EVMpress: Precise Type Inference for Next-Generation EVM Decompilation

Jung Hyun Kim[1][0009−0007−2575−9506], Soomin Kim[2][0000−0003−3129−3857], Jaeseung Choi[3][0000−0002−5493−9174], and Sang Kil Cha[1][0000−0002−6012−7228]

[1] KAIST
{jidoc01,sangkilc}@kaist.ac.kr
[2] KAIST CSRC
soomink@kaist.ac.kr
[3] Sogang University
jschoi22@sogang.ac.kr

Abstract. Analyzing EVM bytecode is imperative because nearly 45% of smart contracts on the Ethereum blockchain lack publicly available source code. While type inference is pivotal for EVM bytecode analysis, it remains unsolved because (1) current tools can only handle a subset of Solidity expressions, and (2) they often produce imprecise results due to unsound heuristics they employ. Furthermore, there is no comprehensive dataset with precise ground truth for evaluating EVM type inference, which hinders the development of new tools and the evaluation of existing ones. Thus, we propose `EVMpress`, a novel bytecode analysis framework that enables accurate type inference for Solidity expressions found in EVM bytecode. We evaluate `EVMpress` on the largest-to-date dataset of EVM bytecode containing more than 370K real-world contracts with precise ground truth for every function and variable. Our evaluation results show that `EVMpress` significantly outperforms existing state-of-the-art tools in terms of its coverage and accuracy. We publicize our dataset as well as our implementation of `EVMpress` to facilitate future research in EVM bytecode analysis.

1 Introduction

Ethereum Virtual Machine (EVM) bytecode analysis is essential for understanding the behavior of smart contracts. Developers deploy smart contracts to the Ethereum blockchain as EVM bytecode, typically *without* including their source code. Indeed, nearly 45% of the smart contracts in use do *not* disclose their source code [6], highlighting the need for EVM bytecode analysis.

Type inference is a crucial step in EVM bytecode analysis as it provides essential information about the types of Solidity expressions, such as functions and variables. Faulty type inference can yield misleading decompiled code [28,31], making it much harder to understand smart contracts. Also, a recent study [11] indicates that type inference affects the readability of decompiled code.

Unfortunately, existing EVM type inference techniques remain *immature* in terms of coverage and accuracy.

First, current EVM type inference techniques can only cover a subset of Solidity expressions. For example, `DeepInfer` [32] can only infer types of public functions, while `VarLifter` [22] can only infer types of global variables. To our knowledge, there is no existing technique that can infer types of private functions.

Second, existing type inference techniques suffer from low accuracy due to the unsound heuristics they employ. For instance, `VarLifter` [22] heuristically considers a single path for each function to infer global variable types, leading to an underestimated result. `Gigahorse` [14] employs several heuristics to identify functions that do not follow the standard function call-return pattern, but these heuristics are often imprecise, leading to both false positives and negatives. Due to the imprecise function identification, `Gigahorse` often fails to recover types of critical functions as our study will show.

To make matters worse, there is no high-quality dataset with precise ground truth for EVM type inference, making rigorous evaluation difficult. Indeed, our preliminary study reveals that approximately 31% of the contracts in the dataset of `VarLifter` [22] are duplicates. Additionally, existing datasets do not provide precise ground truth for private functions and global variables. These issues overestimate the accuracy of existing EVM type inference techniques (see §2.2).

In this paper, we propose `EVMpress` to address all the aforementioned limitations of existing EVM type inference techniques in a unified framework. Specifically, we present a novel design of EVM bytecode analysis framework that enables precise type inference for various kinds of Solidity expressions including public functions, private functions, and global variables. Furthermore, we develop a large-scale dataset with precise ground truth to evaluate our system rigorously.

The key innovation of `EVMpress` is its holistic design that integrates function identification, Control Flow Graph (CFG) recovery, and type inference into a unified static analysis framework. Our design allows us to develop *path-based function identification*, a novel function identification technique that can accurately identify all kinds of functions, including public functions, private functions, as well as intrinsic functions generated by the Solidity compiler. The key insight of path-based function identification is that we can leverage a fundamental characteristic of functions in EVM bytecode: a function always pushes a return address before its entry and pops it after its return, regardless of the path it takes within the function. Therefore, we can always pinpoint a function entry point within a *single* execution path assuming there is a function call in the path. This approach allows us to accurately identify various types of functions, thereby significantly improving the accuracy of type inference.

Additionally, we construct a large-scale dataset with precise ground truth, consisting of 370,745 unique contracts, in order to rigorously evaluate our system. We eliminate duplicate contracts by normalizing EVM bytecode to remove references that can change during compilation and deployment. We also thoroughly build ground truth for function entry points, function types, and variable types with compiler-generated information, such as abstract syntax tree (AST).

Our evaluation shows that `EVMpress` can infer types of Solidity expressions from EVM binaries in an accurate and efficient manner. Specifically, we demon-

Table 1: Comparison of the state-of-the-art EVM bytecode analysis tools.

