Published: 19 June 2025
Accepted: 01 April 2025
Received: 12 September 2024

Citation in BibTeX format

RESEARCH-ARTICLE

Recasting Type Hints from WebAssembly Contracts

KUNSONG ZHAO, The Hong Kong Polytechnic University, Hong Kong, Hong Kong, Hong Kong

ZIHAO LI, The Hong Kong Polytechnic University, Hong Kong, Hong Kong, Hong Kong

WEIMIN CHEN, The Hong Kong Polytechnic University, Hong Kong, Hong Kong, Hong Kong

XIAPU LUO, The Hong Kong Polytechnic University, Hong Kong, Hong Kong, Hong Kong

TING CHEN, University of Electronic Science and Technology of China, Chengdu, Sichuan, China

GUOZHU MENG, Institute of Information Engineering, Beijing, China

View all

Open Access Support provided by:

The Hong Kong Polytechnic University

Institute of Information Engineering

Zhejiang University

University of Electronic Science and Technology of China

Recasting Type Hints from WebAssembly Contracts

KUNSONG ZHAO, The Hong Kong Polytechnic University, China
ZIHAO LI, The Hong Kong Polytechnic University, China
WEIMIN CHEN, The Hong Kong Polytechnic University, China
XIAPU LUO*, The Hong Kong Polytechnic University, China
TING CHEN, University of Electronic Science and Technology of China, China
GUOZHU MENG, Institute of Information Engineering at Chinese Academy of Sciences, China
YAJIN ZHOU, Zhejiang University, China

WebAssembly has become the preferred smart contract format for various blockchain platforms due to its high portability and near-native execution speed. To effectively understand WebAssembly contracts, it is crucial to recover high-level type signatures because of the limited type information that WebAssembly provides. However, existing studies on type inference for smart contracts primarily center around Ethereum Virtual Machine bytecode, which is not applicable to WebAssembly owing to their differing targets and runtime semantics. This paper introduces *WasmHint*, a novel solution that leverages deep learning inference to automatically recover high-level parameter and return types from WebAssembly contracts. More specifically, *WasmHint* constructs a *w*CFG representation to clarify dependencies within WebAssembly code and simulates its execution to capture type-related operational information. By learning comprehensive code semantics, it infers parameter and return types, with a semantic corrector designed to enhance information coordination. We conduct experiments on a newly constructed dataset containing 77,208 WebAssembly contract functions. The results demonstrate that *WasmHint* achieves inference accuracies of 80.0% for parameter types and 95.8% for return types, with average improvements of 86.6% and 34.0% over the baseline methods, respectively.

CCS Concepts: • **Security and privacy** → **Software reverse engineering**.

Additional Key Words and Phrases: Smart Contract, WebAssembly, Type Inference, Deep Learning

ACM Reference Format:
Kunsong Zhao, Zihao Li, Weimin Chen, Xiapu Luo, Ting Chen, Guozhu Meng, and Yajin Zhou. 2025. Recasting Type Hints from WebAssembly Contracts. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE118 (July 2025), 24 pages. https://doi.org/10.1145/3729388

1 Introduction

The blockchain ecosystem is currently exhibiting robust growth and development, particularly following the advent of Ethereum and its smart contracts [64, 65, 80]. However, as decentralized applications continue to evolve, Ethereum is often criticized for its poor scalability, high costs, and low throughput [14]. As the runtime environment of smart contracts, the design of the Ethereum

*Corresponding author.

Authors' Contact Information: Kunsong Zhao, The Hong Kong Polytechnic University, Hong Kong, China, kunsong.zhao@connect.polyu.hk; Zihao Li, The Hong Kong Polytechnic University, Hong Kong, China, cszhli@comp.polyu.edu.hk; Weimin Chen, The Hong Kong Polytechnic University, Hong Kong, China, cswchen@comp.polyu.edu.hk; Xiapu Luo, The Hong Kong Polytechnic University, Hong Kong, China, csxluo@comp.polyu.edu.hk; Ting Chen, University of Electronic Science and Technology of China, Chengdu, China, brokendragon@uestc.edu.cn; Guozhu Meng, Institute of Information Engineering at Chinese Academy of Sciences, Beijing, China, mengguozhu@iie.ac.cn; Yajin Zhou, Zhejiang University, Hangzhou, China, yajin_zhou@zju.edu.cn.

ACM 2994-970X/2025/7-ARTFSE118
https://doi.org/10.1145/3729388

Virtual Machine (EVM) inherently constrains its compatibility and interoperability [80]. To alleviate these issues, a variety of new blockchain platforms have emerged, such as Near [13], Cosmos [51], etc. Moreover, they choose more efficient runtime to support running smart contracts [6].

WebAssembly, also known as Wasm, although originally designed as a complement to JavaScript, becomes the first choice to replace EVM by many blockchain platforms due to its high portability and execution efficiency [6, 13, 44, 57]. Wasm is an efficient compilation target that is compiled from high-level languages such as C/C++ and Rust. The compiled Wasm code can be deployed on blockchains regarding as smart contracts and executed by the Wasm virtual machine [6]. As Wasm becomes more widely adopted on blockchain platforms, developers are increasingly publishing compiled Wasm code as library functions in third-party services, enabling others to more easily develop their own blockchain applications [9, 16]. This, in turn, increases the requirements for developers to understand the Wasm code before using it [48, 55]. Function type signatures play a critical component towards understanding code behaviors and facilitating reverse engineering tools [35, 57, 84]. Consequently, they offer a practical approach to understanding Wasm code, as types are intrinsically linked to low-level Wasm operations [57].

Unfortunately, it is very challenging to recover type signatures from Wasm contracts due to the limited available type information [27]. More specifically, there are merely four low-level primitive types in Wasm including integers and floats with 32 and 64 bits, and all types in high-level languages are converted into one of them. These low-level representations with almost no type information hinder the understanding of Wasm contracts and reverse engineering accordingly. For example, many high-level types, such as pointer, bool and integers can be represented by the same integer with 32 bits in Wasm, which makes it impossible for analyzers to determine its true type. This necessitates the recovery of types used in the high-level languages from which the Wasm contracts are compiled to enhance the understanding of the behaviors of Wasm contracts.

Many existing studies have concentrated on type recovery across a variety of programming languages. These efforts include inferring types from native code compiled from C/C++ [39, 69, 71] as well as predicting variable types in JavaScript [67] and Python [29, 68, 70]. Moreover, recent studies proposed to recovering function type signatures from EVM bytecode by means of symbolic analysis [36] or neural networks [84]. However, due to different targets, instruction operations, and runtime semantics, none of them can be used to recover high-level types from Wasm contracts. Although recent work [48, 57] designed methods for type recovery from Wasm code, they still have the following limitations. First, they defined a type system that was limited to C/C++ and predicted customized types rather than the actual types in source code from Wasm. However, since blockchain systems involve enormous digital assets, developers tend to choose memory-safe programming language, i.e., Rust, for developing smart contracts [13, 19]. This makes such an approach unsuitable for recovering high-level types and will impede the understanding of real-world smart contracts. Second, their work focused on inferring customized types of each parameter in Wasm function declaration [48, 57], which does not align with the actual compilation conversion from high-level languages to Wasm code. As illustrated in § 2.4, multiple parameters in Wasm function will correspond to a single high-level parameter type while the first parameter in the Wasm function declaration could be related to the return value in the original function.

To address the above challenges, the most classical solution is to conduct data flow analysis and summarize various type operation patterns for inferring high-level types [33, 36]. However, due to the linear memory model and dynamically determined control flow instructions in Wasm, it is non-trivial to implement static analysis framework for the low-level Wasm code [44]. In addition, it is unlikely to represent all behaviors associated with type operations through limited manual efforts in advance, i.e., lacking scalability. This motivates us to design a learnable method to automate the understanding of Wasm semantics for recovering high-level types. A recent work SnowWhite

[57] has designed a Wasm to C/C++ type recovery tool with the one-to-one parameter or return correspondence in Wasm code. We improve SnowWhite and addresse new challenge of recovering Rust types from Wasm code, which involves handling non-one-to-one mappings of parameters and return values as illustrated in 2.5.

In this paper, we propose a novel deep learning-based method called *WasmHint* to automatically recover type signatures from Wasm contracts that are compiled from Rust smart contracts. To illustrate the semantic information, we construct the *w*CFG representation for each Wasm function, making the dependencies within the Wasm code explicitly clear. This representation forms the foundation for subsequent feature extraction and deeper semantic understanding (§ 3.2). To extract refined instruction slices related to function parameter operations, we develop an operation-centric code slicing technique that dynamically simulates the execution of instructions in Wasm functions. It can monitor the usage of function parameters in Wasm code to identify relevant instructions. Similarly, we extract operation instructions associated with the return value during the simulation (§ 3.3). To capture operation semantics, we design a semantic encoder that learns implicit features from extracted instruction slices, the constructed *w*CFG, and other additional knowledge. These features are then combined to produce semantic vectors for further type inference (§ 3.4). In the type inference process, we combine classification and generative model paradigms to accurately recover high-level types (§ 3.5). We further design an innovative semantic corrector module to narrow the differences and maximize the similarity between embedded semantics and output types. It ensures that the generated type accurately reflects the semantic understanding derived from type-related operations (§ 3.6).

We conduct comprehensive experiments to evaluate the effectiveness of *WasmHint* for recovering parameter types and return types. Due to the lack of an available dataset, we collect projects related to Rust smart contracts from GitHub and build a new dataset consisting of 77,208 Wasm contract functions for evaluation. The experimental results demonstrate that, overall, *WasmHint* achieves an average accuracy of 80.0% for inferring parameter types and 95.8% for return types. It shows a significant performance improvement, with average gains of 86.6% and 34.0% over the baseline methods, respectively. Further experiments reveal that each component designed in *WasmHint* contributes positively to its effectiveness, and their combination is key to the successful inference of both parameter and return types.

In summary, this paper makes the following major contributions.

