Enabled by this architectural similarity, our first experiment takes a detailed look at individual measurements at the decompilation level. In order to do so a second experiment was conducted, recording the following metrics:

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).
Block in Multiple Functions: Basic blocks that belong to more than one function.
Polymorphic Jump Target: (intra-procedural) Jump instructions with targets not uniquely resolved under the same context.
Timeout: Contracts that have not been decompiled due to timeouts.
Execution Time: Mean decompilation time in seconds.

Most of these metrics in fact address the precision research question (RQ1): *Polymorphic Jump Target* measures the precision of the global analysis level, while *Block in Multiple Functions* and *Unstructured Control Flow* measure the precision of the output IR. *Unresolved Operand* addresses both precision (RQ1) and completeness (RQ2): unresolved operants manifest as incompleteness in the final output, but can be caused by either imprecision at any decompilation stage (e.g., global analysis, function boundary inference), which leads the analysis to consider unrealizable behaviors, or by inferring an incorrect number of arguments in a private function. *Timeout* and *Execution Time* address RQ3.

The results of our experiment using the default parameters of each system are shown in Figure 9. For each metric, we report the *percentage* of contracts in our sample that exhibit the measured pathology: either imprecision/incompleteness, or lack of scalability. For instance, the figure shows that, under Gigahorse, 37.2% of the sampled contracts exhibit at least one operator with unresolved operands, whereas this number drops to 0.5% for Elipmoc.

As can be seen in Figure 9, Elipmoc significantly outperforms Gigahorse in all three (conflicting) quality indicators—scalability, precision and completeness. Since Elipmoc and Gigahorse share their architecture and are both implemented in Datalog, the improvements are directly attributable to the algorithms presented in this paper. In particular, the transactional context sensitivity and the new function reconstruction algorithms have delivered impressive precision improvements.

Elipmoc is also more scalable than Gigahorse, timing out for a mere 4.9% of the sampled contracts, to Gigahorse's 18.7%. In addition to their timeouts, Table 1 also contains the average decompilation

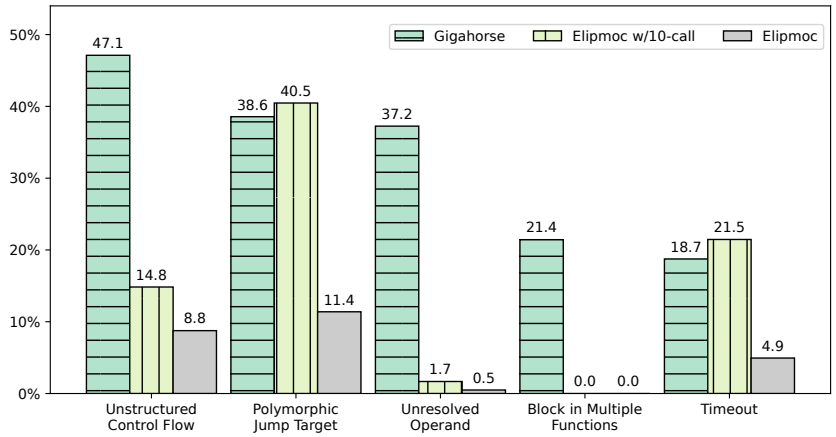

Fig. 9. Comparison between Gigahorse, Elipmoc, and a modified form of Elipmoc with the same context-sensitivity algorithm as Gigahorse. The latter serves to showcase what part of the improvement is due to better context sensitivity vs. other improvements. All metrics show the % of contracts (for Timeout over all contracts in our dataset, for the rest, over the common contracts all 3 tools/configurations manage to decompile) that exhibit the behavior measured—lower is better.

Table 1. Decompilation scalability metrics vs Gigahorse. The " / common" variants show pairwise statistics over the contracts for which neither setup times out.

	Timeouts (%)	*Execution Time* (average)
Elipmoc	4.94 %	2.74s
Gigahorse	18.74 %	4.03s
Elipmoc / common	—	1.23s
Gigahorse / common	—	3.94s

times for Elipmoc and Gigahorse. As can be seen, for the subset of contracts decompiled by both tools, Elipmoc is more than 3 times as fast on average.

