Advancing EVM Decompilation: Enhancing Output via LLM-Driven Post-processing

Jonathan Becker
jonathan@jbecker.dev

December 9, 2024

Abstract

Decompiling Ethereum Virtual Machine (EVM) bytecode into high-level source code is crucial for smart contract security analysis, auditing, and comprehending the behavior of unverified contracts. This paper presents advancements in EVM decompilation techniques, combining traditional static analysis and symbolic execution methods with post-processing driven by Large Language Models (LLMs). By leveraging LLMs, we introduce a pipeline that refines decompiler output, improving code readability, type inference accuracy, and the reconstruction of complex control flows. Our approach mitigates common decompilation challenges — such as incomplete mappings between bytecode and source-level constructs — by providing semantic annotations and correcting structural ambiguities. Experimental evaluations on real-world contracts demonstrate that our LLM-guided post-processing can significantly enhance the clarity and correctness of the resulting code, ultimately enabling more robust security assessments and streamlined development workflows.

Keywords: EVM, Decompilation, Symbolic Execution, Static Analysis, LLM, Smart Contracts

Contents

1 Introduction

1.1 Brief: The Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) is a decentralized computing environment that serves as the backbone of the Ethereum blockchain. Introduced as part of the Ethereum protocol, the EVM enables the execution of self-executing and immutable[1] programs (known as smart contracts) in a decentralized and distributed manner, ensuring that smart contract execution is deterministic and consistent across all nodes in the network [1].

The EVM is a stack-based virtual machine, meaning it operates primarily using a stack data structure for executing instructions. It is Turing-complete, allowing it to perform any computation given sufficient resources. The EVM executes bytecode, a low-level, assembly-like language that is the compiled output of high-level smart contract languages such as Solidity[2]. This bytecode is typically represented as as a byte array or hex string, where each byte corresponds to a specific operation in the compiled program. Each operation in the EVM is defined by an opcode, which dictates the specific action to be performed, such as arithmetic operations, data storage, or control flow management[1][3].

Smart contracts deployed on the Ethereum blockchain are stored as bytecode within the state trie[1] entry for the contract's address. When a contract is invoked, whether through a transaction or another contract, the EVM processes the bytecode step by step, ultimately executing the desired functions. Additionally, a caller must provide a set of arguments in order to invoke a function call. These arguments are encoded into another byte array known as the calldata, which is passed to the contract as part of the transaction. The contract can then access the calldata via various opcodes and use it during contract execution[4]. In addition to calldata, smart contracts can access storage (key-value entries in the state trie that persist on-chain indefinitely), transient storage (which is identical to storage but does not persist after execution), and memory (a simple byte array used for more complex operations).

1.2 Security Implications

Given that smart contracts often handle valuable assets and sensitive operations, combined with the fact that EVM bytecode is immutable once deployed, the security of the EVM and the correctness of smart contract code are paramount. Vulnerabilities in smart contracts can lead to significant financial losses, as evidenced by high-profile incidents such as the DAO hack.[5].

Additionally, due to the decentralized nature of the Ethereum network, there is no source-of-truth for verified smart-contract source code. While the bytecode is stored immutably on-chain, it is up to centralized services, such as Etherscan or Codeslaw, to store and verify that user-provided source code correctly compiles to the bytecode stored on-chain. This leads to two problems; not all smart contracts have verified source code, and not everyone wants to blindly trust centralized services.

This inherent "unverified" nature of EVM data has lead to the birth of open-source tools such as `heimdall-rs` [8], `whatsabi` [9], and `shrnkr` [10][11], which aim to make the inner workings of EVM smart contracts more transparent through decompilation, advanced bytecode analysis, and other methods.

1.3 Smart Contract Decompilation

Smart contract decompilation aims to translate low-level EVM bytecode back into a more understandable, high-level representation resembling the original source code. This process is inherently challenging due to several factors:

- The compilation pipeline from high-level languages to EVM bytecode is lossy; it discards variable names, types, comments, library information, and other structural cues. As a result, the recovered code often lacks semantic clarity and rich type information, making it difficult to infer the original developer's intent.
- Complex control flows, especially those influenced by dynamic runtime values, can obscure the logic of loops, conditional branches, and exception handling.

