WasmWalker: Path-based Code Representations for Improved WebAssembly Program Analysis

by

Mohammad Robati Shirzad

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2023

Abstract

WebAssembly, or Wasm, is a low-level binary language that enables execution of near-native-performance code in web browsers. Wasm has proven to be useful in applications including gaming, audio and video processing, and cloud computing, providing a high-performance, low-overhead alternative to JavaScript in web development. The fast and widespread adoption of WebAssembly by all major browsers has created an opportunity for analysis tools that support this new technology.

In this study, we performed an empirical analysis on the root-to-leaf paths of the abstract syntax trees in the WebAssembly Text format of a large dataset of WebAssembly binary files compiled from over 4,000 source packages in the Ubuntu 18.04 repositories. After refining the collected paths, the initial number of over 800,000 paths was reduced to only 3,352 unique paths that appeared across all of the binary files.

With this insight, we propose two novel code representations for WebAssembly binaries. These novel representations serve not only to generate fixed-size code embeddings but also to supply additional information to sequence-to-sequence models. Ultimately, our approach seeks to help program analysis models uncover new properties from Wasm binaries, expanding our understanding of their potential. We evaluated our new code representation on two applications: (i) method name prediction and (ii) recovering precise return types. Our results demonstrate the superiority of our novel technique over previous methods. More specifically, our new method resulted in 5.36% (11.31%) improvement in Top-1 (Top-5) accuracy in method name prediction and 8.02% (7.92%) improvement in recovering precise return types, compared to the previous state-of-the-art technique, SnowWhite.

Acknowledgements

I would like to express my sincere gratitude to Patrick, for his unwavering support during my studies at the university. His knowledge, feedback, and insights have been crucial in guiding my research and helping me tackle all the challenges that come with graduate studies.

Dedication

This thesis is dedicated to my parents and brother with love, and gratitude for everything they have done for me. I am especially grateful to my parents for their exceptional parenting.

“Plant wheat for a year’s intent, Plant trees for a decade’s scope, Educate people, for a century’s aim.” –Mirza Taghi Khan-e Farahani

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

WebAssembly (Wasm) is a cross-platform low-level binary format that enables browsers to execute code with near-native performance. Wasm is an open standard and currently being maintained by W3C. The technology was created as a collaboration between W3C, Mozilla, Google, Microsoft, and Apple [16]. The development of Wasm began in 2015, and version 1.0 was officially released in 2017. Wasm seeks to go beyond Javascript's limitations, providing a secure platform for executing programs in web browsers. Some of JavaScript's problems include: (1) its dynamic nature requires heroic efforts to execute efficiently; (2) as a client-side language, JavaScript has limited access to memory and the file system; and (3) JavaScript's multiprocessing capabilities are restricted. Wasm was developed with the aim of filling these gaps. In terms of facilitating the deployment of platform-independent code and providing a security model that ensures that the system functions in a secure and reliable manner, the role of Wasm on the client is comparable to the role that Java performed in the early 2000s on the server with the Java Virtual Machine.

Wasm is becoming more and more popular: Wasm's adoption has created new opportunities for web development, enabling programmers to consider using Wasm in performance-critical applications, such as cloud computing, video conferencing, and gaming. Some well-known programming languages, including C, C++, and Rust, now have Wasm compilers available. Wasm's popularity serves as an invitation for more tools and techniques to be created to support this newly-emerged technology.

Recently, researchers have conducted empirical studies on Wasm, primarily targeting three properties: (1) Extent of use: In 2019, Musch et al. [34], after examining the prevalence of WebAssembly in the Alexa's Top 1 million websites, reported that 1 out of 600 sites execute Wasm code. Two years later, Hilbig et al. [18] found that complex, real-world applications are compiled to WebAssembly and used across a wide range of applications. More specifically, in their dataset, which included Wasm binaries from GitHub, npm, and websites from HTTP archive [1], two thirds of binaries had more than 8,700 instructions and the median binary had 14,885 instructions. (2) Security: Studies have focused on the security of WebAssembly and found that some security vulnerabilities that are not considered risks in native binaries are exploitable in WebAssembly [24, 26, 35, 18], showing a need for hardening the security layer of Wasm. In WebAssembly, stack-based buffer overflows are again effective because Wasm binaries do not utilize stack canaries by default. To resolve these issues, some researchers have presented security vulnerability scanner prototypes to mitigate risk [9]. (3) Performance: Jangda et al. [21], after extending the BROWSIX environment [36], conducted a large-scale comparison between Wasm and native code. The results showed slower execution time for Wasm than native code, as executed by Google Chrome and Firefox. More recently, Yan et. al [47] demonstrated that WebAssembly code uses significantly more memory than its JavaScript analog.