- We address the practical problem of inferring high-level types from Wasm contracts. To the best of our knowledge, this is the first work on type recovery focusing on Wasm smart contracts.
- We develop *WasmHint*, a novel method that can extract and learn implicit operational semantics associated with function parameters and return values from Wasm contracts, facilitating the task of inferring high-level types.
- We build a new dataset consisting of 77,208 Wasm contract functions from real-world Rust smart contract projects. To the best of our knowledge, this is the first dataset labeling the low-level Wasm smart contract functions with high-level type information.
- We conduct comprehensive experiments to evaluate the performance of *WasmHint* and the results demonstrate that *WasmHint* is more capable of accurately inferring both parameter and return types than baseline methods, with each component contributing effectively to its success.

2 Background

2.1 Wasm Smart Contracts

Blockchain and smart contracts. Blockchain is a decentralized distributed ledger platform that ensures data immutability in a cryptographic way [80]. Smart contract refers to an executable

program running on blockchains, which is primarily written in high-level languages (such as Solidity [20], C/C++ [8], and Rust [17]) and executed by a virtual machine after compilation without any dependency on trusted third parties [62].
Wasm. Wasm is a binary instruction format executed in a stack-based virtual machine [22]. Due to its portability and high efficiency, many blockchain platforms are adopting it as a runtime environment for executing smart contracts [5, 8]. It allows the same functionality to be executed on various blockchains, providing smart contracts with more flexible cross-platform compatibility [6].
Wasm smart contracts. Wasm smart contracts are written in high-level languages and compiled into low-level Wasm code. Due to the memory safety and powerful macro definition, Rust has become the preferred language for the development of Wasm contracts in many popular blockchains [6, 13, 19]. Therefore, we focus on Wasm smart contracts written in Rust. To the best of our knowledge, no existing work has explored type recovery from Wasm smart contracts.

2.2 LLVM IR

The LLVM intermediate representation (LLVM IR) serves as a powerful and adaptable framework that allows compilers for high-level languages to seamlessly adapt to a variety of targets without necessitating changes to the fundamental logic for each target [10, 52]. A key advantage of LLVM IR is its powerful type system and debug information, showcased through built-in components like *DIType*, *DISubroutineType*, *DICompositeType*, etc [11]. Readers can see more details about these definitions in [10, 11, 52]. This design effectively holds information from the source code, enabling the translation of high-level representations into their low-level equivalents [10]. It provides the opportunity to establish a link between the types of Rust and Wasm.

2.3 Accessing Patterns of Parameters and Returns in Wasm

To create an efficient and platform-independent binary code format, Wasm defines a simple set of types that forms the foundation for building Wasm code [22]. There are four types in Wasm: *i32*, *i64*, *f32*, and *f64*. The first two represent 32-bit and 64-bit integer types, respectively, while the latter two represent 32-bit and 64-bit floating-point types. All these four types can be used for local and global variables, function parameters, and return values [27]. In contrast, as a high-level, memory-safe language, Rust offers a more extensive type system than Wasm [18], which includes six groups of type definitions, such as primitive types, sequence types, pointer types, user-defined (or named) types, function types, and trait types. To facilitate execution, the various types in Rust code are converted to the four low-level types defined in Wasm [27]. Since this process erases rich type details, it prevents an understanding of the functionality of the original code.

To facilitate code understanding, it is essential to grasp Wasm's function access patterns. In Wasm functions, three instructions are used to handle parameter data: *local.get*, *local.set*, and *local.tee* [25]. *local.get* loads a variable and pushes it onto the stack. *local.set* saves the value from the stack into a local variable based on its index (i.e., the operand). *local.tee* operates similarly to *local.set*, but in addition to storing the value in the local variable, it also leaves the value on the stack. Once the function completes execution, Wasm uses the *return* instruction to deliver the result.

2.4 Type Conversion from Rust to Wasm

This subsection analyzes the conversion of function parameters and return values from Rust code to their formats in Wasm code. Note that we assume the conversion follows the compilation behavior of the *rustc* compiler, which relies on LLVM IR as its backend for code generation. To enhance readability, we use a symbol ˆ to represent Wasm types, distinguishing them from Rust types.
Primitive types. The primitive types can be easily matched with the Wasm types in the compilation process. For integers with M bits *i<M>* ($M \in [8, 16, 24, 32]$), since its length does not exceed 32 bits,

```rust
pub fn is_roi_achieved(invs: i128) -> bool {
    let mut final_amount = invs;
    // YEARS: investment cycle
    for _ in 0..YEARS {
      // ROI_RATE: annualized rate of return
      let roi = final_amount * ROI_RATE / 1000;
      final_amount += roi;
    }
    let total_roi_rate = (final_amount - invs)
     * 100 / invs;
    // TGT_RET_RATE: target rate of return
    total_roi_rate >= TGT_RET_RATE
}
```

(a) Example 1: Return of Investment

```wasm
(func $is_roi_achieved (param i64 i64) (
     result i32)
  ...
  local.get 0 ;; Load low word of i128
  i64.store offset=104 ;; Store to memory
  ...
  local.get 1 ;; Load high word of i128
  i64.store offset=112 ;; Store to memory
  i32.const 0 ;; Push constant onto stack
  ...
  loop  ;; Start a loop
  ...
return) ;; Return from the function
```

(b) Compiled Wasm for Example 1

```rust
pub fn vote(cid: u64) -> VoteResult {
    // RECORDS: holding all vote results
    if cid < RECORDS.len() as u64 {
      RECORDS[cid as usize] += 1;
      VoteResult {
        cid, // Candidate id
        // total_votes: total number of votes
        total_votes: RECORDS[cid as usize],
      }
    } else {...} // Handling other situations
}
```

(c) Example 2: Voting

```wasm
(func $vote (param i32 i64)
  ...
  local.get 1 ;; Load the cid
  local.set 6 ;; Store into local variable
  ...
  i64.lt_u ;; Unsigned comparison
  ...
  local.get 0 ;; Load addr of VoteResult
  i32.const 16 ;; Push constant onto stack
  ...
return) ;; Return from the function
```

(d) Compiled Wasm for Example 2

Fig. 1. Examples of two Rust contract functions and the compiled Wasm counterparts.

it can be translated to low-level Wasm type $\hat{i}32$ accordingly. For an integer with 64 bits, as its length just matches that of the lower Wasm type $\hat{i}64$, it can be represented accordingly. As for *i128*, since it is 128 bits long and the lower type has a maximum length of 64 bits, it is broken down into two low-level $\hat{i}64$ types. Accordingly, since there is no distinction between signed and unsigned types in Wasm, signed and unsigned integer types adopt the same conversion specification. More directly, 32- and 64-bit floating point numbers (including *f32* and *f64*) in Rust can naturally be matched with low-level Wasm types $\hat{f}32$ and $\hat{f}64$ respectively, due to their consistent range. A boolean value can be true or false, which can be represented by just one bit. When compiling a *bool* type into Wasm code, the low-level $\hat{i}32$ type is enough to accommodate it. Similarly, a value with *char* type is a Unicode scalar value and can be expressed using an unsigned word with 32-bit length [18]. Thus, it can be represented by $\hat{i}32$ in Wasm accordingly.

Sequence types. A tuple belongs to a structural type that comprises at least one field. It can be translated according to the elements it contains. For example, a tuple containing two 32-bit integers in a function parameter can be represented by two Wasm types $\hat{i}32$. Similarly, an array is a fixed-size sequence consisting of multiple elements with the same type. It can also be converted into Wasm types according to the type of its element. A *slice* represents a dynamic subset of elements of the same type. When parsing a parameter with the type *slice*, it is converted into two $\hat{i}32$ types in Wasm, which refers to the location where elements are stored and the length that needs to be captured.

User-defined (named) types. A *struct* can consist of multiple fields with various types. If a *struct* contains multiple elements with the primitive type *i32* in Rust, it will be represented by

merely a Wasm type $\hat{i32}$ pointing to the address of this struct [26]. Similarly, *enum* contains a set of named constants meanwhile *union* allows a variable to have various types. These two types are also represented as the corresponding addresses in the form of a Wasm type $\hat{i32}$. These types encapsulate associated data to provide a uniform data structure and consistent access patterns [18].
Trait types. A *trait* is a special type in Rust that allows developers to define features or interfaces for behaviors shared between different types [18]. For example, when adding a *trait* to a given struct, the corresponding implementation will translate the function parameter into a $\hat{i32}$ in the compiled Wasm code which represents the self pointer points to the struct itself [26].
Other types. For other types, in the process of compiling Rust code to Wasm, the involved parameters will be referred to by the corresponding pointers, so the type $\hat{i32}$ is used in Wasm.
Returns. In the previous, we focus on the type conversion of the function parameters when compiling Rust to Wasm. In contrast, when returning primitive types in Rust function, the return types are directly mapped to corresponding Wasm types as specified in the *result* part of the Wasm function declaration. However, there are other types that are translated to the first parameter declaration of Wasm function rather than the return declaration [26].

2.5 Motivating Example

Fig. 1 illustrates two examples of smart contract functions written in Rust and their respective Wasm equivalents. As shown in *Example 1* (Fig. 1 (a)), this function determines whether the return on investment has reached the predetermined target (i.e., *TGT_RET_RATE*) by calculating the cumulative return at the end of the investment cycle (i.e., *YEARS*) according to its annualized rate of return (i.e., *ROI_RATE*). It takes as input an *i128* type parameter representing the initial investment amount and returns a boolean value indicating whether the target rate of return has been achieved. When this Rust function is compiled into its Wasm counterpart, as shown in Line 1 of Fig. 1 (b), the parameters in the function declaration are decomposed into two $\hat{i64}$ data types supported by Wasm. Similarly, the return value is converted to a Wasm type, i.e., $\hat{i32}$, and is declared in the *result* part. In the Wasm code, the function can access the corresponding high-level parameters using the instructions *local.get 0* (Line 3) and *local.get 1* (Line 6) to retrieve the values of the first and second parameters, respectively. The two individual parameters can then be combined to reconstruct the original data (i.e., *i128*) using in the Rust code. Obviously, this conversion results in a mismatch between the function declarations in Wasm and the original Rust code. This mismatch brings a gap to understand the code semantics because the high-level type information is not directly preserved.