[1] Under most circumstances. Certain self-destruction patterns may allow for contracts to be re-initialized with different bytecode.

- Compiler optimizations can obfuscate the resulting bytecode, making it much harder to reverse engineer and extract useful information from.

Despite these difficulties, significant progress has been made through static analysis, symbolic execution, and pattern recognition techniques. Contemporary decompilers employ heuristics to identify function boundaries, reconstruct conditional logic, and infer data types where possible. Additionally, new approaches leverage large language models to add semantic layers of understanding — such as suggesting meaningful variable names, detecting design patterns, or classifying functions by intent — thereby reducing the gap between a raw decompiled output and a polished, human-readable source code approximation [12].

Ultimately, while perfect one-to-one reconstruction of the original source code remains elusive, the goal of decompilation is to produce an output that is sufficiently comprehensible for auditing, analysis, and documentation. The integration of LLM-based post-processing methods, as discussed in this work, pushes the frontier in delivering cleaner, more accurate, and more maintainable decompilation results.

2 Existing Techniques and Solutions

Decompiling EVM bytecode into high-level source code involves various methodologies, each leveraging distinct approaches to address the inherent challenges of reverse engineering. This section explores the primary categories of decompilation techniques, including static analysis-based decompilers that examine bytecode structure without execution, symbolic execution-based methods that simulate contract behavior to infer logic, and machine learning-driven approaches that utilize data patterns to enhance decompilation accuracy. By analyzing these existing solutions, we highlight their respective advantages and limitations, setting the stage for introducing our LLM-driven post-processing pipeline that aims to bridge the gaps and improve the overall effectiveness of EVM decompilation.

2.1 Static Analysis-Based Decompilers

2.1.1 Shrnkr

`shrnkr` [11] is a state-of-the-art, static-analysis-based EVM decompiler that significantly advances over previous approaches such as Elipmoc. Unlike purely symbolic execution-based tools or older static methods, `shrnkr` introduces several key innovations that improve scalability, completeness, and precision simultaneously. Its contributions can be summarized as follows:

- **Shrinking Context Sensitivity:** Central to `shrnkr` is a novel form of context sensitivity for global control-flow analysis called *shrinking context sensitivity*. Traditional context-sensitive techniques track a sequence of call-like patterns to maintain precision. However, as contract bytecode grows more complex (especially with optimizations and the adoption of the Yul pipeline), naive approaches can lead to explosion in complexity and cause scalability issues. `shrnkr`'s shrinking context sensitivity actively "forgets" control-flow history once a likely function call returns. This selective pruning of context information enables retaining crucial precision where it matters, while allowing the analysis to scale to large contracts without timeouts.
- **Control-Flow Normalization via Cloning:** In many compiled contracts, the same low-level block is reused to implement multiple high-level constructs, including chained internal function calls and common stack-balancing routines. Such reuse can impede the generation of a clean high-level control flow. `shrnkr` tackles this by identifying frequently reused blocks and cloning them, restoring a more direct and comprehensible structure. This cloning step significantly reduces unstructured control flow and makes the decompiled IR more amenable to auditing and further analysis.
- **Incomplete Global Pre-Analysis:** `shrnkr` employs a preliminary global pre-analysis phase, a bounded but precise exploration that identifies spurious call edges, extraneous public-call inferences, and edges that introduce imprecision. By leveraging results from this pre-analysis, `shrnkr` can refine its main analysis and produce cleaner, more accurate decompilation results. The incomplete pre-analysis filters out noise in the inferred control-flow and improves both precision and scalability.

Collectively, these innovations enable `shrnkr` to cover significantly more code paths, reduce spurious complexity in the recovered IR, and maintain performance — even on large, heavily optimized contracts

produced by the Yul/IR compilation pipeline. Experimental comparisons show that `shrnkr` outperforms previous static and symbolic execution-based decompilers on standard benchmarks, achieving close to full coverage, drastically fewer timeouts, and substantially improved structural clarity and type reconstruction capabilities [11].