	Open?	Identification			Type Inference		
		Pub. Func.	Priv. Func.	Glob. Var.	Pub. Func.	Priv. Func.	Glob. Var.
`Neural-FEBI` [15]	✓†	✓	✓	✗	✗	✗	✗
`Gigahorse` [13, 14, 19]	✓	✓	✓	✗	✓‡	✗	✗
`SigRec` [7]	✗	✓	✗	✗	✓‡	✗	✗
`DeepInfer` [32]	✓†	✓	✗	✗	✓	✗	✗
`VarLifter` [22]	✓	✓	✗	✓	✗	✗	✓
`EVMpress`	✓	✓	✓	✓	✓	✓	✓

† ✓ means *partially* available. `Neural-FEBI` publicize their code and dataset, but not their model parameters. `DeepInfer` lacks their preprocessing module in their code release.
‡ ✓ indicates *partial* support, e.g., `Gigahorse` does not infer return types of public functions.

strate that `EVMpress` can recover the types of previously unhandled private functions with over 96% accuracy. Notably, `EVMpress` finds 30% more private functions than `Gigahorse`, while showing 2× more accurate global variable type inference results than `VarLifter`. Additionally, `EVMpress` achieves 1.6× faster CFG recovery performance on average than existing techniques, and up to 4× faster on large bytecodes. Our main contributions are:

- We propose path-based function identification technique that enables accurate identification of functions in EVM bytecode.
- We design and implement `EVMpress`, a unified EVM bytecode analysis framework that incorporates path-based function identification technique.
- We present the largest-to-date dataset of EVM bytecode with precise ground truth for function and variable types, consisting of 370,745 unique contracts.
- We publicize our tool along with our dataset to support open science[4].

2 Motivation

In this section, we first discuss the limitation and scope of the current state-of-the-art EVM bytecode analysis tools. We then review the limitation of existing datasets for evaluating EVM bytecode analysis tools, and introduce our new dataset. Finally, we present a preliminary study that we performed with a real-world contract in our dataset to motivate our work.

2.1 Previous Tools

We studied five state-of-the-art EVM bytecode analysis tools: `NeuralFEBI` [15], `SigRec` [7], `DeepInfer` [32], `Gigahorse` [13,14,19][5], and `VarLifter` [21]. Table 1

[4] `https://github.com/SoftSec-KAIST/EVMpress`
[5] Although `Gigahorse` [13], `Elipmoc` [14], and `Shrnkr` [19] are tools presented in three different papers, they are continuous work on the same framework and share the same codebase. Therefore, we refer to them as `Gigahorse` for the sake of simplicity.

Table 2: Comparison of the existing datasets.

	Open?	# Orig.	# Dedup.	Ground Truth					
				Identification			Type Inference		
				Pub. Func.	Priv. Func.	Glob. Var.	Pub. Func.	Priv. Func.	Glob. Var.
`Neural-FEBI` [15]	✓	39.0K	22.5K	✓	✓	✗	✗	✗	✗
`Gigahorse`$_G$ [13]	✗	91.8K	–	–	–	–	–	–	–
`Gigahorse`$_E$ [14]	✗	5.0K	–	–	–	–	–	–	–
`Gigahorse`$_S$ [19]	✗	8.0K	–	–	–	–	–	–	–
`SigRec` [7]	✗	119.1K	–	–	–	–	–	–	–
`DeepInfer` [32]	✓	47.8K	47.8K	✓	✗	✗	✓	✗	✗
`VarLifter` [21]	✓	34.8K	23.9K	✗	✗	✓	✗	✗	✓
`EVMpress`	✓	370.7K	370.7K	✓	✓	✓	✓	✓	✓

compares these tools in terms of their features and capabilities. The second column indicates whether tool is publicly available. Note that `Neural-FEBI` and `DeepInfer` only partially publicize their source code or model. Columns three to five in Table 1 indicate which Solidity expressions are identified by each tool. Columns six to eight show whether each tool infers the types of the identified expressions. For functions, we distinguish between public and private functions. In EVM, public functions are those declared with the `external` or `public` keyword in Solidity. Private functions include those marked as `internal` or `private`, as well as compiler-generated intrinsic functions. For variables, we only consider global variables, which are stored in the *storage* area [29], and this is because none of the current tools can recover the types of local variables and extracting ground truth for local variables is challenging.

It is worth noting that none of the tools listed in Table 1 comprehensively recovers all kinds of expression types. `Neural-FEBI` and `Gigahorse` are the only tools that identify the locations of private functions, although they do not recover their types. While most tools support recovering parameter types of public functions, only `DeepInfer` can recover their return types. Rather surprisingly, none of the existing tools can infer the types of private functions, which can significantly limit their usability. We present the first EVM bytecode analysis framework that recovers types for all Solidity expressions listed in the table.

2.2 Previous Datasets

Table 2 summarizes the existing datasets used by the EVM bytecode analyzers listed in Table 1. Note that we distinguish the three datasets used by `Gigahorse` according to the papers in which they were introduced with the suffixes G, E, and S. The second column specifies dataset availability, revealing that among the seven previously proposed datasets, only three are publicly accessible. The third column presents the original number of contracts reported in the paper. Among public datasets, `DeepInfer`'s was the largest with 47.8K contracts. The fourth column shows the number of deduplicated contracts in each dataset obtained by our deduplication method described in §2.3. Notably, over 30% of the contracts

in both the `Neural-FEBI` and `VarLifter` datasets were duplicates, which could substantially distort evaluation results. The rest of the columns summarize what kind of ground truth information is provided in each dataset. We note that none of the publicly available datasets provides comprehensive ground truth. All these limitations summarized in the table motivate us to build a new dataset for EVM bytecode analysis that is (1) publicly available, (2) deduplicated, and (3) contains comprehensive ground truth for every function and variable type.