To recover the high-level parameter type of this function, we need to consider not only the operation instructions associated with the corresponding parameters, but also the appropriate method to combine this information. Unlike the parameter types, the return value is stored in memory rather than being involved in an explicit operations [22]. Fortunately, Wasm uses the *return* instruction to indicate the return from a function (Line 12 in Fig. 1 (b)). Therefore, we need to analyze the sequence of operations that ends up with the *return* instruction in the Wasm code. However, in some cases, the declarations of return values in Wasm are not always explicitly placed in the *result* part. *Example 2* (Fig. 1 (c)) illustrates a voting-related function that updates global voting status and returns the current vote record (i.e.,*VoteResult*) by taking a candidate identifier (i.e., *cid*) as input. The *VoteResult* is a user-defined type that holds the candidate identifier and its total number of votes. Different from the conversion in *Example 1*, as shown in Fig. 1 (d), this Wasm code does not include a *result* part in its function declaration that refers to the return value. Instead, it is transformed into the parameter declaration in the first position and is represented by the $\hat{i32}$ Wasm type. This suggests that inferring the function return types also requires understanding the usage of the first parameter in the Wasm code, rather than relying solely on the instructions associated with the *return* statement.

To address these issues, we can make the model to satisfy: For recovering parameter types, the model needs to extract the contextual operations related to parameter access and leverages the representation learning ability of neural networks to understand the semantic information hidden in these operations. More importantly, due to inconsistencies in parameter correspondence between Rust and Wasm code, the model needs to combine operations related to multiple parameter access patterns and dynamically determines which ones are more important for parameter type recovery. Meanwhile, for the inference of the function return type, the model needs to not only understand the operations related to the return value, but also comprehend and combine the operations associated with the first parameter. This is because the return declaration might be mapped to the first parameter in a Wasm function, and it is not clear whether this behavior occurs without any prior knowledge. As a result, *WasmHint* can infer the original parameter type *i128* and return type *bool* in *Example 1*, as well as the parameter type *u64* and the user-defined return type in *Example 2* by leveraging the semantics of the corresponding Wasm code, respectively.

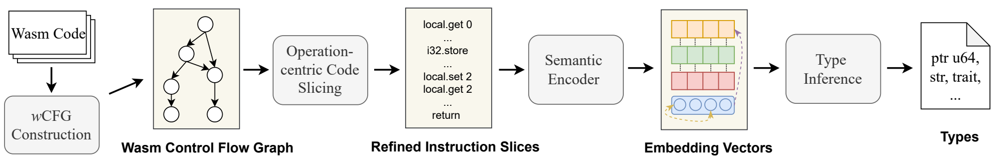


Fig. 2. The overall architecture of *WasmHint*.

3 Design

As mentioned before, in compiled Wasm contracts, all function parameters and returns are represented as one of the four low-level Wasm types, which lacks high-level semantic information and hinders understanding of the original functionality. This section describes how *WasmHint* recovers high-level types from Wasm functions using deep learning techniques. Formally, let Ψ_w be a Wasm contract function that is compiled from its Rust counterpart Ψ_r. Our work aims to recover the high-level Rust types $\mathcal{T}$ by learning the operational semantics associated with the low-level Wasm types $\mathcal{W}$. We encompass the recovery of both parameter types and return types.

3.1 Overview

Fig. 2 illustrates the overall architecture of *WasmHint*, which takes Wasm contract code as input and subsequently generates the high-level parameter and return types in the original Rust smart contracts. *WasmHint* consists of four components including *wCFG Construction*, *Operation-centric Code Slicing*, *Semantic Encoder*, and *Type Inference*. More specifically, *wCFG Construction* develops a Wasm Control Flow Graph (*w*CFG) to accurately represent the control information within a Wasm function. Based on the designed *w*CFG, *Operation-centric Code Slicing* simulates the execution of Wasm code to extract refined instruction slices related to parameters and return values. *Semantic Encoder* constructs encoder networks to capture refined instruction information, which are then integrated to deliver comprehensive semantic representations for subsequent type inference. *Type Inference* infers the possible Rust-level types for parameters and return values. Next, we introduce the design details of each component in *WasmHint* separately.

3.2 *w*CFG Construction

To illustrate the semantic information within a Wasm function Ψ_w, we initiate the process by parsing the instructions to delineate its control flow and subsequently construct the *w*CFG at the basic block level. Specifically, we first disassemble the function Ψ_w to produce the more readable WebAssembly Text Format (WAT) using the WebAssembly binary toolkit [23]. This is because WAT is friendly to static analysis due to its structure representation and semantic reservation [23]. Then, we build wCFG based on the WAT as follows. When encountering control-related instructions, such as *block*, *loop*, *if*, *br*, *br_if*, etc, a new basic block is initialized. This block persists until it encounters another control-related instruction or an *end* or *return* instruction which marks the termination of the current block. These basic blocks are interconnected according to the flows specified by these control instructions. Consequently, this allows for constructing a control flow graph that precisely mirrors the dependencies within the Wasm code, i.e., wCFG. It forms the foundation for subsequent operation-centric code slicing, which is essential for extracting instructions associated with parameters and return values. In essence, the wCFG closely resembles the traditional CFG. We designate it as wCFG to highlight its incorporation of Wasm-specific partition instructions.

3.3 Operation-centric Code Slicing

As previously noted, Wasm code defines its own low-level parameters and returns within its function declarations. While this information does not directly correspond to Rust-level types, it provides valuable insights into how these elements are utilized within the code. To accomplish this, we develop an engine to dynamically simulate the execution of instructions for Wasm functions, which facilitates extracting refined instruction sequences for parameters and return values.

Algorithm 1 shows the process of extracting instruction slices related to parameter operations from Wasm functions. It leverages the wCFG as it delineates the potential execution paths for the instructions (Line 2). Meanwhile, the algorithm identifies the parameters within the Wasm function (Line 3), which are central to the slicing process. It aims at extracting instructions that utilize or are influenced by these parameters during the simulation. We initialize two global variables $\mathcal{ST}$ and $\mathcal{M}$ to record stack and memory states during execution, respectively (Line 4). The simulation initiates at the entry point of wCFG and execute each instruction accordingly (Lines 6-7). We limit the maximum depth of exploration to prevent infinite loops (Lines 10-11). In the simulation, every instruction is executed according to its defined semantics [25]. After execution, updates to the current stack st and memory m are recorded, along with the corresponding adjustments to the global stack $\mathcal{ST}$ and memory states $\mathcal{M}$ (Lines 12-14). For each instruction, the algorithm assesses whether it utilizes a parameter p or is influenced by the $\mathcal{ST}$ or $\mathcal{M}$ associated with p. Should p be implicated, the instruction is recognized as pertinent and incorporated into the instruction slices (Lines 15-19). When meeting branch instructions, the algorithm applies a depth-first approach to recursively navigate to the new target, ensuring a thorough extraction of instructions (Lines 20-22).

When dealing with the stack and memory associated with parameters, information may depend on previous operations. For example, an instruction may read a parameter and store the result on the stack or in memory. This stored information will be used in subsequent operations, requiring us to model and process the movement of this data properly. Therefore, we define the operational semantics of the parameter propagation shown in Fig. 3. We include only the most critical operations for simplicity. S, μ, and Δ represent the stack, memory, and local variables, respectively. Correspondingly, τ_S, τ_μ, and τ_Δ denote their propagation records associated with parameters. Take the *MLOAD* operation as an example. When executing a memory access operation, such as *i32.load*, it retrieves the data v from an address i and pushes it onto the stack S. If either i or v is impacted by a parameter-related value, the final result stored in the stack record (τ_S) will be impacted as well.

Based on the propagation rules, we can gather instruction slices related to parameter operations after the simulation. Meanwhile, for the simulated instruction sequences, we retain those that end with a *return* statement and treat them as slices associated with the return value. During the simulation, we also record the changes in the stack and use them as auxiliary knowledge. This

Algorithm 1: Parameter Operation-centric Code Slicing

```
Input: Wasm function code $C$
Output: Instruction slices $\mathcal{D}$ relevant to wasm parameters
Function ExtractSlices($C$):
    $G \leftarrow$ GenerateCFG($C$);
    $\mathcal{P} \leftarrow$ IdentifyWasmParams($C$);
    $\mathcal{ST}, \mathcal{M} \leftarrow$ Init();
    $\mathcal{D} \leftarrow \{\}$;
    $n \leftarrow$ GetEntryNode(G);
    ExplorePath($n, \mathcal{ST}, \mathcal{M}, \mathcal{P}, \mathcal{D}, 0$);
    return $\mathcal{D}$;
Function ExplorePath($n, st, m, p, D, depth$):
    if $depth >$ MAX_DEPTH then
        return; // Terminate if maximum depth is exceeded
    foreach instruction $i$ in $n$ do
        $st, m \leftarrow$ Execute($i, \mathcal{ST}, \mathcal{M}$);
        $\mathcal{ST} \leftarrow \mathcal{ST} \cup \{st\}$; $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$;
        foreach $p \in \mathcal{P}$ do
            if $p \notin \mathcal{D}$ then
                $\mathcal{D}[p] \leftarrow \emptyset$;
            if IsUsed($p$) $\vee$ IsInfluenced($\mathcal{ST}, p$) $\vee$ IsInfluenced($\mathcal{M}, p$) then
                $\mathcal{D}[p] \leftarrow \mathcal{D}[p] \cup \{i\}$;
        if IsBranchInstr($i$) then
            $tn \leftarrow i$;
            ExplorePath($tn, \mathcal{ST}, \mathcal{M}, \mathcal{P}, \mathcal{D}, depth + 1$);
```

information provides the model with more runtime semantics. According to our statistical analysis, an average of 3.9 code slices can be extracted from a function. These extracted code slices account for 55.6% of the original size of the function on average. We further investigate the impact of the operation-centric code slicing method on the performance of type inference in § 4.6.

3.4 Semantic Encoder

After obtaining instruction slices associated with the operations of function parameters and return values, this subsection details the process of transforming this information into numeric vectors that are suitable for model inputs and constructing learnable models to uncover the implicit semantics embedded within them, as illustrated in Fig. 4.

3.4.1 Token Embedding. For each instruction sequence corresponding to parameter or return value, we extract the opcode sequences to represent it. To characterize each opcode, we draw inspiration from previous work [61] and utilize its textual description along with the number of items placed on and removed from the stack as its features according to the definition in the Wasm opcode documentation [25]. This is because the textual description offers a wealth of semantic information that aids in comprehending the operations performed by the opcode, while the details of stack changes illustrate how an opcode interacts with the stack, revealing the state changes of the runtime code [61]. For the textual description, we convert it into 100-dimensional features based on [72]. For the stack changes, we directly utilize the number of items modified as features. As a result, we can use a 102-dimensional features to represent each opcode.