2.1.2 Whatsabi

`whatsabi` is a lightweight static analysis tool for Ethereum bytecode that focuses on ABI inference rather than full decompilation. It extracts information such as function selectors, event topics, and other ABI elements directly from the EVM bytecode without relying on source code, making it suitable for use on unverified contracts or in environments where only bytecode is available. Unlike traditional decompilers that attempt to recover structured high-level code, `whatsabi` aims to provide a practical, resource-efficient solution for deducing a contract's ABI and proxy configuration.

Key characteristics of `whatsabi` include:

- **Static Analysis for ABI Inference:** `whatsabi` performs a linear-time scan of bytecode to identify function selectors (4-byte method identifiers) and event signatures (32-byte event topics). It leverages known code patterns emitted by standard compilation processes to guess payable and non-payable functions, detect public functions, and identify potential fallback handlers.
- **Proxy Detection and Resolution:** Modern contract architectures frequently use proxy patterns for upgradeability and modularity. `whatsabi` includes built-in heuristics and known storage slot patterns to detect common proxy contracts. It can automatically trace through multiple levels of proxy redirection to find the underlying logic contract and generate a more accurate ABI for the resolved target.
- **Integration with Public Databases:** Beyond basic pattern recognition, `whatsabi` can leverage external signature lookup services (e.g., 4byte.directory, Sourcify, Etherscan, and other third-party sources) to resolve unknown selectors or event topics to human-readable names and parameter lists. This integration allows for more informative results and helps match known standard functions and events.

Unlike full decompilers such as `shrnkr` or `heimdall-rs`, `whatsabi` does not produce a high-level structured code representation. Instead, its strength lies in quick, practical ABI reconstruction and environment-friendly integration. This level of abstraction suffices for many use cases — such as building procedural frontends, enhancing block explorer interfaces, wallet integrations, and preliminary bytecode investigations — without the overhead and complexity of a full decompilation pipeline.

2.2 Symbolic Execution-Based Decompilers

2.2.1 heimdall-rs

`heimdall-rs` [8] is a versatile, open-source decompilation and analysis toolkit for Ethereum smart contracts, designed to produce human-readable approximations of high-level code from raw EVM bytecode. While traditional static analysis or symbolic-execution-based decompilers often focus on a single methodology, `heimdall-rs` leverages a hybrid approach that integrates both symbolic execution and structured post-processing. This synergy enables richer semantic inference, more accurate type recovery, and a final output that closely resembles original Solidity code constructs.

At the core of `heimdall-rs`'s symbolic execution strategy is the notion of recursively "branching" virtual machines — new VMs are spawned at conditional jumps (`JUMPI`) to exhaustively trace all reachable execution paths, capturing the program's state transformations along the way. This produces a control flow grpah (CFG) that reflects the contract's possible behavioral trajectories rather than just its syntactic structures. From this CFG, `heimdall-rs` attempts to recover higher-level constructs by interpreting low-level operations (such as `MSTORE`, `SLOAD`, and `CALLDATALOAD`) as typed variables, structured conditionals, loops, return statements, and even Solidity-like event emissions. The result is a reconstruction process that, while not perfect, yields code that more closely resembles the original Solidity patterns than raw disassembly.

A key aspect of `heimdall-rs` is its effort to reintroduce semantic meaning to otherwise opaque code fragments. By symbolically tracking data dependencies, it can infer argument types, derive variable sizes (e.g., identifying values as `address`, `uint256`, or `bytes32`), and detect well-known function signatures or events. Post-processing steps then leverage both deterministic heuristics and external signature databases to resolve common selectors and map them back to familiar Solidity function names. This type inference and naming scheme bridges the gap between raw EVM opcodes and more meaningful, readable representations.