2.3 Our Dataset

We introduce the largest-to-date dataset for EVM bytecode analysis that addresses the aforementioned limitations, following the steps outlined below.

Contract Collection We collected all Ethereum bytecode deployed up to January 2025, which includes 68.7M contracts. We then filtered out self-destructed contracts and contracts with no incoming transactions to retain a meaningful dataset. Next, we selected the contracts whose Solidity source code is available on Etherscan [3] and can be compiled with the Solidity compiler version 0.4.11 or higher. This was because our ground truth generation process requires compiler versions higher than or equal to 0.4.11, the first release that introduces the `--standard-json` option for emitting Abstract Syntax Trees (ASTs). Lastly, we deduplicated the contracts that have identical EVM bytecode. Consequently, we were able to obtain 610K contracts along with their source code.

Contract Compilation Next, we compiled the collected source code from the previous step using the same compiler version and compiler options listed in Etherscan to reproduce the same bytecode deployed on the Ethereum blockchain while obtaining extra data to construct the ground truth. To obtain the extra data, we used the `--standard-json` option. For those compiled with the `--via-ir` option, which does not emit the compiled-generated intrinsic functions into the ASTs, we patched the Solidity compiler to emit them, and later used them to generate the ground truth. This step produces the EVM bytecode, assembly code, and ASTs for each contract.

Normalization and Deduplication Deduplicating contracts by bytecode alone is insufficient, as some differ only in addresses or auxiliary data, which are not relevant to the core execution logic of the contract. Thus, we further normalized the contracts in two steps. First, we removed auxiliary data from the bytecode. Next, we found the locations of address references in the bytecode, and zeroed them out to ignore the differences in addresses. As a result, 48.1% of contracts were duplicates, leaving 370K unique contracts.

Ground Truth Generation Finally, we generated the ground truth for each contract using the EVM bytecode, assembly code, and ASTs generated by the

```
1  contract GovernanceRouter {
2    function transferGovernor(uint32 _domain, ...) public {
3      // ...
4      formatTransferGovernor(_domain, ...);
5    }
6    function formatTransferGovernor(uint32 _domain, ...) private {
7      clone(mustBeTransferGovernor(ref(_domain, ...)));
8    }
9    function ref(...) private returns(...) {
10     if (...) { ... }
11     else { ... }
12   }
13   function mustBeTransferGovernor(...) private returns (...) {
14     ...
15   }
16   function clone(...) private returns (...) {
17     ...
18   }
19 }
```

Fig. 1: Simplified source code of the `GovernanceRouter` contract, which is located at address `0xfbea6d67ddd90e1f726c2622c6c42b016fdad5a7`.

compiler. Our ground truth includes (1) function addresses and their signatures, and (2) global variable locations and their types. To locate functions in the EVM bytecode, we analyzed the assembly code, which shows annotated assembly instructions with source code line information. Once we identified the entry point of each function, we checked the corresponding byte offset in the bytecode to obtain the function address. We then extracted from the corresponding AST the function signature, which includes the parameter types as well as the return type. We similarly identified the locations and types of global variables by analyzing the ASTs and the EVM bytecode based on the official documentation of the Solidity compiler [30]. Consequently, we obtained 370,745 unique contracts with the ground truth information for every function and global variable type. To the best of our knowledge, this is the *largest* dataset for EVM bytecode analysis with comprehensive ground truth information.

2.4 Motivating Example

Can the new dataset provide new insights into the limitations of existing EVM bytecode analysis tools? As we will show in the evaluation section §4, our dataset reveals that the state-of-the-art EVM bytecode analysis tools exhibit significantly low accuracy in identifying functions. For example, we found that the F1 score of `Gigahorse` in terms of detecting private function is only 71.0%.

Figure 1 illustrates one such example, named the `GovernanceRouter` contract. It includes the `formatTransferGovernor` function (Line 6), which is called by the `transferGovernor` function (Line 4). In Line 7, the `formatTransferGovernor` function calls three private functions in sequence: `ref`, `mustBeTransferGovernor`, and `clone`. Unfortunately, `Gigahorse` fails to identify the `clone` function despite the simple structure of the `formatTransferGovernor`

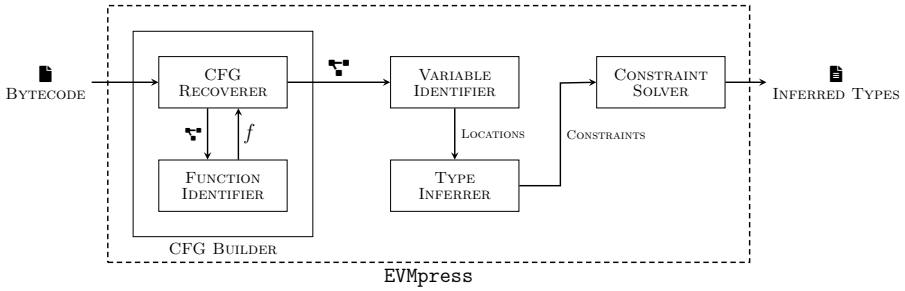

Fig. 2: `EVMpress` architecture.

function—no loops nor conditionals. The primary reason for this failure is that `Gigahorse` requires a function entry point to be reachable from multiple call sites, which is not the case for `clone`. We note that `Gigahorse` is prone to such errors because it relies on declarative patterns found in the EVM bytecode to identify functions. This limitation motivates us to develop a unified static analysis framework for EVM bytecode that can procedurally identify functions without relying on such patterns.