In addition to the instruction sequences, *w*CFG itself can be used as a complementary feature because it contains the overall semantics of the Wasm function. Since our *w*CFG represents each

$$\frac{v \text{ is value from param index } idx}{\tau_S, \tau_\mu, \tau_\Delta, S, \mu, \Delta \vdash \text{input}(idx) \Downarrow \langle v, P_{in}(idx) \rangle}\ \textsc{Pinput}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, p \rangle \quad S' \hookleftarrow S.push(v) \quad \tau'_S \hookleftarrow \tau_S.push(p)}{\tau_S, \tau_\mu, \tau_\Delta, S, \mu, \Delta, pc \rightsquigarrow \tau'_S, \tau_\mu, \tau_\Delta, S', \mu, \Delta, pc+1}\ \textsc{Lget}$$

$$\frac{\begin{array}{c}\tau_S, \tau_\mu, \tau_\Delta, \mu, \Delta, S \vdash e \Downarrow \langle v, p \rangle \qquad v = S.pop() \qquad t = \tau_S.pop() \\ \Delta' \hookleftarrow \Delta[\circ \leftarrow v] \qquad \tau'_\Delta \hookleftarrow \tau_\Delta[\bullet \leftarrow t]\end{array}}{\tau_S, \tau_\mu, \tau_\Delta, S, \mu, \Delta, pc \rightsquigarrow \tau_S, \tau_\mu, \tau'_\Delta, S, \mu, \Delta', pc+1}\ \textsc{Lset}$$

$$\frac{\begin{array}{c}\tau_S, \tau_\mu, \tau_\Delta, S, \Delta, \mu \vdash e \Downarrow \langle v, p \rangle \qquad i = S.pop() \qquad p_i = \tau_S.pop() \qquad v = \mu[i] \\ p_v = \tau_\mu[i] \qquad S' \hookleftarrow S.push(v) \quad \tau'_S \hookleftarrow \tau_S.push(P_M(p_i, p_v))\end{array}}{\tau_S, \tau_\mu, \tau_\Delta, S, \mu, \Delta, pc \rightsquigarrow \tau'_S, \tau_\mu, \tau_\Delta, S', \mu, \Delta, pc+1}\ \textsc{Mload}$$

$$\frac{\begin{array}{c}\tau_S, \tau_\mu, \tau_\Delta, \mu, \Delta, S \vdash e_1 \Downarrow \langle v_1, p_1 \rangle \quad \tau_S, \tau_\mu, \tau_\Delta, \mu, \Delta, S \vdash e_2 \Downarrow \langle v_2, p_2 \rangle \quad v_2 = S.pop() \\ p_2 = \tau_S.pop() \qquad v_1 = S.pop() \qquad p_1 = \tau_S.pop() \\ \mu' \hookleftarrow \mu[v_1 \leftarrow v_2] \qquad \tau'_\mu \hookleftarrow \tau_\mu[p_1 \leftarrow P_M(p_1, p_2)]\end{array}}{\tau_S, \tau_\mu, \tau_\Delta, S, \mu, \Delta, pc \rightsquigarrow \tau_S, \tau'_\mu, \tau_\Delta, S, \mu', \Delta, pc+1}\ \textsc{Mstore}$$

$$\frac{\begin{array}{c}\tau_S, \tau_\mu, \tau_\Delta, \mu, \Delta, S \vdash \hat{e} \Downarrow \langle \hat{v}, \hat{p} \rangle \qquad \hat{v} = S.pop() \qquad \hat{p} = \tau_S.pop() \\ v = INST(\hat{v}) \qquad S' \hookleftarrow S.push(v) \quad \tau'_S \hookleftarrow \tau_S.push(P_{INST}(\hat{p}))\end{array}}{\tau_S, \tau_\mu, \tau_\Delta, S, \mu, \Delta, pc \rightsquigarrow \tau'_S, \tau_\mu, \tau_\Delta, S', \mu, \Delta, pc+1}\ \textsc{Stack}$$

Fig. 3. Operational semantics of the parameter propagation.

basic block as a node, the features of each node should be designed to capture the information pertaining to the instructions within that specific basic block. To achieve this purpose, we characterize each basic block according to the usages of instructions as follows.

- 10 features related to the frequency of low-level type operations: *logical_i32*, *logical_i64*, *logical_f32*, *logical_f64*, *arithmetic_i32*, *arithmetic_i64*, *bitwise_i32*, *bitwise_i64*, *arithmetic_f32*, and *arithmetic_f64*;
- 5 features related to the frequency of variables, constants, memory, and function calls: *local_variable*, *global_variable*, *memory*, *constant*, and *has_function_call*;
- 4 features related to the frequency of other instructions: *number_of_instrs*, *parametric*, *conversion*, and *others*;
- 7 features related to the conditional instructions: *is_end_with_block*, *is_end_with_loop*, *is_end_with_if*, *is_end_with_br*, *is_end_with_br_if*, *is_end_with_return*, and *is_end_with_unreachable*.

We further transform the instruction sequences within each basic block into 100-dimensional vectors to capture their semantics based on [72]. Consequently, we can represent each basic block by a 126-dimensional node feature.

Beyond opcode sequences and *w*CFG features, we also retain the constants corresponding to the operands of the constant definition instructions. This practice is crucial because, during the conversion from high-level types to Wasm types, constant values provide essential information for type differentiation. For instance, both *u8* and *u16* types are converted to the same $\hat{i32}$ type in Wasm. To represent the numeric ranges for *u8* and *u16*, the Wasm code utilizes distinct instructions: *i32.const 255* and *i32.const 65535*. These instructions place the respective constant values *255* and *65535* on the stack, thereby preserving the unique characteristics of each type. To incorporate this constant information, we gather constant definitions from the Wasm function and construct a

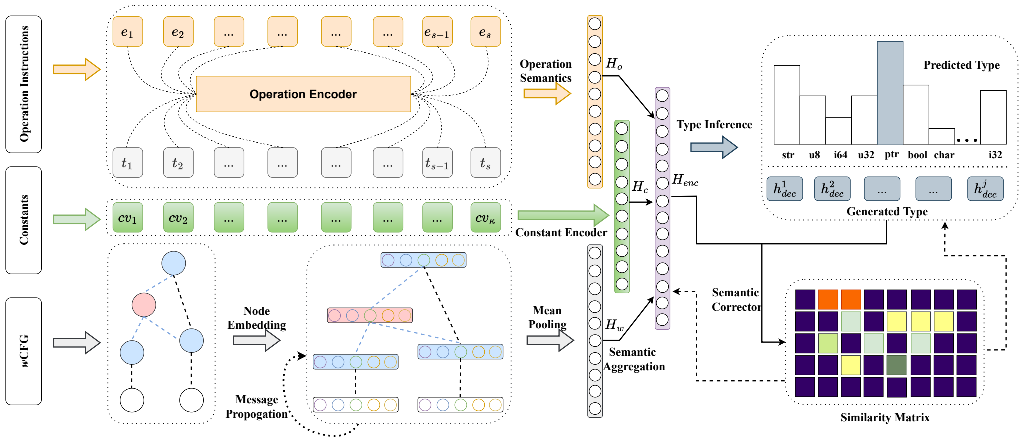


Fig. 4. The semantic learning process of *WasmHint*.

lookup table. This enables the model to directly access the initial embedding vectors of the constants during the training process.

3.4.2 Semantic Learning. To understand the underlying knowledge from the information collected above, we proceed to develop the semantic encoder networks to learn the hidden semantics.
Operation Encoder. For a given opcode sequences $SEQ = \{t_1, t_2, \dots, t_s\}$ (s is the sequence length) associated with a parameter or return value, we embed each token t_s into the initialized vectors $e_s = \Upsilon t_s$ where Υ is a mapping function that converts a token into a 102-dimensional vector as defined in § 3.4.1. To understand the operation semantics behind this sequence, we leverage the encoder component of the Transformer architecture [78] due to its satisfactory performance in many sequence modeling tasks [35, 40, 58, 73]. Specifically, it transforms the initial vectors into three distinct matrices: Q, K, and V, which represent sets of queries, keys, and values, respectively. According to this, the updated outputs can be calculated as follows.

$$H_{attn}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d}})\ V \tag{1}$$

where d represents the dimensions of keys. Recognizing that not all tokens within the opcode sequence contribute equally to the understanding of type-related operations, this mechanism allows the model to selectively evaluate the importance of each token in relation to its corresponding high-level type [78, 84]. Furthermore, residual connections and position-wise feed-forward layers are introduced [46, 78], which is formulated as follows.

$$H = \text{ReLU}((H_{prev} + H_{attn})\theta_1 + b_1)\theta_2 + b_2 \tag{2}$$

where ReLU serves as a nonlinear activation function; H_{prev} denotes the representation immediately prior to the application of the attention mechanism; θ_1, θ_2, b_1, and b_2 are trainable parameters. Additionally, the layer normalization strategy [30] is employed to improve training stability and accelerate model convergence [30].

By stacking multiple such layers, the model can capture various dependencies within the extracted sequence, thereby enhancing its understanding of the type-related semantics associated with parameter or return operations. Additionally, we treat stack records as natural language and convert them into embedding vectors based on [72]. These stack vectors are sequentially added to the respective instruction vectors for providing supplementary semantics.

Constant Encoder. Assume the set of constants collected from a Wasm function is $CV = \{cv_1, cv_2, \dots cv_\kappa\}$ where κ is the number of unique constants. We embed each constant into the initial vector by querying the lookup table ω and produce $ce_\kappa = \omega cv_\kappa$. Since it is not possible to predict in advance which constants will be useful, we incorporate an attention mechanism [81] to dynamically assess the significance of each constant. This mechanism is defined as follows.

$$H_c = \sum_{i=1}^{\kappa} \frac{\exp(f_{score}(ce_i))}{\sum_{j=1}^{\kappa} \exp(f_{score}(ce_j))} ce_i \tag{3}$$

where f_{score} is the scoring function which is implemented by feed-forward neural networks. We introduce the *Constant Encoder* as an independent component since the *Operation Encoder* is tailored to capture contextual relationships between instructions and does not inherently address the distinct nature of constants. In Wasm code, constants are explicitly defined through dedicated instructions [25]. Therefore, we integrate an attention mechanism, enabling it to dynamically identify and prioritize relevant constants for type inference.