The Gigahorse authors report a much lower, just 0.02%, timeout ratio [Grech et al. 2019a]. The discrepancy is due to two reasons. First, the lower timeout number is for a setting with a 1-call-site-sensitive analysis. In contrast, we measure with the default setting of the Gigahorse code [Grech et al. 2019b], which is tuned towards precision, with a more expensive 10-call site sensitivity. Second, our evaluation set uses more modern smart contracts, which are larger in size and complexity. The Gigahorse dataset consisted of all contracts deployed on the Ethereum blockchain from its creation, in July 2015, until April 2018. Such a dataset is more representative of an era when contract deployment was orders of magnitude cheaper (so the chain is full of toy experiments) and the mainsteam applications were simpler (simple tokens, DAOs). (In the past year alone, gas prices denominated in ETH grew by 2.5x and the cost of ETH in dollars grew by 7x, resulting in contract deployment becoming more than 15x more expensive.) On the contrary, the Elipmoc dataset contains contracts that were first deployed between April and September of 2021, focusing on a much more mature space including complex Decentralized Finance (DeFi) applications.

Figure 9 also presents results on a modified version of Elipmoc, which has the same context sensitivity algorithm as Gigahorse (10-call-site sensitivity). This quasi-Elipmoc decompiler also shows significant precision improvements relative to Gigahorse, indicating the positive impact of contributions such as path-sensitive function reconstruction. At the same time, it shows clearly the impact of better context sensitivity, when every other aspect of the decompiler remains unchanged.

Figure 1, in the Introduction, shows a sample of decompilation output (in human-readable form) where imprecision elements are illustrated.

Scalability and size distribution. To get better insights on the scalability of the two tools (RQ3), we compare the success rates of Elipmoc and Gigahorse for contracts of different sizes. The results are summarized in the following table:

Bytecode Size	[0,5KB)	[5KB,10KB)	[10KB,15KB)	[15KB,20KB)	[20KB,max)
Elipmoc	2547 (**99.8%**)	1023 (**96%**)	536 (**82.6%**)	254 (**86.1%**)	393 (**89.5%**)
Gigahorse	2538 (**99.5%**)	909 (**85.3%**)	405 (**62.4%**)	124 (**42.0%**)	87 (**19.8%**)
Total	2552	1065	649	295	439

The results of this comparison can help us get a better understanding of our dataset and the modern smart contracts for which it is representative. Over half of the contracts in it are smaller than 5 kilobytes; these small (often hand-tweaked) contracts are often wallet contracts deployed at scale (with different parameters, e.g., customer address) by exchange services or proxy contracts utilized by many DeFi protocols. The vast majority of these small contracts can be decompiled by both Elipmoc and Gigahorse. More diverse contracts that warrant analysis efforts are larger. Looking at the results for contracts larger than 5 kilobytes, the increased difficulty of decompiling modern contracts becomes apparent. Elipmoc handily outperforms Gigahorse across all different contract sizes.

7.2 Comparison to Panoramix

Panoramix (originally "Eveem") [Kolinko and Palkeo 2020] is an open-source EVM decompiler, the most-widely used, since it has been deployed in the foremost Ethereum blockchain explorer, *etherscan.io*. We compare to Panoramix commit `876a8de`.

Panoramix uses symbolic execution, so it decompiles programs as many different execution sequences and reconstructs a high-level representation of these. Therefore, the architecture of Panoramix and Elipmoc are significantly different and simple metrics are not useful when comparing the fidelity of the approaches. For instance, Panoramix does not infer private functions, so function-based metrics cannot apply. Panoramix also regularly decompiles a limited number of execution sequences of a program, therefore entire branches of the program may be missing. On the other hand, since Panoramix uses a symbolic execution approach, it often inlines entire sequences of instructions or even unrolls loops. Combining these two properties leads Panoramix to produce programs with a larger number of variables, but capturing fewer behaviors. In order to have a meaningful comparison, we run queries on the decompiled representations and produce high-level metrics with end-user value. These measure completeness and are resilient to inlining/unrolling and different output forms.