3 Design

This section presents the design of `EVMpress`. We first present the overall architecture of `EVMpress`. We then describe the details of path-based function identification. Finally, we conclude this section by illustrating the overall workflow of `EVMpress` using a running example.

3.1 Overview

Figure 2 shows the overall architecture of `EVMpress`, which takes in an EVM binary as input, and outputs the inferred types of functions and variables. At a high level, `EVMpress` runs in four main steps. CFG Builder first identifies functions and reconstructs the Control-Flow Graph (CFG) of each function from the given EVM bytecode. Subsequently, Variable Identifier identifies variables used by each function and returns the locations of the identified variables to Type Inferrer. A variable here is any recoverable high-level construct–function parameters, return values, local variables, and global variables. Variable Identifier identifies variables used by each function and returns their locations to Type Inferrer. Type Inferrer then collects type constraints for each identified variable and returns them to Constraint Solver. Finally, Constraint Solver solves the constraints and infers the types of each variable and outputs the results.

CFG Builder reconstructs the intra-procedural CFG for each function from the given EVM bytecode. It first recursively traverses the control flow of the

EVM bytecode starting from the entry point of the contract. Every time a jump instruction is encountered, CFG Builder consults with Function Identifier to check if the jump instruction is a function call. This is where our path-based function identification comes into play, which enables robust function identification from highly optimized EVM bytecode (§3.2).

Variable Identifier analyzes every data access in the CFG and identifies locations of variables used by each function. It returns the location information of identified variables, such as calldata, memory, and storage as well as their addresses or offsets, to Type Inferrer.

Type Inferrer collects type hints for each identified variable to generate type constraints. Specifically, we analyze variable access patterns by traversing the CFG as in the classic approaches [7,20]. For instance, when a variable is accessed by applying bit-masking operations, we can infer the bit-width of the variable. Similarly, when a variable is used as a conditional, we can infer that the variable is a boolean type.

Constraint Solver solves the type constraints generated by Type Inferrer and infers the types of each variable. We use the traditional (naive) bottom-up evaluation to saturate the constraint set [5].

3.2 Path-based Function Identification

The complexity of EVM bytecode optimization makes it challenging to identify function entry points, especially for private functions. Recall from §2.4 that previous approaches [13,14,19] suffer from both false positives and false negatives when identifying function entry points as they rely on declarative heuristics that are not robust against highly optimized EVM bytecode.

Our technique, on the other hand, leverages the invariant of a function call, which is that there must be a function entry point in any execution path between a return address definition and the corresponding return instruction. Suppose f and g are two functions in a contract where f is a caller of g, and g is a complex function containing many execution paths. Let $callsite(f, g)$ be the program point in f where g is called. When g returns to f using a jump instruction, the function entry point of g is always within an execution path between the $callsite(f, g)$ and the return instruction of g, no matter which path in g is taken. Therefore, we can simply investigate a single path, instead of all paths, to efficiently identify the function entry point of g. Depending on the complexity of g, our algorithm can significantly reduce the number of paths to be traversed, thus improving the efficiency of function identification.

At a high level, path-based function identification runs on every jump instruction encountered while recovering the CFGs (CFG Builder in Figure 2).

1. For every jump instruction encountered, we consider it as a potential return instruction r if there is no immediate jump target on top of the stack.
2. For each potential return instruction, we follow the use-def chain to find the definitions of the jump target address.

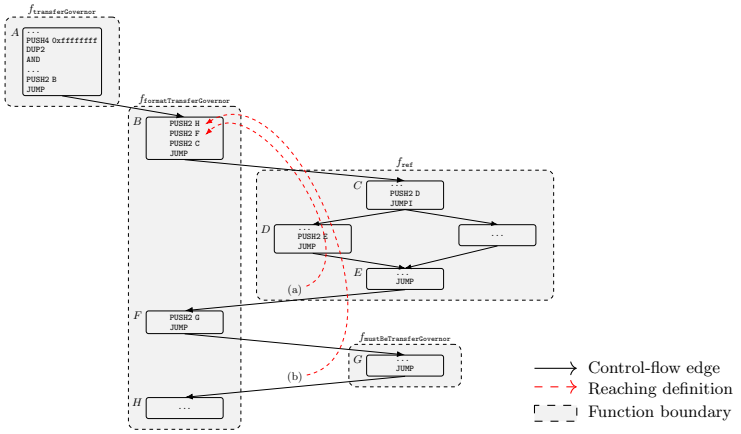

Fig. 3: CFG of the `GovernanceRouter` contract shown in Figure 1.

3. We then extract a random path from the definition to r and traverse it, excluding the first basic block, to identify whether there is a block that can be a function entry point. Specifically, a basic block in the path is a function entry point if it does not satisfy any of the following conditions:
   C1. The basic block has been identified as a return point of a function.
   C2. The basic block is included in the CFG of another function.
   C3. The basic block is a fall-through of another basic block.