***w*CFG Encoder**. To extract the implicit function semantics from *w*CFG, inspired by the previous work [76], we employ the graph aggregation and update technique because it can facilitate the propagation of information across different basic blocks throughout the graph during model training [45, 76]. Given a *w*CFG $G = (V, E)$ where $V = \{bb_1, bb_2, \dots, bb_\iota\}$ refers to the set of nodes and E denotes the set of edges. Here, ι indicates the number of basic blocks. We initialize each node in the *w*CFG according to the designed characteristics as shown in § 3.4.1. For each node, a set of neighbor nodes are sampled (marked as $\mathcal{N}_V$). Then, we aggregate the neighbor information as follows.

$$h^l_{\mathcal{N}_V} = \frac{1}{|\mathcal{N}_V|} \sum_{bb' \in \mathcal{N}_V} h^{l-1}_{bb'} \tag{4}$$

where $h^l_{\mathcal{N}_V}$ refers to the feature vector aggregating the neighbor information at the l-th layer. Then, we turn to update the node embedding by combining the vectors from its previous layer with the aggregated neighborhood feature, which is formulated as follows.

$$h^l_{bb} = \mathrm{ReLU}(\mathcal{W}^l [h^{l-1}_{bb} | h^l_{\mathcal{N}_V}] + b^l) \tag{5}$$

where h^l_{bb} denotes the updated node feature; $\mathcal{W}^l$ and b^l represent the trainable parameters.

After updating the node embedding vectors through multiple layers, we employ mean pooling to aggregate all node vectors and generate the overall embedding vector for a *w*CFG, denoted as H_w.

Semantic Aggregator. As illustrated in § 2.4, we do not know in advance which Wasm parameters will correspond to the original ones. Following the type conversion, we combine three consecutive parameters and their corresponding operations as candidate semantics. Furthermore, for each Wasm parameter, the corresponding low-level Wasm type is placed at the beginning of the operation sequence, separated by the token *<sep>*, to integrate these Wasm types into the overall semantics. In contrast to handling parameter conversion, the return value might be represented either within the *result* declaration or assigned to the first Wasm parameter. To accommodate these variations, we compile two sequences: one that details operations involving the first Wasm parameter, and another that captures the operations associated with the return value. We then average the features from these sequences and utilize them as the overall operation features, denoted as H_o.

Following the aforementioned steps, we succeed in deriving the operation semantics, function semantics, and constant features. The next step involves consolidating these features to create a comprehensive feature representation. To dynamically determine the impact of these features on type inference, we introduce an attention mechanism, which is defined as follows.

$$H_{enc} = \sum (\mathrm{softmax}(\mathcal{W}[H_o, H_c, H_w] + b) \odot [H_o, H_c, H_w]) \tag{6}$$

$$
\begin{aligned}
type &:= base\ type \mid named\ type \mid pointer\ type \mid composite\ type \\
base\ type &:= u8 \mid u16 \mid u32 \mid u64 \mid u128 \mid i8 \mid i16 \mid i32 \mid i64 \mid i128 \mid bool \mid char \mid \\
&\quad str \mid union \mid enum \mid struct \mid trait \mid array \mid slice \mid tuple \mid func \\
pointer\ type &:= ptr \mid ptr\ base\ type \mid ptr\ named\ type
\end{aligned}
$$

Fig. 5. Type set that *WasmHint* can infer

where $\sum(\cdot \odot \cdot)$ denotes the element-wise multiplication of the normalized attention weights with the feature vectors, followed by their summation; W and b are trainable parameters. Building on the fused features, we next explain how to perform the type inference.

3.5 Type Inference

After obtaining the embedding vector H_{enc} for a particular parameter or return value, we employ a two-stage type inference strategy, which first predicts the basic type and then infers the composite type. Given the type set $\mathcal{T}$ defined in Fig. 5, our type prediction stage uses a multi-layer perceptron network to map H_{enc} into a vector $H_{cls} \in \mathbb{R}^{|\mathcal{T}|}$. We apply the cross-entropy loss function [15] as the principal optimization objective to update the learnable parameters during the training phase.

$$\mathcal{L}_{ty} = -\sum_i \hat{y}_i \log(y_i) + (1 - \hat{y}_i)\log(1 - y_i) \tag{7}$$

where $\hat{y}_i$ and y_i are the predicted type and ground truth type, respectively. During the prediction phase, we select the type with the highest probability from H_{cls} as the predicted type.

In contrast, inferring the composite type presents a challenge, as it is impractical to predefine a finite list that encompasses all possible composite types. Inspired by previous studies [57, 84], we adopt a generation model to automatically generate potential type combinations from the type set defined in Fig. 5. Note that this type set is recursive and can represent all possible composite types by recursively combining other types. More specifically, for the embedding vector H_{enc}, our generation model combines it with the input token to generate the target type token as follows.

$$h^j_{dec} = \text{softmax}_{\mathcal{T}}(Decoder(H_{enc}, h^{j-1}_{dec})) \tag{8}$$

where h^j_{dec} denotes the output of j-th decoding step; *Decoder* is implemented as a one-step decoding function using the recurrent neural network. This approach enables every step within the decoding network to thoroughly attend to and capture the related operation semantics [78].

The type inference initiates with a unique starting token *<sos>* and concludes with another token *<eos>*. To enhance model convergence speed and improve stability, we introduce a scheduled sampling strategy that enables the model to dynamically decide whether subsequent inputs should be derived from its previous outputs or from the actual type tokens. This strategy can minimize discrepancies between the generated tokens and their corresponding real types [31, 82]. Similarly, during the training phase, we apply the cross-entropy loss to refine the model parameters and, during the inference phase, select the token with the highest probability as the output at each step.

3.6 Semantic Corrector

The role of *Semantic Encoder* is to understand the type-related semantics associated with operations of parameters or return values, while the *Type Inference* is tasked with producing the true target types based on the insights gleaned from the *Semantic Encoder*. Due to the integration of semantic information from multiple parameter or return operation sequences via a semantic aggregation process in *Semantic Encoder*, a divergence may emerge between the initial and the combined semantic representations. Given the distinct learning objectives of the two components, discrepancies will arise between their outputs. To bridge this divide, we design an innovative module called *Semantic Corrector* to narrow the differences between embedded semantics and output types. Specifically, considering the outputs from the *Semantic Encoder* (i.e., H_{enc}) and the decoded output probabilities from H_{enc} during the *Type Inference* stage (i.e., H_{dec}), the loss can be calculated as follows.

$$\mathcal{L}_{sim} = cosine\ (H_{enc}, H_{dec}) \tag{9}$$

where *cosine* means the cosine similarity calculation. Since the *Semantic Encoder* and *Type Inference* are two distinct network modules, their representation spaces may not naturally align [63]. By incorporating a similarity loss, the decoding process within *Type Inference* stage can more effectively utilize the information captured by the *Semantic Encoder* to ensure semantic consistency. By maximizing this similarity, the inferred types can be prompted to align more closely with the understanding of the original inputs of *Semantic Encoder*, thereby improving the consistency and type inference performance of the model. Consequently, *WasmHint* can be trained jointly to achieve the overall optimization objective.

$$\mathcal{L}_{obj} = \mathcal{L}_{ty} - \mathcal{L}_{sim} \tag{10}$$

Integrating the *Semantic Corrector* into the design of *WasmHint* encourages the *Semantic Encoder* and *Type Inference* components to not only concentrate on their specific tasks but also to ensure a knowledge consistency between them, ultimately focusing on generating more precise types.

4 Evaluation

4.1 Data Collection

As a deep learning model, *WasmHint* needs sufficient training data. This data needs to not only contain a wide variety of Wasm smart contracts but also has labels and enough scale to train the model. Unfortunately, there are no readily available datasets to meet this requirement. Therefore, we build a new dataset from scratch according to the following three steps.

Rust smart contract collection. Since we focus on Rust smart contracts in this paper, we retrieve from GitHub with the keywords "*Rust Smart Contract*" and download all the projects. We manually analyze the 768 collected projects and remove those that implement concrete blockchain platforms rather than smart contracts. As a result, we retain a total of 369 smart contract projects.

Wasm smart contract generation. To generate the corresponding Wasm contracts, we use the *cargo* toolchain to build each project and specify the compile target as *wasm32*. In addition, we also add a *RUSTFLAGS* command with the parameters "*–emit=llvm-ir*" and "*-C debuginfo=2*" in which the former tells the Rust compiler to emit LLVM IR during the compilation and the latter enables the compiler to output debugging information of source code level. This additional information can help us match source Rust functions to low-level Wasm functions and extract the corresponding label information from it.

Matching & labeling. Recall that our purpose is to infer high-level types of function parameters and return values from the Wasm contract functions. Thus, after acquiring these compiled smart contracts along with supplementary information, we align the low-level Wasm contracts with the corresponding high-level Rust information at the function level. More specifically, we match each Wasm function to its corresponding representation in LLVM IR. At the same time, we interpret the debugging information generated to annotate each Wasm function with the corresponding high-level type information, encompassing both parameters and return values. Based on the debugging information in LLVM IR, we first obtain the meta information of the source code level for each function from the corresponding definition of *DISubprogram*. To obtain the parameter types and return types, for each *DISubprogram*, we obtain all the type information from *DISubroutineType* and extract the raw types by recursively parsing type data according to its type scope *DIScope* in

LLVM IR. Since the extracted high-level types may have nested structures, to facilitate the model training, we treat them as composite types and convert them into corresponding type sequences. This operation facilitates the representation of complex types as the recursive combinations of types defined in Fig. 5 while reducing the size of the type vocabulary [57]. After data cleaning, we eventually obtain a total of 77,208 Wasm contract functions with type labels to train and evaluate *WasmHint* for the type inference task. The five most common types in the dataset, defined by our type set, are as follows: *ptr* (25.19%), *named type* (15.54%), *pointers to named types* (13.92%), *trait* (11.02%), and *str* (5.92%). Additionally, we observe that both primitive and composite types are present in the type definitions of parameters and return values.