An excellent such metric is the number of unique `CALL` instruction signatures (performing calls outside the contract to a function with the given signature) that are present in the decompiled output. This is a metric that is highly resilient to different approaches to decompilation: the same call could be inlined many times, yet will only be counted once. Similarly, no matter what the decompiler does (e.g., condense/optimize the presentation of input instructions), the call signatures should always be present in the decompiled output, if the code block containing them has been successfully covered (i.e., decompiled). The metric is also independently interesting, since many

Table 2. Decompilation completeness and scalability metrics vs Panoramix. "Panoramix Partial" cuts off the analysis of any function that times out (at the default 120sec) and continues with the rest of the functions. Decompilation is then considered to partially succeed if the total time taken is less than 300sec—2.5x the time allotted to Elipmoc or regular Panoramix. The " / common" variants show pairwise statistics over the contracts for which neither setup times out.

	Unique External Calls	*Unique Events*	*Timeouts* (%)	*Execution Time* (average)
Elipmoc	11816	11640	5.0%	2.75s
Panoramix	7048	7128	17.94%	15.6s
Panoramix Partial	9322	9030	7.12%	35.05s
Elipmoc / common	7307	7043	—	1.41s
Panoramix / common	6437	6686	—	14.12s
Elipmoc / common	9798	9104	—	1.62s
Panoramix Partial / common	8379	8381	—	28.59s

security analyses are queries that start with external calls (e.g., reentrancy, gas denial-of-service [Grech et al. 2018], or tainted calls [Brent et al. 2020]).

The full set of metrics used for this comparison includes:

- **Unique External Calls:** Number of unique call signatures found.
- **Unique Events:** Number of unique event signatures found.
- **Timeouts:** Percentage of contracts that have timed out or otherwise report a decompilation failure.
- **Execution Time:** Mean decompilation time in seconds.

Table 2 shows the results of the comparison of Elipmoc with Panoramix. The most striking metric is the number of unique external call signatures in the decompiled output. Elipmoc manages to discover 11816 signatures in all contracts, against 7048 for Panoramix—a 67% increase. This showcases Elipmoc's completeness/coverage: it roughly shows how much more code a user can decompile with Elipmoc. A similar result is apparent in the unique events metric, measuring the unique signatures of event-emitting instructions found in the decompilation output. Panoramix misses many such instructions due to incompleteness.

To analyze this result more deeply, Figure 2 also contains more detailed comparisons. First, we introduce the "Panoramix Partial" setup, which allows Panoramix to continue decompilation even if a complex function times out: decompilation moves on to the next function and is considered to terminate if the overall time does not exceed 300sec (2.5x the default timeout). Even this setup, with 2.5x the global timeout of Elipmoc, misses a large number of externally-observable instructions (calls and events).

There are two reasons why Elipmoc recovers more externally-visible instructions. It decompiles more contracts, and it decompiles *more code in the same contracts*. To quantify the latter, Figure 2 shows the results of pairwise comparisons with Panoramix and Panoramix-Partial over the "common" contracts decompiled successfully by both systems being compared. The Elipmoc decompilation exhibits significantly higher completeness (of 13% and 17% on the External Calls metric) even when considering contracts that Panoramix also decompiles.

The overall result can be seen as indicative of the superior completeness of a static analysis approach, relative to a symbolic execution approach. For applications such as decompilation, where *all* input code needs to participate in the output, symbolic execution is likely easier in engineering

terms but cannot approach the completeness of covering all possible branches via exhaustive static analysis.

Elipmoc additionally exhibits both higher speed (5x lower average execution time) and greater scalability/robustness than Panoramix. Elipmoc times out in 5% of the contracts [10], versus Panoramix, which either times out or registers errors for 17.9%.

Scalability and size distribution. To further examine the scalability of the two tools (RQ3), we compare the success rates of Elipmoc and Panoramix for contracts of different sizes. The results appear in the following table:

Bytecode Size	[0,5KB)	[5KB,10KB)	[10KB,15KB)	[15KB,20KB)	[20KB,max)
Elipmoc	2547 (**99.8%**)	1022 (**96%**)	535 (**82.4%**)	253 (**85.8%**)	393 (**89.5%**)
Panoramix	2483 (**97.3%**)	925 (**86.9%**)	455 (**70.1%**)	160 (**54.2%**)	80 (**18.2%**)
Total	2552	1065	649	295	439

As can be seen, Panoramix success rates drop rapidly for larger contracts, which are the ones more likely to require automated analysis. For the largest size class, contracts with at least 20KB of bytecode, the Panoramix success rate drops to just 18%.