While the current process is already effective for many practical use cases, its integration with LLM-guided post-processing (as explored in this work) aims to refine the final output further. By layering large language model capabilities atop the generated high-level code, `heimdall-rs` can propose semantically appropriate variable names, detect subtle design patterns, and clarify complex branching logic. This synergistic combination of symbolic execution and LLM enhancements moves beyond straightforward pattern recognition, establishing a pipeline that can yield clearer, more comprehensible decompiled code that closely aligns with the developer's original intent.

2.3 Machine Learning and Data-Driven Methods

2.3.1 LLM4Decompile

`LLM4Decompile` [12] represents a significant leap forward in the use of machine learning for binary decompilation. Departing from traditional approaches that rely heavily on symbolic execution or static analysis, `LLM4Decompile` leverages large language models to transform disassembled binary instructions directly into coherent, high-level source code. By treating decompilation as a translation task, the system aligns low-level assembly-like representations to idiomatic high-level constructs.

Key contributions and features of `LLM4Decompile` include:

- **Direct Binary-to-High-Level Translation:** Rather than refining the output of a traditional decompiler, `LLM4Decompile` engages in an end-to-end approach by training LLMs directly on large set of assembly-source code pairs. This data-driven approach encourages the model to learn compiler patterns, enabling it to produce executable C code that closely resembles the original logic and structure.
- **Refined-Decompile Enhancement:** Beyond the end-to-end approach, `LLM4Decompile` also improves upon the refined-decompile strategy, where the model is fine-tuned to postprocess and correct the pseudo-code output of a classical tool like Ghidra [13]. This two-stage refinement significantly elevates code readability, correctness, and the overall success rate of producing re-executable code.
- **Robustness Across Optimization Levels:** The model's training methodology integrates multiple optimization levels of compiled code. As a result, `LLM4Decompile` maintains strong performance even on heavily optimized binaries, a scenario where static-analysis-based methods often struggle.
- **Security and Practical Considerations:** Acknowledging potential concerns regarding misuse, `LLM4Decompile` demonstrates that standard binary obfuscation techniques, such as control-flow flattening, remain effective in preventing unauthorized decompilation of protected software.

Ultimately, `LLM4Decompile` exemplifies the promise of large-scale, learned models in bridging the gap between low-level binaries and human-readable source code. By integrating massive training data, multiple optimization strategies, and refined post-processing, it illustrates how LLM-driven frameworks can bring precision, clarity, and speed to the complex domain of smart contract and binary decompilation.

3 Enhancing Decompilation via LLM-Driven Post-processing

While the above methods for decompilation are all capable of producing a high-level representation of the bytecode, the challenges presented in 1.3 still arise. No matter how sophisticated, decompilers cannot recover semantic information lost during compilation; such as variable names, comments, etc. Additionally, decompilers typically produce code that is often hard to read or riddled with compiler artifacts.

However, recent advancements in LLM reasoning [14][15] have introduced strategies that enable large language models to consistently produce more logically coherent and semantically aware transformations of code. By applying these improved reasoning capabilities to the post-processing step, it becomes possible to enrich the already decompiled code with context-aware adjustments that were previously unattainable. The pipeline not only fixes syntactic or structural issues but also introduces semantic meaning where none was recoverable from pure bytecode analysis. The result is cleaner, more readable code that closely aligns with the original developer's intent, ultimately streamlining both audit and analysis processes.

Our efforts will focus on `heimdall-rs`[2], where we will contribute to the `decompile` module of the toolkit by integrating these advanced LLM-based post-processing techniques. Specifically, we aim to implement features that leverage large language models to automatically restore meaningful variable names, eliminate redundant compiler artifacts, and enhance overall code readability. Additionally, by incorporating semantic enrichment processes, the `decompile` module will be able to infer and insert contextual information that bridges the gaps left by the compilation process.