This context motivated us to develop new tools and techniques for static analysis of Wasm. We focus particularly on techniques that can help with deep learning: the application of deep learning in static analysis has brought about significant advancements in the field. Deep learning models are able to analyze large datasets quickly and can predict properties of unseen data with high accuracy and consistency. This has made deep learning one of the most popular tools for static analysis, and its popularity is growing as the learning methods are constantly evolving. Deep learning can be used for many interesting properties that we target in static analysis. For instance, researchers have developed deep learning tools to detect security vulnerabilities in high level programming languages [29, 48, 7], and also native binaries [4]. Other applications include defect prediction [31, 37] and clone detection [28, 45].

The input provided to deep learning models greatly impacts their performance and accuracy. If the input contains the right information and is *structured* correctly, models can make highly-accurate predictions. However, if the input data contains redundant information or is poorly structured, this can slow down the training process and reduce the accuracy of the models. Our goal is to devise a useful structure. Inputs can be fixed-sized or variable-sized.

Fixed-sized inputs work when the data's number of features is known: datapoints are vectors of numbers. Vector dimensions are associated with defined features. In static analysis, inputs are usually programs or parts of programs. Yet, computer programs are

[1]https://httparchive.org/

inherently variable-sized information. Thus, to use them as inputs to e.g. feedforward neural networks, we must embed them into fixed-sized vectors—a process known as code embedding [12]. Researchers proposed many approaches for embedding high-level code [2, 14, 11, 3] and native binaries [8, 49, 39, 46].

Unlike fixed-sized inputs, variable-sized inputs do not have an a priori known number of features. Variable-sized inputs are typically used when features are more complex and cannot be easily transformed into numerical values. In the case of static analysis, the input can directly be the sequence of program statements. More complex deep learning models, including RNNs, LSTMs [19], Transformer Networks [44], and GPT-3 [10], process variable-sized inputs.

In this project, we aim to incorporate Wasm's AST information inside the input to deep learning models. Towards this goal, we introduce WasmWalker, a pipeline for extracting AST information from Wasm binaries. Using WasmWalker, we conducted an empirical study to find the most frequent paths that emerge in the abstract syntax tree of WebAssembly Text (WAT) format of Wasm binaries across a large dataset. We used the same dataset that was used in SnowWhite [27], a study that offers a framework for recovering precise C/C++ high level types from low-level limited Wasm type system. The dataset includes 6.3 million type-labeled Wasm samples, extracted from 300,905 object files, which were compiled from 4,081 C and C++ Ubuntu packages.

Problem definition. The problem addressed in this project is to investigate how deep learning models can benefit from Wasm abstract syntax tree information. Our study aims include (1) identifying the most frequent paths that emerge in the AST of Wasm binaries and (2) using these paths to develop a fixed-sized feature vector for each Wasm function that can be transformed to code representations for deep learning models.

Initially, WasmWalker gathered over 800,000 root-to-leaf paths. We refined these paths to obtain a more compact and abstract representation that respects the nested structure of ASTs and the order of non-terminals within root-to-leaf paths. Our refined paths include 3,352 paths encoding conditional and loop nested structures. Building on the refined paths, we developed a fixed-sized feature vector for each Wasm function—the path vector. Each dimension of our path vectors corresponds to one of the 3,352 paths we identified, and the numerical value for that dimension represents the number of times that path occurred in the Wasm function's AST.

Using our path vectors, we developed two different code representations for Wasm functions: (1) a path sequence, that indicates the appearance of each path along with a numerical value that corresponds to the number of times that path appeares in the Wasm function, and (2) a code embedding, similar to code2vec [3], that reduces the dimensionality of our path vectors from 3,352 to 50.