Decompilation Output. Informally, Elipmoc and Panoramix produce code that is roughly equally compact. Occasionally Panoramix (which uses a Python-esque syntax and not Solidity-like) condenses simple expressions better, but most of the reduced output size is due to incompleteness, for large contracts. Looking at the median and mean sizes of the output produced by the two tools, gives us interesting insides on their inner workings. The *median* output size for Elipmoc is 6579 bytes and 4568 for Panoramix. However, if we were to take the *mean* instead of the *median*, the picture would be highly skewed, at 33545 bytes for Panoramix vs. 10354 for Elipmoc. This is due to enormous outliers, however—e.g., the largest output for Panoramix (due to over-eager loop unrolling) is 1.4MB, for a contract that Elipmoc decompiles into under 14KB.

Figure 10 demonstrates the source-like output that Elipmoc and Panoramix produce for the continuation of the `transfer` function snippet from Figure 1. Panoramix completely lacks private function inference (no `SafeMul`, `SafeDiv` calls), as if all private calls have been inlined. This leads to significant increase in output size, which can harm readability.

7.3 Design Decisions

Elipmoc's comparison to Gigahorse in Figure 9 has already shown the importance of better context sensitivity. We next briefly investigate the design decisions of Elipmoc's transactional context sensitivity and the contributions of each component towards precision (RQ1), completeness (RQ2) and scalability (RQ3). We try the following two modifications to the algorithm:

- Removal of the public function context component, keeping only the private function component.
- Using the public function component, replacing the private function one with a 10-call site sensitivity component.

We compare these modified context sensitivities with the original, full transactional context sensitivity and 10-call site sensitivity.

As shown in Figure 11, the full transactional context sensitivity performs better than any combination of its components. Both of the modified contexts perform better than Gigahorse's 10-call site sensitivity across all 3 of the (precision and scalability) metrics displayed. Looking more closely, the introduction of the public function component provides a big precision improvement, and a lesser, though significant, reduction of timeouts. On the other hand, the private function

[10]The timeout rate for Elipmoc rises slightly, from 4.9% to 5.0%, compared to Section 7.1, since the optional source unparser module is now required: comparisons with Panoramix can only be done on high-level output.

```
else {
  v3 = _SafeMul(_commission, varg1);
  v4 = _SafeDiv(_ethPerToken, v3);
  v5 = _SafeSub(varg1, _balanceOf[msg.sender]);
  _balanceOf[msg.sender] = v5;
  v6 = _SafeSub(varg1, _totalSupply);
  _totalSupply = v6;
  v7 = msg.sender.call().value(v4).gas(!v4 * 2300);
  require(v7);
  v8 = _SafeSub(_commission, 100);
  v9 = _SafeDiv(100, v4);
  v10 = _SafeMul(v8, v9);
  v11 = _exitWallet.call().value(v10).gas(!v10 * 2300);
  require(v11);
  emit Transfer(msg.sender, _rubusOrangeAddress, varg1);
  emit MoreData(v4, _ethPerToken);
  v2 = v12 = 1;
}
return v2;
```

(a) Code decompiled by Elipmoc

```
else:
  if _value:
      require _value
      require _value * withdrawCommission / _value == withdrawCommission
  require priceEthPerToken
  require _value <= balanceOf[caller]
  balanceOf[caller] -= _value
  require _value <= totalSupply
  totalSupply -= _value
  call caller with:
     value _value * withdrawCommission / priceEthPerToken wei
       gas 2300 * is_zero(value) wei
  require ext_call.success
  require withdrawCommission <= 100
  if _value * withdrawCommission / priceEthPerToken / 100:
      require _value * withdrawCommission / priceEthPerToken / 100
      require (100 * _value * withdrawCommission / priceEthPerToken / 100) - (withdrawCommission * _value *
     withdrawCommission / priceEthPerToken / 100) / _value * withdrawCommission / priceEthPerToken / 100 == -
     withdrawCommission + 100
  call exitWalletAddress with:
     value (100 * _value * withdrawCommission / priceEthPerToken / 100) - (withdrawCommission * _value *
     withdrawCommission / priceEthPerToken / 100) wei
       gas 2300 * is_zero(value) wei
  require ext_call.success
  log Transfer(
        address from=_value,
        address to=caller,
        uint256 tokens=rubusOrangeAddress)
  log 0xab7f846d: _value * withdrawCommission / priceEthPerToken, priceEthPerToken
return 1
```