3.1 Decoding Internal Calls

A crucial enhancement we implemented lies in resolving internal calls made by the smart contract. EVM bytecode often contains invocations to functions within other smart contracts (internal calls), represented by EVM opcodes such as `STATICCALL` or `CALL`. Without source-level signatures or ABI definitions, these external calls appear as 4-byte selectors and memory pointers which do not give much context as to what they do:

```
STATICCALL(gas, address, argsOffset, argsSize, retOffset, retSize)
```

Currently, heimdall can decompile the above assembly into something more human-readable, but it still lacks meaningful context:

```
address(var_a)
    .Unresolved_0x23b872dd{gas: var_e, value: var_f}(
        abi.encodePacked(memory[..])
    )
```

To provide richer semantic context, we integrated a step that attempts to resolve these internal calls back into meaningful function signatures. By leveraging `heimdall-rs`' built-in `decode` module (which relies on common pattern recognition and known selectors) [4], we convert these unreadable external calls into their structured, easily digestible equivalent[3]:

```
address(var_a).transferFrom(abi.encodePacked(memory[..]))
```

We must then map the types returned from the `decode` module to specific locations in memory to determine where each external calldata variable is located. This is trivial for simple types, such as `address` or `uint256`, but gets more complicated when dynamic types such as `uint256[]` take up multiple words in memory. After locating each argument in the external calldata, we can then construct the final, fully decoded external call:

```
address(var_a)
        .transferFrom(var_b, var_c, var_d)
```

3.2 LLM-Driven Post-processing

The LLM-driven post-processing phase refines the decompiled output, transforming the high-level generated code into a more human-readable, semantically meaningful representation. This involves multiple subtasks:

[2]The current commit hash of heimdall-rs before our constibutions is 90e6622f0724af40ba82c26c3d2a40d585373beb
[3]Note that `gas` and `value` fields are omitted in the decoded call. This is only the case if these values are trivial.

- **Variable Renaming:** LLMs infer meaningful variable names based on usage context. Instead of generic `var_a`, `var_b`, etc., the LLM suggests descriptive identifiers like `sender`, `recipient`, or `tokenAmount`, guided by contextual clues (e.g., decoded function names, usage in decoded external calls, arithmetic operations, or storage references).
- **Type Clarification:** When ambiguous, LLMs suggest likely data types. For instance, a value consistently operated on using arithmetic might be labeled as an unsigned integer, whereas a value passed into an address-specific operation might be confidently typed as an `address`.
- **Code Structuring and Comment Inference:** Beyond pure renaming and typing, LLMs improve the code layout and may introduce helpful inline comments. For example, a storage load followed by a simple equality check might prompt the model to annotate the conditional logic (e.g., `// Check if sender is the contract owner`).
- **Pattern Recognition and Rewriting:** LLMs can detect common Solidity design patterns — such as the `withdraw` functions or ERC20 methods — and rewrite them in a canonical form. This reduces the cognitive load required to understand compiler-generated artifacts or uncommon code arrangements.

3.2.1 Prompt Engineering

To bridge the gap between raw, messy decompiled output and polished, human-readable code, we carefully engineered a prompt that guides the LLM through a structured post-processing pipeline. Prompt engineering involves crafting the exact textual instructions given to the LLM, ensuring that it performs the desired transformations reliably and consistently.

The prompt we designed sets clear expectations and constraints, focusing on tasks such as inferring variable purposes, suggesting meaningful names, and adding relevant comments. It emphasizes minimal extraneous chatter and instructs the model to maintain a strict output format that preserves code blocks while excluding any additional explanations. In addition, it requests the LLM to append a `@notice` tag at the end of the pre-existing docstring, summarizing the function's purpose. This final request ensures that the docstring closely resembles a well-commented, developer-friendly snippet of code, rather than a raw, featureless artifact of the decompilation pipeline.

Below is the final prompt that we implemented into `heimdall-rs 0.8.5`:

```
The following solidity code was generated by a decompiler, and is very messy
and lacking variable names and comments as a result. This solidity code may
not compile either, and may include bits that are not valid solidity code.

Your task is to postprocess this decompiled code, cleaning it up and making
it more readable. Specifically:
- Add comments where you are able to infer what the code does.
- Rename variables where you can infer what the variable is used for.
- Remove redundant checks or lines that do not affect the overall
  functionality of the code.
- Add an `@notice` to the end of the pre-existing docsting with a short
summary of the function. Ensure the notice follows the indentation and
spacing of the rest of the docstring.

Respond in the format ```{{code}}```. Respond only with the code after your
modifications have been applied.