4.2 Experimental Setup

We implement*WasmHint* in about 6,200 lines of Python code. The libraries PyTorch [15] and transformers [21] are employed to facilitate the model construction. For the hyper-parameters, six layers of *Semantic Encoder* with eight attention heads are stacked to make the model fully capture type-related semantics. Moreover, we configure the hidden size to 256 and set the batch size to 64. The Adam algorithm [50] is used to optimize model parameters. We randomly allocate 80% of the collected data for training, with the remaining 20% serving as the test set to prevent overlap. Our experiments are run on a Ubuntu server equipped with NVIDIA GeForce RTX 3090 24GB GPUs.

4.3 RQ1: How effective is *WasmHint*?

Approach & Results. *WasmHint* aims to infer the high-level types of function parameters and return values from Wasm smart contracts by learning the operation semantic information. To measure the effectiveness of *WasmHint* for type inference, we train classification models to identify basic types and generation models to infer composite types. We report the accuracy of *WasmHint* on our collected dataset for inferring both parameter and return types. Overall, *WasmHint* achieves an average accuracy of 80.0% for inferring parameter types and 95.8% for recovering return types. Additionally, for composite types, *WasmHint* attains an accuracy of 64.5% for generating parameter types and 79.5% for generating return types.
Manual Analysis. We conduct a manual analysis of the prediction results and present representative examples to demonstrate the model's practical capabilities. As shown in Fig. 6 (a), this example implements the operation of withdrawing funds in the CosmWasm platform. When recovering the first parameter at Rust level, *WasmHint* collects the parameter loading and corresponding subsequent operations in the Wasm code. Finally, *WasmHint* can successfully infer it as a pointer to a used-defined named type (Fig. 5). We consider this prediction valuable for reverse engineers seeking to comprehend the Wasm code. This is because the named type, such as *cosmwasm_std::Extern*, may delineate the interaction interface between the smart contract and external calls. When a function includes this parameter, it signifies that the function will trigger the modification of blockchain states. Such possible insights help reverse engineers pinpoint vulnerable features, as state changes on the blockchain affect user assets [37]. Fig. 6 (b) shows another instance of a price sale contract modifying the market state. It is notable that even though the return value is present in the Rust code (Line 1), there is a lack of a *result* declaration in the corresponding Wasm code (Line 8). This absence could impede reverse engineers in comprehending the source code. In contrast, *WasmHint* can infer this return type by learning details associated with the first Wasm parameter, enabling reverse engineers to understand the expected function behavior [84].

We also manually analyze the incorrect inference and identify the following causes of failure. First, *WasmHint* will confuse the types *u32* and *ptr*, accounting for 2.1% of the cases. As shown in Fig. 7 (a), *WasmHint* incorrectly predicts the second parameter type *u32* as *ptr*. During compilation, all high-level types are transformed to basic numeric types; specifically, both *u32* and *ptr* are

```rust
fn withdraw(deps:&mut Extern, id:u64)-> Response {
  // Load contract storage
  let mut ido = Ido::load(&deps.storage, id)?;
  ... // Details are omitted for simplicity
  ido.withdrawn = true;
  ido.save(&mut deps.storage)?; Ok( ... )}
// Wasm function declaration
func $withdraw (param i32 i64) (result i32)
```

(a) Example of a CosmWasm contract

```rust
pub fn process(&mut self) -> Result<()> {
  let market = &mut self.market;
  let clock = &self.clock;
  ... // Details are omitted for simplicity
  market.state = MarketState::Ended;
  Ok(())}
// Wasm function declaration
func $process (param i32 i32)
```

(b) Example of a price sale contract

Fig. 6. Representative examples of the predicted types.

mapped to low-level $\hat{i32}$ types in Wasm, as described in § 2.4. When involving memory operations, pointers will be treated as ordinary integers, such as calculating offsets. Likewise, operations involving *u32* might simply involve arithmetic manipulations on pointers, which will not be clearly distinguishable in the compiled Wasm code. Second, *WasmHint* will incorrectly identify *trait* and *user-defined (named)* types, which constitute 1.8% of the cases. As illustrated in Fig. 7 (b), *WasmHint* cannot infer this parameter as *trait*. This is because in Rust, *traits* are commonly used to implement polymorphism for *user-defined* types. Although each *user-defined* type can implement multiple *traits*, this is usually not directly observable in compiled Wasm code. Consequently, when a function accepts arguments of any type associated with a specific *trait*, its representation in Wasm might be limited to non-specific type function calls. This representation fails to explicitly outline the underlying type relationships, complicating the distinction between these types in the Wasm code. Third, *WasmHint* will confuse the types *ptr* and *str* in the absence of explicit manipulation of string content within Wasm functions, accounting for 1.5% of the cases. As shown in Fig. 7 (c), *WasmHint* inaccurately infers the second parameter as *ptr* instead of *str*. This confusion arises because in Rust, the *str* type is a dynamically sized type that holds textual data and can only be instantiated as a pointer type [18]. The *str* type is stored in memory as a combination of a pointer to the data and the length of that data. Correspondingly, in Wasm, operations involving a *str* typically start by retrieving the pointer and then accessing the data it references. As a result, without sufficient contextual information, distinguishing between a *ptr* and a *str* can be difficult.

Answer to RQ1: *WasmHint* can precisely inferring high-level types from Wasm contracts, achieving overall accuracies of 80.0% for parameter types and 95.8% for return types.

4.4 RQ2: How does *WasmHint* compare to the existing methods?

Approach. Given that some existing studies have focused on recovering parameter and return types, we want to explore how *WasmHint* performs compared to these state-of-the-art methods. More specifically, we compare *WasmHint* with two baseline methods: SnowWhite [57] and Eklavya [39]. The former utilizes a neural machine translation-based generation model to infer possible types [57], while the latter employs a classification model based on a collected set of types to predict possible types [39]. Since they cannot be directly used for recovering high-level Rust types from Wasm code, we adapt these methods to our task while preserving their fundamental model architectures. More specifically, the adaptation process of each baseline involves in three phases: instruction sequence extraction, sequence learning, and type inference. In instruction sequence extraction phase, SnowWhite extracts context within a fixed-size window around parameter or return-related instructions according to its original design [57], while Eklavya employs our operation-centric code slicing method (§ 3.3) to deal with the Wasm code as it is initially designed for C/C++ code [39]. In sequence learning phase, both methods use their own model architectures with default settings for

```
pub fn add_config_lines(ctx:
    Context,index: u32) ->
    Result<()> {...
  let total = index.add(ctx.
    config_lines.len())
   .ok_or(...)?; ... }
```

(a) Case 1

```
impl<'info> TryFrom<Cancel<'
    info>> for Initializer {
  fn try_from(value: Cancel<'
    info>) -> Result<Self>{
    let init_key = &value.
    initializer.key;...}}
```

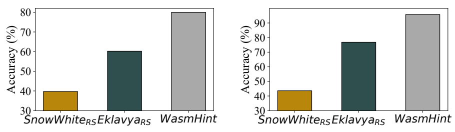
(b) Case 2

```
pub fn get_order(&self, id:
    String) -> Order {
  let order = match self.
    order_list.get(&id) {
  ...};
  order}
```

(c) Case 3

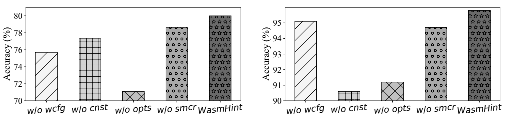
Fig. 7. Examples of the incorrect inference by *WasmHint*.

learning, i.e., bidirectional LSTM model with global attention for SnowWhite and recurrent neural networks (RNN) for Eklavya. Since these models can merely handle one-to-one parameter or return correspondence, we employ average pooling operations across multiple instruction sequences to aggregate the learned information for facilitating subsequent type inference. In the final stage, SnowWhite continues to leverage the bidirectional LSTM model, whereas Eklavya persists with the RNN model for type inference. The difference is that both models apply our defined recursive type language as shown in Fig. 5. We refer to these two adapted baselines as SnowWhite$_{RS}$ and Eklavya$_{RS}$, respectively, to prevent confusion with the original methods. To make a fair comparison, we use the same 80/20 training/testing dataset as *WasmHint* for experiments. We report the average accuracy of these methods in separately inferring parameter and return types.
Results. Fig. 8 illustrates the average accuracy of *WasmHint* and the two baselines. We can find that *WasmHint* obtains the average improvements by 101.3% and 25.7% compared to SnowWhite$_{RS}$ and Eklavya$_{RS}$ when inferring parameter types, respectively. Meanwhile, *WasmHint* achieves average improvements of 119.8% and 24.6% compared to the baseline methods in recovering return types. This result indicates that SnowWhite$_{RS}$ and Eklavya$_{RS}$ fail to effectively learn the operational semantics of function parameters and return values, thereby impairing its ability to accurately recover types. Furthermore, due to the fixed set of type labels, Eklavya$_{RS}$ is unable to adapt to new type combinations. Instead, *WasmHint* extracts and integrates semantic information from various aspects of Wasm code, enhancing its accuracy in type recovery. Overall, *WasmHint* demonstrates average improvements of 63.5% and 72.2% in recovering parameter and return types, respectively.



(a) Parameter (b) Return

Fig. 8. Comparison results with baselines.



(a) Parameter (b) Return

Fig. 9. Ablation results of *WasmHint*.

Answer to RQ2: *WasmHint* outperforms the baseline methods, illustrating average improvements of 63.5% and 72.2% in inferring parameter and return types, respectively.

4.5 RQ3: How does *WasmHint* compare to pre-trained models for binary code?

Approach. Since pre-training techniques have been utilized for binary code analysis [66], we want to investigate how *WasmHint* performs in comparison to these pre-training techniques for binary or assembly code. We introduce three traditional binary/assembly code models: Instr2Vec [54], Asm2Vec [41], and PalmTree [59]. Specifically, Instr2Vec learns embeddings for individual instructions collected from instruction sequences [54]. Asm2Vec generates embedding vectors for each assembly code by employing the PV-DM model [53] and incorporating the syntactic structure of instructions [41]. PalmTree adopts the BERT architecture [40] and designs three pre-training objectives to generate instruction embeddings [59]. In addition, we also introduce two state-of-the-art models: CodeArt [77] and WaDec [75]. CodeArt incorporates attention regularization into Transformer to learn embeddings from stripped binary code [77]. WaDec fine-tunes the CodeLlama-7b-hf model to learn Wasm code for decompilation [75]. For each model, we design two strategies: direct fine-tuning and integration with *WasmHint*. In the former, feedforward layers are added to each model, and the model parameters are fine-tuned directly on the Wasm code without incorporating any design from *WasmHint*. In contrast, the latter strategy replaces the token embedding component of *WasmHint* with each model to generate the initial embeddings.