We evaluated the prediction accuracy of our code representations using the dataset provided in SnowWhite [27]. We conducted two experiments. First, we attempted to predict actual method names based on Wasm binaries. Second, similar to SnowWhite, we aimed to recover high-level C/C++ return types from Wasm primitive types. Our findings indicate that a hybrid approach, where a selection of instructions is concatenated to our path sequence, resulted in the best models. Specifically, our new representation led to a 5.36% (11.31%) improvement in Top-1 (Top-5) accuracy in method name prediction and a 8.02% (7.92%) improvement in recovering precise return types. Also, we demonstrate the effectiveness of our embeddings in clustering method names that contain semantically similar method bodies close together.

Contributions: This paper makes the following contributions:

- We conduct the first empirical study on the most frequent paths in the ASTs of Wasm binaries over a large-scale dataset.
- We propose two novel representations of Wasm functions: (1) a path sequence containing AST path information, and (2) a code embedding that leverages the information we gathered about paths.
- We show that our path sequence in combination with Wasm instructions improves accuracy compared to the previous state-of-the-art, by evaluating our approach over two use cases: method name prediction and recovering precise return types.

Data Availability Statement: Datasets generated and analyzed in this work are available at `https://zenodo.org/record/7763463`. To learn about working with our replication package and how to reproduce our results, see Appendix B.

Chapter 2

Motivating Example

In this section, we discuss how Wasm binaries are generated from C/C++ source code using emscripten. Then, we illustrate how we tranform Wasm binaries to WAT files and extract AST paths from them, providing concrete examples. Finally, we demonstrate how we compute two different code representations using the AST paths.

The dataset that we used in this work contains Wasm binaries that are created by compiling C/C++ code using the Emscripten toolchain. Emscripten [1] is an open source compiler toolchain that compiles C/C++ programs, or any other languages that use LLVM as a part of their compilation process, into WebAssembly. LLVM [20] is a compiler infrastructure that offers a set of modular and reusable compiler technologies. The LLVM Intermediate Representation (LLVM IR) is an assembly-like language that acts as a middle ground between (programmer-written) high-level code and (executable) native code. An intermediate representation makes it possible for compiler developers to only focus on writing the frontend part of the compiler and to delegate the responsibility of dealing with target architecture to the back-end part of the compilation toolchain. Emscripten's primary work is performed by a series of Wasm-specific optimization passes. To produce LLVM IR code, Emscripten uses Clang, which is a part of the LLVM compiler infrastructure. As a backend, i.e. to transform LLVM IR to Wasm binaries, after using LLVM's optimizer tools to improve performance and efficiency, Emscripten uses the Binaryen tool [33] to produce Wasm binaries.

Our objective is to analyze the Wasm binary files that we obtain. However, these binary files contain a stream of binary values that make it challenging to perform structural analysis. To overcome this, we use the WebAssembly Text (WAT) format, which is a human-readable representation of Wasm code. We convert a Wasm binary to its re-

```c
int sign(int a) {
  int s = 0;
  if(a > 0) {
    s = 1;
  } else if (a < 0) {
    s = -1;
  }
  return s;
}
```

(a) C code

```
sign:
  .functype     sign (i32) -> (i32)
# %bb.0:
  local.get     0
  i32.const     31
  i32.shr_s
  i32.const     1
  local.get     0
  i32.const     1
  i32.lt_s
  i32.select
  end_function
```

(b) LLVM IR code

```
00000000: 0061 736d 0100 0000
00000008: 0192 8180 8000 1660
00000010: 017f 017f 6000 017f
00000018: 6003 7f7f 7f01 7f60
00000020: 0000 6001 7f00 6003
00000028: 7f7e 7f01 7e60 027f
00000030: 7f01 7f60 067f 7c7f
00000038: 7f7f 7f01 7f60 027f
00000040: 7f00 6002 7e7f 017f
00000048: 6004 7f7e 7e7f 0060
00000050: 047f 7f7f 7f01 7f60
...
```

(c) Wasm binary code

```
(func (type 0) (param i32) (result i32)
  (local i32 ... i32)
  global.get 0
  local.set 1
  ...
  block
    block
      local.get 11
      i32.eqz
      br_if 0
      ...
  return)
```

(d) WAT code

```
func,global.get,0
func,local.set,1
...
func,block,block,local.get,11
func,block,block,i32.eqz
func,block,block,br_if,0
...
func,return
```