```{decompiled code}```
```

This prompt explicitly instructs the LLM to:

- Preserve code formatting and place the final output inside a code block.
- Avoid extraneous commentary outside the code block, ensuring clean integration into the decompiler's pipeline.
```

- Include annotations that are contextually sound and aligned with typical Solidity coding conventions.

The constraints are deliberately strict to prevent the model from drifting into undesired output formats or introducing inaccuracies, a common challenge when working with LLMs on code refinement tasks. By providing a direct code snippet and requesting a code-only response, the prompt reduces any ambiguity about the final desired output format.

3.2.2 Design Choices

In selecting the model and finalizing the prompt, we considered several key factors. We chose `o1-mini` — a variant of OpenAI's advanced reasoning models — due to its balance of speed, reasoning capabilities, and accessibility. These attributes made `o1-mini` an optimal choice, as it provides:

- **Speed:** Rapid inference times are essential when integrating LLM-driven post-processing into a toolchain that security auditors and developers may use repeatedly. `o1-mini` responds swiftly, minimizing overhead and allowing for real-time code refinement in iterative audit loops.
- **Advanced Reasoning Capabilities:** The complexity of post-processing decompiled code—particularly when inferring variable semantics, usage patterns, and suitable comments—requires strong reasoning abilities. `o1-mini` supports chain-of-thought style reasoning, enabling it to make context-sensitive inferences and propose logically consistent improvements to the code.
- **Accessibility:** A model that can run via accessible APIs ensures broad usability.

In summary, our design choices in model selection and prompt construction focus on practicality, reproducibility, and clarity. By employing `o1-mini` and a carefully crafted prompt, the pipeline integrates advanced LLM reasoning seamlessly, resulting in a more effective and user-friendly decompilation process.

4 Results

The results of our LLM-driven post-processing pipeline are highly promising. For well-known contracts like WETH9 — an extensively tested, standardized ERC20 contract — the output is nearly indistinguishable from the original source code. The enhanced variable naming, comment insertion, and functional annotations align so closely with the canonical WETH9 implementation that the resulting code is immediately recognizable and audit-ready. Simpler contracts, such as those employing built-in functions like `ecrecover`, also benefit significantly from this process, emerging as structured, intelligible code that greatly aids in comprehension and verification.

While many functions are successfully clarified and refactored, our approach still faces challenges with certain classes of complex functions. In particular, functions with unresolved or uncommon signatures, heavily obfuscated logic, or deeply nested internal calls remain difficult to reconstruct into a straightforward, semantically clear form. In these cases, the resulting code may still contain placeholder names, incomplete type annotations, or ambiguous control flow structures. Future enhancements will focus on improving these edge cases, potentially by introducing more robust pattern recognition or improved CFG analysis.