Table 1. Comparison results between *WasmHint* and pre-trained models.

Method	Instr2Vec	Asm2Vec	PalmTree	CodeArt	WaDec	*WasmHint*
Parameter	33.3 $\longrightarrow$ 80.2	29.2 $\longrightarrow$ 43.4	29.7 $\longrightarrow$ 78.7	34.3 $\longrightarrow$ 79.0	25.9 $\longrightarrow$ 80.4	80.0
Return	67.6 $\longrightarrow$ 94.8	66.8 $\longrightarrow$ 67.0	67.1 $\longrightarrow$ 92.8	69.2 $\longrightarrow$ 90.7	66.1 $\longrightarrow$ 96.1	95.8

Results. Table 1 shows the average accuracy of *WasmHint* and the ten baselines. The values on the left and right of the arrows indicate the results of direct fine-tuning and integration with *WasmHint*, respectively. We can see that *WasmHint* achieves the average improvements by 140.2%, 174.0%, 169.4%, 133.2%, and 208.9% compared to the five baselines with direct fine-tuning strategy when inferring parameter types, respectively. Furthermore, *WasmHint* obtains average improvements of 41.7%, 43.4%, 42.8%, 38.4%, and 44.9% compared to the baselines in recovering return types. When integration them with *WasmHint*, the inference accuracies for parameters and returns demonstrate average improvements of 139.0% and 31.1% respectively across the five baseline methods. In particular, by integrating WaDec with *WasmHint*, the inference performance surpasses the original design of *WasmHint* compared with other pre-trained baselines. This improvement is attributed to prior knowledge of Wasm instructions learned in WaDec. These results also demonstrate that while the code pre-training models have shown success in some binary analysis tasks [66], additional designs are necessary for performance improvement in specific tasks like our type inference.

Answer to RQ3: *WasmHint* is superior to using binary or assembly code pre-trained models directly, and integrating these models with *WasmHint* can improve their performance.

4.6 RQ4: How do the designed components impact *WasmHint*?

Approach. As described in § 3.4, *WasmHint* aggregates related instruction operations, extracted constants, and the constructed *w*CFG to form the overall semantics for type inference. Additionally, it includes a semantic corrector to refine the differences between the outputs of the semantic encoder and the inferred types, as shown in § 3.6. Therefore, we want to investigate whether these aggregated features and the designed module genuinely improve the performance of *WasmHint* for type inference. To achieve this goal, we design several variants by individually removing different components. This results in the following variants: discarding the *w*CFG (*w/o wcfg*), excluding the constants (*w/o cnst*), and removing the type-related operations (*w/o opts*). Additionally, we include a variant that does not use the semantic corrector (*w/o smcr*).

Results. Fig. 9 presents the average results for *WasmHint* and the four variants. The results illustrate the effectiveness of each designed component in enhancing the performance of *WasmHint*. Specifically, the average accuracy of *WasmHint* achieves improvements by 5.7%, 3.5%, 12.5%, and 1.8% compared to the four variants in recovering parameter types, respectively. Meanwhile, it obtains average improvements by 0.7%, 5.7%, 5.0%, and 1.1% compared to the variants in inferring return types, respectively. Although the constructed *w*CFG contains code semantics, it fails to capture detailed instruction operations. In contrast, when the operation-related sequences are integrated, *WasmHint* not only assimilates the relevant operational semantics but also harnesses global code information from the *w*CFG, thereby enhancing its capability for type recovery. Additionally, constant values influence the performance of *WasmHint* in recovering both parameter and return types. This effect demonstrates that the presence of constants allows the model to understand the operations of specific types, thus improving its inference accuracy. Moreover, the semantic corrector module ensures that the encoder outputs are aligned with the predicted types, further enhancing the inference performance of *WasmHint*.

Answer to RQ4: *WasmHint* gains significant advantages from the designed components, improving its capability for inferring both parameter and return types.

4.7 RQ5: What is the overhead of *WasmHint*?

Approach. *WasmHint* is a data-driven, learning-based model that typically requires graphics processing unit (GPU) to accelerate its training process. This drives us to explore the resource usage and time consumption associated with *WasmHint*. Therefore, we execute and record the GPU memory usage and time overhead during the training and testing of *WasmHint*. We report the memory usage in Megabyte (MB) and average time in seconds.

Results. The total GPU memory usage of *WasmHint* for inferring function parameter and return types amounts to 8,962 MB during training and 8,427 MB during testing. This result indicates that our model has low memory consumption, facilitating training on a variety of GPU devices. Moreover, the total training time for the parameter type inference model is 12,950 seconds, while the return type recovery model requires 5,981 seconds. Additionally, the average time to recover the parameter and return types for each function is 0.002 seconds and 0.0004 seconds, respectively. Although training *WasmHint* is time-consuming, it requires only a one-time, offline effort. Once trained, the model can be repeatedly used online.

Answer to RQ5: *WasmHint* exhibits low overhead in inferring parameter and return types.

5 Discussion

We discuss the limitations of *WasmHint* and possible solutions to be explored in this section.

First, *WasmHint* is designed as a deep learning-based architecture whose effectiveness is inherently limited by the training data itself because of the data-driven nature [28, 79, 84]. This situation will result in suboptimal inference performance for some of the rare types. One viable solution involves adopting data augmentation techniques, such as synthetic data generation [60], to expand the dataset volume while preserving the fundamental semantic integrity. These strategies offer promising avenues to enhance the capabilities of *WasmHint* to handle rare types.

Second, when extracting instruction sequences related to parameter or return operations, the input parameter itself is treated as a symbolic value (§ 3.3). During the parameter propagation process, particularly in scenarios involving memory operations, accurately calculating memory slots poses challenges due to the reliance on symbolic computations. In addition, the simulation cannot guarantee every instruction is covered in a function, as it requires balancing instruction extraction with the need to avoid infinite loops. These will lead to a partial information loss in the extracted operation sequences to some extent. To mitigate this issue, we enhance our model by incorporating the overall code semantics from *w*CFG, which helps compensate for the information loss resulting from potentially incomplete operation semantics.

Third, the current design of *WasmHint* does not explicitly consider the impact of compilation optimization when compiling Rust smart contracts into Wasm counterparts. These optimizations, such as inline expansion, loop optimization, and vectorization, are applied by the compiler to enhance performance [7]. They will lead to certain type-related operations being hidden, potentially hindering the ability of *WasmHint* to accurately capture semantic operations.

Fourth, *WasmHint* primarily focuses on Rust, a language favored by developers for developing smart contracts due to its memory-safe capabilities [6, 13]. It is straightforward to determine if a Wasm code is compiled from Rust by examining its contents. For example, symbols such as *rust_panic* can serve as an interface with the Rust standard library to handle unrecoverable errors via the panic mechanism [17] in Wasm. Given sufficient data, *WasmHint* can also be applied to Wasm compiled from other languages, requiring extra effort to analyze their compilation strategies.

6 Related Work

There is a lot of existing work focused on the reverse engineering of smart contracts from EVM bytecode, such as Erays [86], Vandal [32], Mythril [12], and Gigahorse [43]. In addition to reversing bytecode to a high-level representation, other work involved recovering data types from the EVM bytecode, such as EVM bytecode decompiler [2], online solidity decompiler [3], JEB [4], Eveem [1], SigRec [36], and DeepInfer [84]. However, all these efforts merely focused on the reverse engineering of the EVM bytecode, which cannot be applied to recover high-level types from Wasm code due to the differences in instruction operations and runtime semantics [22, 24, 25].

Other work also focused on understanding Wasm code, including function purpose identification [74], type inference [57], and reverse engineering [42, 48, 75]. However, all these existing studies are confined to high-level C/C++ code, limiting its practical application in the blockchain domain. Moreover, some efforts concentrated on inspecting Wasm code using various program analysis-based techniques, including dynamic analysis [56], symbolic execution [47], and fuzzing techniques [49]. Additionally, there has been a concern about the security analysis of Wasm code [38, 55] and Wasm runtimes [34, 83, 85]. Different from these studies, our work concentrates on recovering high-level Rust types of parameters and return values from Wasm contracts, which can be regarded as a complement to existing studies.

7 Conclusion

We present *WasmHint*, a novel solution that leverages deep learning inference to automatically recover high-level parameter and return types from Wasm smart contracts. This method simulates code execution to capture type-related operations and designs an encoder network to learn the underlying code semantics. We collect and build the first dataset focusing on type recovery in Wasm contracts from real-world Rust smart contract projects. The comprehensive experiments demonstrate that *WasmHint* produces more accurate high-level types than baseline methods, with each designed component playing a crucial role in its success. In the future, we plan to extend our method to support the analysis of Wasm contracts compiled from other languages.

8 Data Availability

Our experimental materials are available at https://github.com/sepine/WasmHint.

Acknowledgments

We sincerely thank the anonymous reviewers for their constructive comments. This work is partly supported by Hong Kong RGC Projects (PolyU15224121, PolyU15231223), Hong Kong ITF Project (No. PRP/005/23FX), National Natural Science Foundation of China (62332004), and Sichuan Provincial Natural Science Foundation for Distinguished Young Scholars (2023NSFSC1963).