3.3 Running Example

To illustrate how `EVMpress` works, let us consider the same example contract in Figure 1. We present the corresponding CFG in Figure 3. We denote each function in Figure 1 with $f_{\texttt{name}}$, where `name` is the name of the function. Note that we exclude the last function call (of `clone`) to simplify the illustration.

CFG Builder starts to analyze the given EVM bytecode. The very first step is to identify public functions by analyzing the function dispatcher located at the beginning of the EVM bytecode. In this example, CFG Builder identifies the public function `transferGovernor` defined in Line 2 of Figure 1. It then starts to disassemble the function to recover the basic block A of the CFG in Figure 3. When it encounters the jump instruction in A, it checks whether there is an *immediate* jump target, which comes from the same basic block, on top of the stack. Since there is a `PUSH` instruction right before the jump that pushes the address of B onto the stack, we know that this is *not* a returning edge. Therefore, we continue to disassemble the basic block B and perform the same analysis until we reach the basic block E, which does not include a push instruction before its jump instruction. At this point, CFG Builder follows the use-def chain (the red dotted arrow) of the jump target (a) to know that it jumps to the basic

block F. Since E does not have an immediate jump target, the jump instruction in E is considered as a potential return instruction.

Recall from §3.2 that we now select a random path from the definition site B to E and traverse it to identify whether there is a function entry point. Note that we have not yet identified any function except `transferGovernor`, which is a public function. Suppose we selected the path $B \to C \to D \to E$. We then traverse the path, excluding B, to check whether there is a basic block that can be a function entry point. The first block to consider is C, which does not satisfy any of the conditions listed in §3.2. Therefore, we identify C as a function entry point (of the function `ref`) and mark all reachable basic blocks from C to E as the CFG of the function.

Similarly, we can detect $f_{\texttt{mustBeTransferGovernor}}$ when we reach the basic block G. In this case, the reaching definition of the jump target (b) is at the basic block B because B pushes the address of H onto the stack even before it makes a call to $f_{\texttt{ref}}$. This is so-called Continuation Passing Style (CPS) function call [14], often found in EVM bytecode. Path-based function identification gracefully handles this case by selecting and analyzing a single path from B to G. Suppose we selected the path $B \to C \to D \to E \to F \to G$. Since C, D, and E satisfies condition C2, and F satisfies condition C1, we skip them and identify G as a function entry point. We follow the same procedure to identify all the CFGs in the bytecode, and proceeds to the next step, which is variable identification.

Variable Identifier analyzes the CFGs of the identified functions and identifies variables used by each function. For instance, we can easily detect function parameters by analyzing the stack before making a function call, as our path-based function identification can precisely identify which jump instruction is a function call. In this running example, Variable Identifier detects that the function `transferGovernor` passes the first parameter `_domain` to `formatTransferGovernor`. Thus, it returns the corresponding stack offset of the parameter to Type Inferrer.

Finally, Type Inferrer collects type hints for the given parameter `_domain` of `transferGovernor`. Specifically, it collects two type hints about the parameter. First, it identifies that `transferGovernor` computes the logical-and of `_domain` and a constant value, `0xffffffff` ($= 2^{32} - 1$), which indicates that `_domain` is a 32-bit number. Furthermore, it realizes that the computed value is never sign-extended with a `SIGNEXTEND` instruction, which indicates that `_domain` is an unsigned integer. These two type hints are then passed to Constraint Solver, which will conclude that `_domain` is a 32-bit unsigned integer, i.e., `uint32`.

3.4 Implementation

We implemented `EVMpress` with 10K SLoC of F#. We used B2R2 [16], a binary analysis framework, to disassemble and lift EVM instructions, and implemented CFG Builder as a middle-end module in B2R2. CFG Builder employs an incremental data-flow analysis [26] to track use-def chains of stack values on-the-fly, enabling it to identify jump targets and recognize patterns of public function

entry points. Variable Identifier uses an Static-Single Assignment (SSA) form of the Control-Flow Graph (CFG), which CFG Builder emits, to parse data access instructions (e.g., `SLOAD`, `CALLDATALOAD`, `MLOAD`) and identify variable locations. Type Inferrer also operates on an SSA-formed CFG to parse instructions that provide useful type hints (e.g., `AND` for bit-width, `SIGNEXTEND` for signed type) and collects constraints over the variables identified by Variable Identifier (e.g., `HasBitWidth(x, 160)`, `UsedAsSigned(y)`). We apply a set of rules, including memory aliasing and type inference rules, to propagate constraints until no new ones can be derived for each variable, eventually obtaining constraints that describe variable types (e.g., `HasType(x, uint32)`).

4 Evaluation

In this section, we evaluate `EVMpress` to answer the following research questions.

RQ1. How accurate is `EVMpress` in terms of function identification, variable identification, and type inference?
RQ2. How does `EVMpress` compare with the state-of-the-art tools in terms of function identification, variable identification, and type inference?
RQ3. Does path-based function identification enable efficient function identification?