(b) Code decompiled by Panoramix

Fig. 10. Comparison of Elipmoc and Panoramix output. Slight simplification (names, casts) to fit space.

context alone greatly improves scalability while also improving precision. (Recall that the private function context contains a small subset of the jump sites, based on an *a priori* classification of jump instructions.) The marriage of the two components produces significant benefits, however.

8 RELATED WORK

The popularity of the Ethereum platform and the fact that most deployed smart contracts only have low-level EVM bytecode resulted in the emergence of many EVM decompilation tools from both academia [Brent et al. 2018; Grech et al. 2019a; Zhou et al. 2018] and industry [eth 2018; Falliere

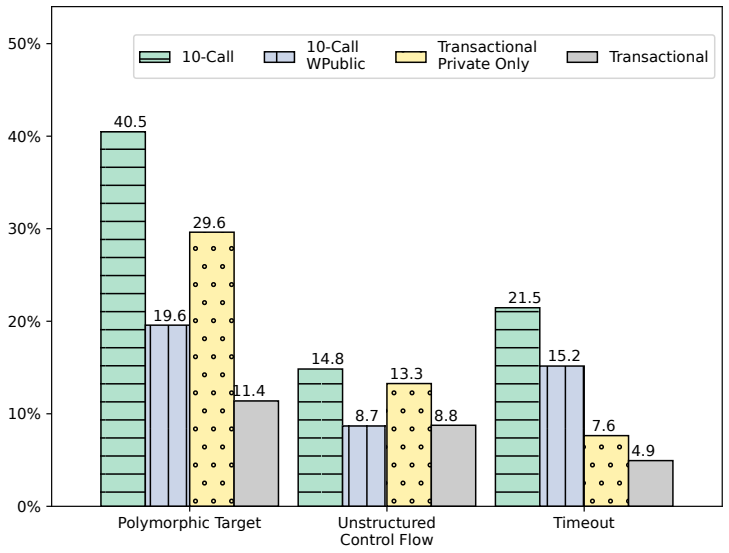

Fig. 11. Impact of the various components of transactional context sensitivity on performance and precision.

2019; Kolinko and Palkeo 2020; TrustLook 2019]. We earlier compared to the tools that have a current claim of being "the best".

The recent EtherSolve [Contro et al. 2021] tool builds a global CFG as a building block for static analyses. Its modeling of the stack is, however, limited to the parts of it that affect the construction of the CFG, producing an IR that contains stack-altering instructions. This design decision dictates that security analyses built on top of EtherSolve will have to model the stack in order to obtain accurate data-flow and value-flow information, effectively limiting its usability as an out-of-the-box static analysis framework. This conclusion can be supported by the fact that most published static analysis research targetting the EVM [Brent et al. 2020; Grech et al. 2018; Tsankov et al. 2018] has been developed on top of well-formed three-address-code representations.

Recently, the SigRec [Chen et al. 2021] tool has been proposed to recover the public function signatures for smart contracts when their source-code is not available. SigRec proposes the use of *type-aware symbolic execution (TASE)* for the detection of known patterns used by the Solidity and Vyper compilers in order to infer the types of public function parameters. This is orthogonal to our work which focuses on providing a precise and complete IR, including the reconstruction of internal functions. The detection of such patterns can be performed on top of the IR produced by Elipmoc as done in the recent Ethereum "memory" analysis [Lagouvardos et al. 2020], which supports a subset of the patterns supported by SigRec while providing a general model for the different "memory" buffers of the EVM.