4.1 WETH9

Below is an example transformation demonstrating how the raw decompiled output for a WETH9 function (a widely recognized and audited codebase) is improved into a cleaner, nearly source-equivalent snippet:

```
/// @custom:selector    0xa9059cbb
/// @custom:signature   transfer(address arg0, uint256 arg1) public returns (bool)
/// @param              arg0 ["address", "uint160", "bytes20", "int160"]
/// @param              arg1 ["uint256", "bytes32", "int256"]
function transfer(address arg0, uint256 arg1) public returns (bool) {
    address var_a = address(msg.sender);
    var_b = 0x03;
    require(!storage_map_c[var_a] < arg1);
    require(address(msg.sender) == (address(msg.sender)));
    var_a = address(msg.sender);
    var_b = 0x04;
    var_a = address(msg.sender);
    address var_b = keccak256(var_a);
```

```
    require(storage_map_c[var_a] == 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff);
    var_a = address(msg.sender);
    var_b = 0x03;
    storage_map_c[var_a] = storage_map_c[var_a] - arg1;
    var_a = address(arg0);
    var_b = 0x03;
    storage_map_c[var_a] = storage_map_c[var_a] + arg1;
    uint256 var_c = arg1;
    emit Transfer(address(msg.sender), address(arg0), arg1);
    var_c = 0x01;
    return 0x01;
    var_a = address(msg.sender);
    var_b = 0x04;
    var_a = address(msg.sender);
    var_b = keccak256(var_a);
    require(!storage_map_c[var_a] < arg1);
    var_a = address(msg.sender);
    var_b = 0x04;
    var_a = address(msg.sender);
    var_b = keccak256(var_a);
    storage_map_c[var_a] = storage_map_c[var_a] - arg1;
    var_a = address(msg.sender);
    var_b = 0x03;
    storage_map_c[var_a] = storage_map_c[var_a] - arg1;
    var_a = address(arg0);
    var_b = 0x03;
    storage_map_c[var_a] = storage_map_c[var_a] + arg1;
    var_c = arg1;
    emit Transfer(address(msg.sender), address(arg0), arg1);
    var_c = 0x01;
    return 0x01;
    if (address(msg.sender) == (address(msg.sender))) {
        var_a = address(msg.sender);
        var_b = 0x03;
        storage_map_c[var_a] = storage_map_c[var_a] - arg1;
        var_a = address(arg0);
        var_b = 0x03;
        storage_map_c[var_a] = storage_map_c[var_a] + arg1;
        var_c = arg1;
        emit Transfer(address(msg.sender), address(arg0), arg1);
        var_c = 0x01;
        return 0x01;
    }
}
```

After LLM-guided refinement:

```
/// @custom:selector    0xa9059cbb
/// @custom:signature   transfer(address recipient, uint256 amount) public returns (bool)
/// @param              recipient The address to transfer to
/// @param              amount The amount to be transferred
/// @notice             Transfers `amount` tokens from the caller to `recipient`
function transfer(address recipient, uint256 amount) public returns (bool) {
    address sender = msg.sender;

    // Check if the sender has enough balance
    require(balances[sender] >= amount, "Insufficient balance");

    // Subtract the amount from the sender's balance
    balances[sender] -= amount;

    // Add the amount to the recipient's balance
    balances[recipient] += amount;

    // Emit the Transfer event
    emit Transfer(sender, recipient, amount);

    return true;
}
```

4.2 Ecrecover

Similarly, a simpler function using `ecrecover` is transformed from a cluttered snippet into a more recognizable and readable block:

```
/// @custom:selector    0x19045a25
/// @custom:signature   Unresolved_19045a25(uint256 arg0, uint256 arg1) public payable returns (address)
/// @param              arg0 ["uint256", "bytes32", "int256"]
/// @param              arg1 ["uint256", "bytes32", "int256"]
function Unresolved_19045a25(uint256 arg0, uint256 arg1) public payable returns (address) {
    uint256 var_a = var_a + (0x20 + (((0x1f + (arg1)) / 0x20) * 0x20));
    uint256 var_b = (arg1);
    var_c = msg.data[36:36];
    require(var_a.length == 0x41);
    require(!(bytes1(var_d)) < 0x1b);
    require(bytes1(var_d) == 0x1b);
    require(bytes1(var_d) == 0x1b);
    uint256 var_e = 0;
    var_a = 0x20 + var_a;
    uint256 var_f = uint256(arg0);
    bytes1 var_g = bytes1(var_d);
    uint256 var_h = uint256(var_i);
    uint256 var_j = uint256(var_k);
    address var_l = ecrecover(uint256(arg0), bytes1(var_d), uint256(var_i), uint256(var_k));
    require(var_l);
    var_f = address(var_l);
    return address(var_l);
    var_e = 0;
    return 0;
    if ((var_d + 0x1b) == 0x1b) {
        if ((var_d + 0x1b) == 0x1b) {
            var_e = 0;
            return 0;
            var_e = 0;
            return 0;
```

```
        }
    }
}
```