4.1 Experimental Setup

Comparison Targets. Recall from §2.1 that only `Gigahorse` [13, 14, 19] and `VarLifter` [22] are fully available, although they do not support all the features that `EVMpress` supports. Thus, our evaluation focuses on comparing `EVMpress` with these two tools. We ran `Gigahorse` with the `--disable_inline` option, which prevents the default inlining of private functions. We found that this option improves `Gigahorse`'s function identification accuracy by about 10%. Particularly, we used `Gigahorse` commit `18ce2685` and `VarLifter` commit `47f9e681` for our evaluation.

Our Environment. We conduct our experiments on a server equipped with two 22-core Intel Xeon CPU E5-2699 processors and 512 GB RAM, running Ubuntu 22.04. We use 88 ($= 22 \times 2 \times 2$) virtual cores to run the tools in parallel.

4.2 RQ1: Accuracy Evaluation

To measure the accuracy of `EVMpress`, we ran it on our large-scale dataset introduced in §2.3. The last row of Table 3 shows the results of our evaluation. For function identification, we show precision, recall, and F1-score for both public and private functions. For variable identification and type inference, we show success rates.

Table 3: Comparison of function identification, variable identification, and type inference accuracy. Bold numbers denote the best results among the tools.

	Function Identification						Variable Identification					Type Inference				
	Public.			Private.			Public.		Private.			Public.		Private.		
	Prec. (%)	**Rec. (%)**	**F1 (%)**	**Prec. (%)**	**Rec. (%)**	**F1 (%)**	**Par. (%)**	**Ret. (%)**	**Par. (%)**	**Ret. (%)**	**Glob. (%)**	**Par. (%)**	**Ret. (%)**	**Par. (%)**	**Ret. (%)**	**Glob. (%)**
`VarLifter`	–	–	–	–	–	–	–	–	–	–	49.4	–	–	–	–	48.6
`Gigahorse`	94.9	99.4	97.1	78.1	65.0	71.0	82.8	–	**99.9**	**99.9**	–	82.8	–	–	–	–
`EVMpress`	**99.2**	**99.7**	**99.5**	**96.4**	**95.8**	**96.1**	**99.4**	**97.5**	98.6	99.4	**94.1**	**96.6**	**94.2**	**96.4**	**97.8**	**88.8**

Overall, `EVMpress` demonstrates high accuracy across all metrics, achieving 95% or higher in every case except for global variable type inference. Given that type inference is a challenging task and there are only a few existing tools that support it, 88.8% accuracy for global variable type inference is still a promising result. We further discuss why `EVMpress`'s global variable type inference is not as accurate as the other metrics in §4.3 when we compare the result with that of `VarLifter`.

4.3 RQ2: Comparison with State-of-the-Art Tools

We compared `EVMpress` with the two state-of-the-art tools, `Gigahorse` and `VarLifter`, in terms of function identification, variable identification, and type inference as shown in Table 3. It is important to note that `EVMpress` is the only tool capable of inferring the types of private functions and the return values of public functions.

`EVMpress` vs. `Gigahorse`. `EVMpress` outperforms `Gigahorse` in all the metrics, except for private function parameter and return value identification. Although `EVMpress` shows about 99% of accuracy, `Gigahorse` achieves slightly higher accuracy for private function parameter and return value. This is simply because `Gigahorse` recovers significantly fewer private functions than `EVMpress`—the recall of `Gigahorse` for private function identification is only 65%. As a result, `Gigahorse` only recovers types of the most basic private functions and misses many highly optimized ones. This leads to a slightly higher accuracy than `EVMpress`, but only because it analyzes a much smaller and simpler subset of private functions.

For public functions, both `EVMpress` and `Gigahorse` achieved higher than 99% recall in identifying them. However, `Gigahorse` showed significantly lower precision than `EVMpress` because it often misidentifies some intrinsic functions as public functions. Although both tools were able to identify most of the public functions, `Gigahorse`'s F1-score for identifying their parameters was only 82.8%, while `EVMpress` achieved 99.4%. This is because `Gigahorse` relies on an external database based on the hash of the function signature, which can lead to false negatives if the function signature is not present in the database.

Table 4: Averaged CFG recovery time and total execution time on our dataset.

	CFG Recovery Time	Total Execution Time	Error Rate
`Gigahorse`	4.5s	**4.6s**	0.04%
`EVMpress`	**2.7s**	8.3s	**0.02**%

`EVMpress` vs. `VarLifter`. `EVMpress` consistently outperforms `VarLifter` across all metrics. In particular, `EVMpress` achieves nearly twice the F1-score for global variable type recovery compared to `VarLifter`, which only reached 48.6% accuracy. This result is rather surprising, as the accuracy of `VarLifter` reported in its original paper [22] was 96.2%. There are several reasons for this discrepancy. First, we manually inspected the results of `VarLifter` and found that it *does not* recover the offset of global variables that have the same slot number. Second, `VarLifter` had difficulty recovering the types of both the key and value in mapping variables, which are commonly used in real-world contracts. Third, the authors used the highly duplicated dataset and its partial ground truth information (recall from §2.3).

Although `EVMpress` achieves 88.8% accuracy in global variable type inference, we further analyzed the causes of the remaining errors. We found that `EVMpress` struggles to distinguish between `bool` and `uint8` types, which have the same sizes and are often used interchangeably in the bytecode. Interestingly, by ignoring the distinction between these two types, we were able to achieve 95% accuracy in global variable type inference.