A plethora of Java bytecode decompilers [Benfield 2020; Dupuy 2020; Gómez-Zamalloa et al. 2009; Miecznikowski and Hendren 2002; Proebsting and Watterson 1997; Strobel 2020; Various 2020] have been developed over the years. The Dava [Miecznikowski and Hendren 2002] decompiler that is part of the Soot framework [Vallée-Rai et al. 1999] employs AST-level transformations in order to improve the quality of the decompiled output. It is important to realize that decompilation over Java bytecode is technically an entirely dissimilar problem to EVM decompilation. Java bytecode contains large amounts of high-level information (types, classes, methods, arrays, calls, jumps only to known labels) and enforces strict constraints (stack depth and type of contents are invariant

for every path reaching a program point). As such, the challenge of Java bytecode decompilers is how fully they can recover the rich idioms of the source syntax, not whether they can produce a high-level program in the first place. Still, works [Hamilton and Danicic 2009; Harrand et al. 2019] evaluating different Java bytecode decompilers have shown that providing syntactically correct and semantically equivalent Java code for realistic programs is still an open issue.

Binary disassembly [Ben Khadra et al. 2016; Flores-Montoya and Schulte 2020; Kruegel et al. 2004] and decompilation [Brumley et al. 2013; Cifuentes 1994; Katz et al. 2018; Van Emmerik 2007; Yakdan et al. 2016; Yakdan et al. 2015] are closer to the EVM decompilation problem, although control flow is still more disciplined than in the EVM. Many standard techniques had already been developed in the mid-90s [Cifuentes 1994]. The focus is on the x86 architecture, which is in some ways even easier to decompile once a reliable disassembly is produced. Inferring function boundaries and arguments in this domain is aided by standard calling conventions, ISA support of function calls and returns and a standardized call stack structure. Closer in spirit to our work, the Ddisasm [Flores-Montoya and Schulte 2020] tool implements a disassembler for x64 binaries written in Datalog, and the OOAnalyzer [Schwartz et al. 2018] system uses a Prolog-based reasoning system in order to recover C++ abstractions.

9 CONCLUSIONS

We presented Elipmoc, a decompiler for Ethereum VM bytecode. Elipmoc integrates high-precision algorithms and design decisions that target a balance of precision and scalability. As a result it decidedly advances the state of the art in a technically very challenging domain. The core of Elipmoc includes algorithms for reconstructing high-level control flow and function structure from an optimized CPS representation—a problem that is of unexpectedly high value in the context of the EVM and is so fundamental that will likely also arise in future domains.

ACKNOWLEDGMENTS

We gratefully acknowledge funding by the Hellenic Foundation for Research and Innovation (HFRI project DEAN-BLOCK).

REFERENCES

2018. Online Solidity Decompiler. http://ethervm.io/decompile

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

Lee Benfield. 2020. CFR - another java decompiler. https://www.benf.org/other/cfr/

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

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]

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

JP Morgan Chase. 2020. Quorum: A permissioned implementation of Ethereum supporting data privacy. https://github.com/jpmorganchase/quorum

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

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..

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

Dedaub. 2019. Rising Gas Prices are Threatening our Security (no, it's not the Saudi attack). https://medium.com/dedaub/rising-gas-prices-are-threatening-our-security-no-its-not-the-saudi-attack-4b7aa4878e83

Dedaub. 2021a. EIP-3074 Impact Study. https://docs.google.com/document/d/1itvPn7BhZ9N8h27d1Ig5C86_FZpyG5_cdpsuPJYmb-o/edit?usp=sharing

Dedaub. 2021b. Ethereum Pawn Stars: '$5.7M in hard assets? Best I can do is $2.3M'. https://medium.com/dedaub/ethereum-pawn-stars-5-7m-in-hard-assets-best-i-can-do-is-2-3m-b93604be503e

Dedaub. 2021c. Killing a Bad (Arbitrage) Bot ... to Save its Owners. https://medium.com/dedaub/killing-a-bad-arbitrage-bot-f29e7e808c7d

Dedaub. 2021d. Look Ma', no source! Hacking a DeFi Service with No Source Code Available. https://medium.com/dedaub/look-ma-no-source-hacking-a-defi-service-with-no-source-code-available-c40a6583f28f