After LLM-guided refinement:

```
/// @custom:selector    0x19045a25
/// @custom:signature   recoverAddress(bytes32 hash, bytes signature) public payable returns (address)
/// @param              hash ["bytes32"]
/// @param              signature ["bytes"]
/// @notice             Recovers the address associated with the provided signature.
function recoverAddress(bytes32 hash, bytes memory signature) public payable returns (address) {
    // Ensure the signature length is correct (65 bytes)
    require(signature.length == 65, "Invalid signature length");

    // Extract signature parameters
    bytes1 v = signature[64];
    require(uint8(v) == 27 || uint8(v) == 28, "Invalid signature 'v' value");

    bytes32 r;
    bytes32 s;

    // Split the signature into r and s variables
    assembly {
        r := mload(add(signature, 32))
        s := mload(add(signature, 64))
    }

    // Recover the signer's address
    address recoveredAddress = ecrecover(hash, uint8(v), r, s);
    require(recoveredAddress != address(0), "Invalid signature");

    return recoveredAddress;
}
```

5 Future Work

While the LLM-driven post-processing pipeline demonstrates substantial improvements in code clarity, there remain several avenues to explore:

- **Further Pattern Recognition:** Integrating more advanced heuristics or a richer signature database into `heimdall-rs` could help the LLM identify and refine unusual patterns in EVM bytecode.
- **Domain-Specific Fine-Tuning:** Fine-tuning the LLM with domain-specific data, such as common DeFi protocols or frequently used contract libraries, could enable more accurate inference of variable types, function semantics, and naming conventions.
- **Interactive Refinement Loops:** Implementing a feedback loop where developers can guide the refinement by approving or rejecting certain LLM suggestions could yield incremental improvements in code quality. Over time, this feedback can inform better prompting strategies and model fine-tuning.
- **Integration with Security Tooling:** Combining LLM-guided post-processing with automated vulnerability detection tools may further expedite security audits, enabling analysts to quickly zero in on potential flaws in a clearer, more readable code representation.

As LLM capabilities continue to evolve and our toolchain matures, we anticipate that these enhancements will push decompilation results closer to a near-source experience, enabling broader and more efficient analysis of smart contracts at scale.

References

[1] Vitalik Buterin. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. 2014. https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf

[2] Solidity Contributors. *The Solidity Compiler*. 2014. https://github.com/ethereum/solidity

[3] smlXL. *EVM Codes*. 2024. https://evm.codes

[4] Jonathan Becker. *On Decoding Raw Calldata*. 2023. https://jbecker.dev/research/decoding-raw-calldata

[5] Cryptopedia Staff. *What Was The DAO?* 2023. https://www.gemini.com/cryptopedia/the-dao-hack-makerdao

[6] Etherscan. 2024. https://etherscan.io

[7] Codeslaw. 2024. https://codeslaw.app

[8] Jonathan Becker, et al. *Heimdall*. 2022. https://github.com/jon-becker/heimdall-rs

[9] Andrey Petrov, et al. *Whatsabi*. 2022. https://github.com/shazow/whatsabi

[10] Debaub. *Debaub EVM Decompiler* 2024 https://app.dedaub.com/decompile

[11] Sifis Lagouvardos, et al. *The Incredible Shrinking Context... in a decompiler near you* 2024 https://arxiv.org/html/2409.11157v1

[12] Hanzhuo Tan, et al. *LLM4Decompile: Decompiling Binary Code with Large Language Models* 2024 https://arxiv.org/pdf/2403.05286

[13] National Security Agency *Ghidra* 2019 https://github.com/NationalSecurityAgency/ghidra

[14] Lifan Yuan, et al. *Advancing LLM Reasoning Generalists with Preference Trees* 2024 https://arxiv.org/pdf/2404.02078

[15] OpenAI. *Introducing OpenAI o1* 2024 https://openai.com/o1/