4.4 RQ3: Impact of Path-based Function Identification

Recall from §3.2 that path-based function identification is a key component of `EVMpress` that reduces the number of paths to be analyzed while precisely recovering the CFG. Having established the high accuracy of `EVMpress` in the previous evaluations, we now assess how path-based function identification specifically improves the runtime performance of `EVMpress`.

Table 4 shows the averaged time consumption of `EVMpress` and `Gigahorse` on our dataset. We exclude `VarLifter` from this comparison since it does not report CFG recovery time. The last column shows the runtime error rate of each tool. When the tool reaches the timeout, we consider it as a runtime error, too.

Notably, the CFG recovery time of `EVMpress` is nearly 66% faster than it of `Gigahorse`. The difference becomes even more significant on large bytecodes (of top 10K bytecodes by size in our dataset), where `EVMpress` achieves nearly 4× faster code recovery than `Gigahorse`. This result confirms that path-based function identification effectively reduces the number of paths to be analyzed by `EVMpress`, leading to faster CFG recovery.

As the third column of Table 4 shows, the total time consumption of `EVMpress` is slightly higher than that of `Gigahorse`. This slight increase is due to `EVMpress` performing a more comprehensive analysis than `Gigahorse`. Specifically, `EVMpress` recovers additional information such as the types of private function parameters,

private function return values, and public function return values, which `Gigahorse` does not support. Nonetheless, `EVMpress`'s total time is comparable to `Gigahorse`, which is a significant achievement, rendering `EVMpress` a practical tool for EVM bytecode analysis.

5 Discussion

While `EVMpress` advances the current state of EVM type recovery, it still has several limitations.

Transient Storage. Since Solidity 0.8.28, the language supports *transient storage*, a temporary and cheaper storage area. `EVMpress` does not yet support transient storage, but extending our approach to handle it is straightforward since its mechanism is similar to storage area. However, transient storage is not yet common in real-world contracts, and our dataset contains no such contracts.

Unused Variables. Our variable identification and type inference rely on the behavioral hints of variables found in EVM bytecode. Therefore, our approach cannot recover the types of variables that are defined but not used in the code. This limitation is common in type inference, and hence, is beyond the scope of this paper.

6 Related Work

There are numerous studies on identifying functions in traditional binary code. They often utilize pattern matching [4, 17, 23], compiler metadata [25], or probabilistic models [18] to identify functions in binary code.

However, it is not feasible to apply these techniques to EVM bytecode as it introduces additional challenges due to the lack of explicit call and return instructions, especially for private functions. `Gigahorse` [13, 14] leverage declarative code patterns such as Continuation-Passing Style (CPS) calls to detect private functions, while `Neural-FEBI` [15] employs a deep learning-based approach to identify private functions. On the other hand, `EVMpress` introduces a procedural approach to identify private functions, which is more efficient than declarative approaches and does not require a large training dataset like deep learning-based approaches.

A common approach for inferring public function parameters is to use prebuilt databases of known function selectors [1, 2, 12, 13]. While effective, this method cannot be applied to closed-source smart contracts. To address this, `SigRec` [7] symbolically analyzes the prologue of public functions, extracting argument variables and their types by tracking how calldata is loaded into memory. Other tools, such as `VarLifter` [22] and `Crush` [27], focus on recovering types of variables in the EVM storage area, which holds contract assets. However, because these tools are tailored to specific EVM data areas, they lack a unified analysis framework for all variable types. Deep learning-based approaches [32] have also been proposed to recover types of public function parameters and return values, but their effectiveness depends heavily on the quality of training data and they

require substantial computational resources. In contrast, `EVMpress` is the first tool to comprehensively recover all types of variables and their types in EVM bytecode, without incurring heavy computational costs.

The rise of DeFi and NFTs highlights the need for scalable EVM bytecode analysis to ensure smart contract security and reliability. Several tools [1, 2, 9, 10, 13, 14, 24] decompile EVM bytecode into human-readable code to aid manual and automated analysis. Although `EVMpress` is not a decompiler, it can serve as a pre-processor for decompilers as it can recover precisely functions and variables in EVM bytecode. Furthermore, our precise CFG recovery and type inference techniques can benefit vulnerability detection tools that leverage EVM bytecode analysis [8] by providing accurate information about functions and their types.

7 Conclusion

In this paper, we studied the current limitations of existing EVM bytecode analysis techniques and proposed a novel framework that addresses these limitations. We also presented the largest dataset for EVM bytecode analysis to date, comprising over 370K unique contracts with precise ground truth. We evaluated our framework on this dataset and demonstrated its effectiveness in recovering types of global variables and functions.

Acknowledgments. This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2025-02263143, Development of Cybersecurity Threat Response Technologies for Satellite Ground Stations).