Dedaub. 2021e. Verkle Gas Cost Changes Insights. https://docs.google.com/document/d/1s3qqzbkQFPcNvhzKPdnxg3MlFbv0YjK1z02SxRtdMs8/edit#heading=h.sldu0oqtgkoq

Dedaub. 2021f. Yield Skimming: Forcing Bad Swaps on Yield Farming. https://medium.com/dedaub/yield-skimming-forcing-bad-swaps-on-yield-farming-397361fd7c72?source=friends_link&sk=d146b3640321f0a3ccc80540b54368ff

E. Dupuy. 2020. Java Decompiler. http://java-decompiler.github.io/

Nicolas Falliere. 2019. Ethereum Smart Contract Decompiler. https://www.pnfsoftware.com/blog/ethereum-smart-contract-decompiler/

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

Miguel Gómez-Zamalloa, Elvira Albert, and Germán Puebla. 2009. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Inf. Softw. Technol.* 51, 10 (Oct. 2009), 1409–1427. https://doi.org/10.1016/j.infsof.2009.04.010

Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019a. 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

Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019b. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. https://doi.org/10.5281/zenodo.2578692 Research artifact corresponding to ICSE'19 technical paper "Gigahorse: Thorough, Declarative Decompilation of Smart Contracts".

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

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

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

Immunefi. 2021. Harvest Finance Uninitialized Proxies Bug Fix Postmortem. https://medium.com/immunefi/harvest-finance-uninitialized-proxies-bug-fix-postmortem-ea5c0f7af96b

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

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.

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

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.

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

Michales, Jonah. 2021. Inside the War Room That Saved Primitive Finance. https://medium.com/immunefi/inside-the-war-room-that-saved-primitive-finance-6509e2188c86

Jerome Miecznikowski and Laurie J. Hendren. 2002. Decompiling Java Bytecode: Problems, Traps and Pitfalls. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, London, UK, UK, 111–127.

http://dl.acm.org/citation.cfm?id=647478.727938

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.

Primitive Finance. 2021. PrimitiveFi post-mortem analysis. https://primitivefinance.medium.com/postmortem-on-the-primitive-finance-whitehack-of-february-21st-2021-17446c0f3122

Todd A. Proebsting and Scott A. Watterson. 1997. Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?). In *Proceedings of the 3rd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3* (Portland, Oregon) *(COOTS'97)*. USENIX Association, Berkeley, CA, USA, 14–14. http://dl.acm.org/citation.cfm?id=1268028.1268042

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

Olin Shivers. 1991. *Control-flow analysis of higher-order languages*. Ph. D. Dissertation. Carnegie Mellon University. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.75.2777&rep=rep1&type=pdf

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

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

Mike Strobel. 2020. Procyon. https://bitbucket.org/mstrobel/procyon/wiki/Java%20Decompiler

Rei Thiessen and Ondřej Lhoták. 2017. Context Transformations for Pointer Analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 263–277. https://doi.org/10.1145/3062341.3062359

TrustLook. 2019. Smart Contract Guardian - Trustlook SECaaS. https://www.trustlook.com/services/smart.html

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

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13–. http://dl.acm.org/citation.cfm?id=781995.782008

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

Various. 2017. GitHub - vyperlang/vyper: Pythonic Smart Contract Language for the EVM. https://github.com/ethereum/solidity

Various. 2018. GitHub - ethereum/solidity: The Solidity Contract-Oriented Programming Language. https://github.com/ethereum/solidity

Various. 2018a. GitHub - OpenZeppelin/openzeppelin-contracts: OpenZeppelin Contracts is a library for secure smart contract development. https://github.com/OpenZeppelin/openzeppelin-contracts

Various. 2018b. Porosity – a decompiler for EVM bytecode into readable Solidity-syntax contracts. https://github.com/comaeio/porosity

Various. 2020. Fernflower. https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine

Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. http://gavwood.com/paper.pdf.

K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. 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.

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

Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. 2018. Erays: Reverse Engineering Ethereum's Opaque Smart Contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1371–1385. https://www.usenix.org/conference/usenixsecurity18/presentation/zhou