References

[1] 2019. Eveem. https://eveem.org.
[2] 2019. EVM bytecode decompiler. https://github.com/MrLuit/evm.
[3] 2021. Online solidity decompiler. https://ethervm.io/decompile.
[4] 2022. Ethereum smart contract decompiler. https://www.pnfsoftware.com/blog/ethereum-smart-contract-decompiler.
[5] 2024. Astar. https://astar.network/.
[6] 2024. CosmWasm. https://cosmwasm.com/.
[7] 2024. Customizing Builds with Release Profiles in Rust. https://doc.rust-lang.org/book/ch14-01-release-profiles.html.
[8] 2024. EOSIO Overview. https://developers.eos.io/manuals/eos/latest/index/.
[9] 2024. Framework for building smart contracts in Wasm for the Cosmos SDK. https://github.com/CosmWasm/cosmwasm.
[10] 2024. LLVM IR. https://llvm.org/docs.
[11] 2024. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html.
[12] 2024. Mythril. https://github.com/ConsenSys/mythril.
[13] 2024. Near Blockchain. https://near.org/.
[14] 2024. POL: One token for all Polygon chains. https://polygon.technology/papers/pol-whitepaper/.
[15] 2024. PyTorch Framework. https://pytorch.org/.
[16] 2024. Rust library for writing NEAR smart contracts. https://github.com/near/near-sdk-rs.
[17] 2024. Rust Programming Language. https://www.rust-lang.org/.
[18] 2024. The Rust reference: types. https://doc.rustlang.org/reference/types.html.
[19] 2024. Solana Blockchain. https://solana.com/.
[20] 2024. The Solidity Contract-Oriented Programming Language. https://github.com/ethereum/solidity/.
[21] 2024. Transformers: State-of-the-art Machine Learning for Pytorch, TensorFlow, and JAX. https://github.com/huggingface/transformers.
[22] 2024. WebAssembly. https://webassembly.org/.
[23] 2024. The WebAssembly binary toolkit. https://github.com/WebAssembly/wabt.
[24] 2024. WebAssembly functions. https://webassembly.github.io/spec/core/syntax/modules.html#functions.
[25] 2024. WebAssembly opcodes. https://pengowray.github.io/wasm-ops/.
[26] 2024. WebAssembly-Rust Programming Language. https://www.rust-lang.org/what/wasm.
[27] 2024. WebAssembly types. https://webassembly.github.io/spec/core/syntax/types.html.
[28] Kuznetsova Alina, Hassan Rom, Alldrin Neil, Uijlings Jasper, Krasin Ivan, Pont-Tuset Jordi, Kamali Shahab, Stefan Popov, Malloci Matteo, Alexander Kolesnikov, et al. 2020. The Open Images Dataset V4. *International Journal of Computer Vision* 128, 7 (2020), 1956–1981.
[29] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 91–105.
[30] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).
[31] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. *Advances in Neural Information Processing Systems (NeurIPS)* 28 (2015).
[32] 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 preprint arXiv:1809.03981* (2018).
[33] Juan Caballero and Zhiqiang Lin. 2016. Type inference on executables. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 1–35.
[34] Shangtong Cao, Ningyu He, Xinyu She, Yixuan Zhang, Mu Zhang, and Haoyu Wang. 2023. WRTester: Differential Testing of WebAssembly Runtimes via Semantic-aware Binary Generation. *arXiv preprint arXiv:2312.10456* (2023).
[35] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. 2021. Decision transformer: Reinforcement learning via sequence modeling. *Advances in Neural Information Processing Systems (NeurIPS)* 34 (2021), 15084–15097.
[36] Ting Chen, Zihao Li, Xiapu Luo, Xiaofeng Wang, Ting Wang, Zheyuan He, Kezhao Fang, Yufei Zhang, Hang Zhu, Hongwei Li, et al. 2021. Sigrec: Automatic recovery of function signatures in smart contracts. *IEEE Transactions on Software Engineering (TSE)* 48, 8 (2021), 3066–3086.
[37] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1503–1520.
[38] Weimin Chen, Zihan Sun, Haoyu Wang, Xiapu Luo, Haipeng Cai, and Lei Wu. 2022. Wasai: uncovering vulnerabilities in wasm smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and*

Analysis (ISSTA). 703–715.

[39] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural nets can learn function type signatures from binaries. In *Proceedings of the 26th USENIX Security Symposium (USENIX SEC)*. 99–116.

[40] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[41] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*. 472–489.

[42] Weike Fang, Zhejian Zhou, Junzhou He, and Weihang Wang. [n. d.]. StackSight: Unveiling WebAssembly through Large Language Models and Neurosymbolic Chain-of-Thought Decompilation. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*.

[43] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In *Proceedings of 41st IEEE/ACM International Conference on Software Engineering (ICSE)*. 1176–1186.

[44] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 185–200.

[45] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems (NeurIPS)* 30 (2017).

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[47] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. *EOSAFE*: Security analysis of *EOSIO* smart contracts. In *30th USENIX Security Symposium (USENIX SEC)*. 1271–1288.

[48] Hanxian Huang and Jishen Zhao. 2024. Multi-modal Learning for WebAssembly Reverse Engineering. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[49] Bo Jiang, Zichao Li, Yuhe Huang, Zhenyu Zhang, and W Chan. 2022. Wasmfuzzer: A fuzzer for webassembly virtual machines. In *Proceedings of the 34th International Conference on Software Engineering and Knowledge Engineering, (SEKE)*. 537–542.

[50] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[51] Jae Kwon and Ethan Buchman. 2019. Cosmos whitepaper. *A Netw. Distrib. Ledgers* 27 (2019), 1–32.

[52] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of 2004 International Symposium on Code Generation and Optimization (CGO)*. 75–86.

[53] Quoc Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31st International Conference on Machine Learning (ICML)*. 1188–1196.

[54] Young Jun Lee, Sang-Hoon Choi, Chulwoo Kim, Seung-Ho Lim, and Ki-Woong Park. 2017. Learning binary code with deep learning to detect software weakness. In *Proceedings of the 9th International Conference on Internet (ICONI)*.

[55] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything old is new again: Binary security of *WebAssembly*. In *Proceedings of the 29th USENIX Security Symposium (USENIX SEC)*. 217–234.

[56] Daniel Lehmann and Michael Pradel. 2019. Wasabi: A framework for dynamically analyzing webassembly. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1045–1058.

[57] Daniel Lehmann and Michael Pradel. 2022. Finding the dwarf: recovering precise types from WebAssembly binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 410–425.

[58] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[59] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. Palmtree: Learning an assembly language model for instruction embedding. In *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 3236–3251.

[60] Zhuoyan Li, Hangxiao Zhu, Zhuoran Lu, and Ming Yin. 2023. Synthetic Data Generation with Large Language Models for Text Classification: Potential and Limitations. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[61] Ruichao Liang, Jing Chen, Kun He, Yueming Wu, Gelei Deng, Ruiying Du, and Cong Wu. 2024. Ponziguard: Detecting ponzi schemes on ethereum with contract runtime behavior graph (crbg). In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. 1–12.

[62] Lu Liu, Lili Wei, Wuqi Zhang, Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2021. Characterizing transaction-reverting statements in ethereum smart contracts. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 630–641.

[63] Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov, Marjan Ghazvininejad, Mike Lewis, and Luke Zettlemoyer. 2020. Multilingual Denoising Pre-training for Neural Machine Translation. *Transactions of the Association for Computational Linguistics (TACL)* 8 (2020), 726–742.

[64] Zuchao Ma, Muhui Jiang, Feng Luo, Xiapu Luo, and Yajin Zhou. 2025. Surviving in Dark Forest: Towards Evading the Attacks from Front-Running Bots in Application Layer. In *Proceedings of the 34th USENIX Security Symposium (USENIX SEC)*.

[65] Zuchao Ma, Muhui Jiang, Xiapu Luo, Haoyu Wang, and Yajin Zhou. 2025. Uncovering NFT Domain-Specific Defects on Smart Contract Bytecode. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2025).

[66] Alwin Maier, Felix Weißberg, and Konrad Rieck. 2024. On the Role of Pre-trained Embeddings in Binary Code Analysis. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*. 1143–1158.

[67] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. Nl2type: inferring javascript function types from natural language information. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*. 304–315.

[68] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4py: Practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. 2241–2252.

[69] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. Stateformer: Fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 690–702.

[70] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static inference meets deep learning: a hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. 2019–2030.

[71] Yaohui Peng, Jing Xie, Qiongling Yang, Hanwen Guo, Qingan Li, Jingling Xue, and Mengting Yuan. 2023. Statistical Type Inference for Incomplete Programs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 720–732.

[72] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[73] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[74] Alan Romano and Weihang Wang. 2023. Automated WebAssembly Function Purpose Identification With Semantics-Aware Analysis. In *Proceedings of the ACM Web Conference 2023*. 2885–2894.

[75] Xinyu She, Yanjie Zhao, and Haoyu Wang. 2024. WaDec: Decompiling WebAssembly Using Large Language Model. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 481–492.

[76] Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. 1–13.

[77] Zian Su, Xiangzhe Xu, Ziyang Huang, Zhuo Zhang, Yapeng Ye, Jianjun Huang, and Xiangyu Zhang. 2024. Codeart: Better code models by attention regularization when symbols are lacking. *Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE)* (2024), 562–585.

[78] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS)* 30 (2017).

[79] Xiaozhi Wang, Ziqi Wang, Xu Han, Wangyi Jiang, Rong Han, Zhiyuan Liu, Juanzi Li, Peng Li, Yankai Lin, and Jie Zhou. 2020. MAVEN: A Massive General Domain Event Detection Dataset. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1652–1671.

[80] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014), 1–32.

[81] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. 1480–1489.

[82] Wen Zhang, Yang Feng, Fandong Meng, Di You, and Qun Liu. 2019. Bridging the Gap between Training and Inference for Neural Machine Translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*.

[83] Yixuan Zhang, Shangtong Cao, Haoyu Wang, Zhenpeng Chen, Xiapu Luo, Dongliang Mu, Yun Ma, Gang Huang, and Xuanzhe Liu. 2023. Characterizing and detecting webassembly runtime bugs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 33, 2 (2023), 1–29.

[84] Kunsong Zhao, Zihao Li, Jianfeng Li, He Ye, Xiapu Luo, and Ting Chen. 2023. DeepInfer: Deep Type Inference from Smart Contract Bytecode. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* 745–757.

[85] Shiyao Zhou, Muhui Jiang, Weimin Chen, Hao Zhou, Haoyu Wang, and Xiapu Luo. 2023. WADIFF: A Differential Testing Framework for WebAssembly Runtimes. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 939–950.

[86] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. 2018. Erays: reverse engineering ethereum's opaque smart contracts. In *Proceedings of the 27th USENIX Security Symposium (USENIX SEC).* 1371–1385.

Received 2024-09-12; accepted 2025-04-01