References

1. Porosity (2017), `https://github.com/msuiche/porosity`
2. Eveem (2019), `https://eveem.org`
3. Etherscan (2025), `https://etherscan.io/`
4. Bao, T., Burket, J., Woo, M., Turner, R., Brumley, D.: ByteWeight: Learning to recognize functions in binary code. In: Proceedings of the USENIX Security Symposium. pp. 845–860 (2014)
5. Ceri, S., Gottlob, G., Tanca, L., Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases: An Overview. Springer (1990)
6. Chaliasos, S., Gervais, A., Livshits, B.: A study of inline assembly in solidity smart contracts. In: Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. pp. 1123–1149 (2022)
7. Chen, T., Li, Z., Luo, X., Wang, X., Wang, T., He, Z., Fang, K., Zhang, Y., Zhu, H., Li, H., et al.: Sigrec: Automatic recovery of function signatures in smart contracts. IEEE Transactions on Software Engineering **48**(8), 3066–3086 (2021)
8. Choi, J., Kim, D., Kim, S., Grieco, G., Groce, A., Cha, S.K.: Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In: Proceedings of the International Conference on Automated Software Engineering. pp. 227–239 (2021)

9. Contro, F., Crosara, M., Ceccato, M., Dalla Preda, M.: Ethersolve: Computing an accurate control-flow graph from ethereum bytecode. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC). pp. 127–137. IEEE (2021)
10. Dedaub: Dedaub bytecode decompiler (2025), `https://app.dedaub.com/decompile`
11. Eom, H., Kim, D., Lim, S., Koo, H., Hwang, S.: R2I: A relative readability metric for decompiled code. In: Proceedings of the 32nd ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 383–405 (2024)
12. ethervm: Online solidity decompiler (2024), `https://ethervm.io/decompile`
13. Grech, N., Brent, L., Scholz, B., Smaragdakis, Y.: Gigahorse: thorough, declarative decompilation of smart contracts. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 1176–1186. IEEE (2019)
14. Grech, N., Lagouvardos, S., Tsatiris, I., Smaragdakis, Y.: Elipmoc: Advanced decompilation of ethereum smart contracts. In: Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. pp. 1–27 (2022)
15. He, J., Li, S., Wang, X., Cheung, S.C., Zhao, G., Yang, J.: Neural-FEBI: Accurate function identification in ethereum virtual machine bytecode. Journal of Systems and Software **199**(C), 1–16 (2023)
16. Jung, M., Kim, S., Han, H., Choi, J., Cha, S.K.: B2R2: Building an efficient front-end for binary analysis. In: Proceedings of the NDSS Workshop on Binary Analysis Research (2019)
17. Kim, H., Lee, J., Kim, S., Jung, S., Cha, S.K.: How'd security benefit reverse engineers? the implication of intel CET on function identification. In: Proceedings of the International Conference on Dependable Systems and Networks. pp. 559–566 (2022). https://doi.org/10.1109/DSN53405.2022.00061
18. Kim, S., Kim, H., Cha, S.K.: FunProbe: Probing functions from binary code through probabilistic analysis. In: Proceedings of the International Symposium on Foundations of Software Engineering. pp. 1419–1430 (2023)
19. Lagouvardos, S., Bollanos, Y., Grech, N., Smaragdakis, Y.: The incredible shrinking context... in a decompiler near you. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (2025)
20. Lee, J., Avgerinos, T., Brumley, D.: TIE: Principled reverse engineering of types in binary programs. In: Proceedings of the Network and Distributed System Security Symposium. pp. 251–268 (2011)
21. Li, S., Su, Z.: Finding unstable code via compiler-driven differential testing. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 238–251 (2023). https://doi.org/10.1145/3582016.3582053
22. Li, Y., Song, W., Huang, J.: Varlifter: Recovering variables and types from bytecode of solidity smart contracts pp. 1–29 (2024)
23. Meng, X., Miller, B.P.: Binary code is not easy. In: Proceedings of the International Symposium on Software Testing and Analysis. pp. 24–35 (2016). https://doi.org/10.1145/2931037.2931047
24. MrLuit: Evm bytecode decompiler (2018), `https://github.com/MrLuit/evm`
25. Pang, C., Yu, R., Xu, D., Koskinen, E., Portokalidis, G., Xu, J.: Towards optimal use of exception handling information for function detection. In: Proceedings of the International Conference on Dependable Systems and Networks. pp. 338–349 (2021). https://doi.org/10.1109/DSN48987.2021.00046

26. Pollock, L.L., Soffa, M.L.: An incremental version of iterative data flow analysis. IEEE Transactions on Software Engineering **15**(12), 1537–1549 (1989)
27. Ruaro, N., Gritti, F., McLaughlin, R., Grishchenko, I., Kruegel, C., Vigna, G.: Not your type! detecting storage collision vulnerabilities in ethereum smart contracts. In: Proc. Netw. Distrib. Syst. Secur. Symp. pp. 1–17 (2024)
28. Schwartz, E.J., Lee, J., Woo, M., Brumley, D.: Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In: Proceedings of the USENIX Security Symposium. pp. 353–368 (2013)
29. Team, S.: Introduction to smart contracts (2025), `https://docs.soliditylang.org/en/latest/introduction-to-smart-contracts.html`
30. Team, S.: Layout of state variables in storage and transient storage (2025), `https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html`
31. Wiedemeier, J., Tarbet, E., Zheng, M., Ko, S., Ouyang, J., Cha, S.K., Jee, K.: PyLingual: Toward perfect decompilation of evolving high-level languages. In: Proceedings of the IEEE Symposium on Security and Privacy. pp. 2976–2994 (2025)
32. Zhao, K., Li, Z., Li, J., Ye, H., Luo, X., Chen, T.: Deepinfer: Deep type inference from smart contract bytecode. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 745–757 (2023)