(e) Extracted raw paths with DFS

$$\text{path\_vector} = \begin{matrix}[1, & 0, & 2, \\ 0, & ..., & 4]\end{matrix}_{3,352}$$

$$\text{path\_sequence} = \langle 1, 1\rangle \ \langle 3, 1\rangle \ ... \ \langle 3352, 2\rangle$$

$$\text{code\_embedding} = \begin{matrix}[0, & 0, & 0.8898107, \\ ..., & ..., & 0.60188985]\end{matrix}_{50}$$

(f) Code representations

Figure 2.1: This figure illustrates the files generated while C code to Wasm binaries and then extracting paths from WAT files.

spective WAT file using the wasm2wat module of WABT [43], a binary toolkit for Wasm. While there are some similarities between LLVM code and WAT code, they serve different purposes. LLVM files are an intermediate representation used during compilation, optimization, and code generation and are not intended for human consumption. In contrast, WAT files are commonly used for debugging and inspecting Wasm code, and they offer a tree-like structure of Wasm binaries that can be parsed into an AST for structural analysis, which is crucial for our analysis objectives.

Figure 2.1 illustrates different code files that we create until we extract the necessary AST information for further analysis. In subfigure 2.1a, we have a simple `sign()` function that outputs the sign of the integer input. There are three possible outputs: 0 if the input is 0; and 1 or -1 if the input is a positive or negative integer, respectively. Using Emscripten, we then compile the C function to a LLVM IR function, shown in subfigure 2.1b. We can select different optimization levels to create LLVM code—the code shown in the subfigure is generated using the -O3 option.

This LLVM code begins by fetching the value of the first argument of the function and placing it at the top of the stack. Then, it pushes the value of 31 onto the stack. This is done because the type `i32` has 32 bits, and in order to obtain the sign bit of the input argument, we need to right shift (using the `i32.shr_s` instruction) the input argument by 31 bits, so that the sign bit is in the least significant position. The possible values on the top of the stack after the `i32.shr_s` instruction are 0 for non-negative inputs and -1 for negative inputs. Next, the code pushes the constant value 1 onto the stack and checks whether the input is less than 1 using the `i32.lt_s` instruction. The `i32.select` instruction is then used to select either the constant value 1 or the value produced by the `i32.shr_s` instruction (which can be 0 or -1) based on the result of the comparison.

At the last stage of compilation, Emscripten generates Wasm binaries, shown in subfigure 2.1c. As discussed earlier, a Wasm binary is a stream of binary values. To ease analysis, we convert it to WAT format. As illustrated in subfigure 2.1d, the WAT files yield a tree-like representation of Wasm binaries. We can extract all paths within this tree using a Depth-First Search (subfigure 2.1e). Note that the shown extracted paths are raw paths. We carry out a refinement process to reduce the number of paths, and only keep paths that are more semantically valuable. We discuss the refinement process more in-depth in Section 3.

We extract the AST paths from all the Wasm binaries within our dataset. The total number of all the paths after refinement is 3,352. We represent each Wasm function using a feature vector, which we call the path vector. This vector has 3,352 dimensions. The i^{th} element of the vector corresponds to the number of times path i occurs in the

AST of that function. We build two code representations using our path vectors: (1) a path sequence that is a sequential representation of the path vector, but with normalized occurrence values, and (2) a code embedding similar to that of code2vec [3], which is a distributional representation of Wasm code using fewer (50) dimensions. Subfigure 2.1f shows the structure of our path vector, path sequence, and code embedding. In this subfigure, *path_vector* has value 2 at index 2. This means that the third path (path vector indices are zero-based) in our paths set has appeared twice in our AST. *path_sequence* yields a sequence representation of *path_vector*, after removing zero-valued paths and normalizing the values. *code_embedding* embeds *path_vector* into a 50-dimensional vector using a feedforward-neural network. Like code2vec [3], we use method names as our target property to create our embeddings. Yet, our approach differs from that of code2vec in two main ways. Firstly, we record root-to-leaf rather than leaf-to-leaf paths, as we believe that the bag-of-paths method would lead to a quadratic blowup in the number of dimensions for low-level code with no additional benefit. Secondly, we do not include AST terminals in our vectors, as these are typically integers associated with memory locations and can vary significantly even for highly similar functions.

In the next section we describe WasmWalker, the pipeline we developed for extracting paths from WAT files. Then we discuss empirical results about extracted paths, the decisions behind our path refinement process, and more details about our code representations.

Chapter 3

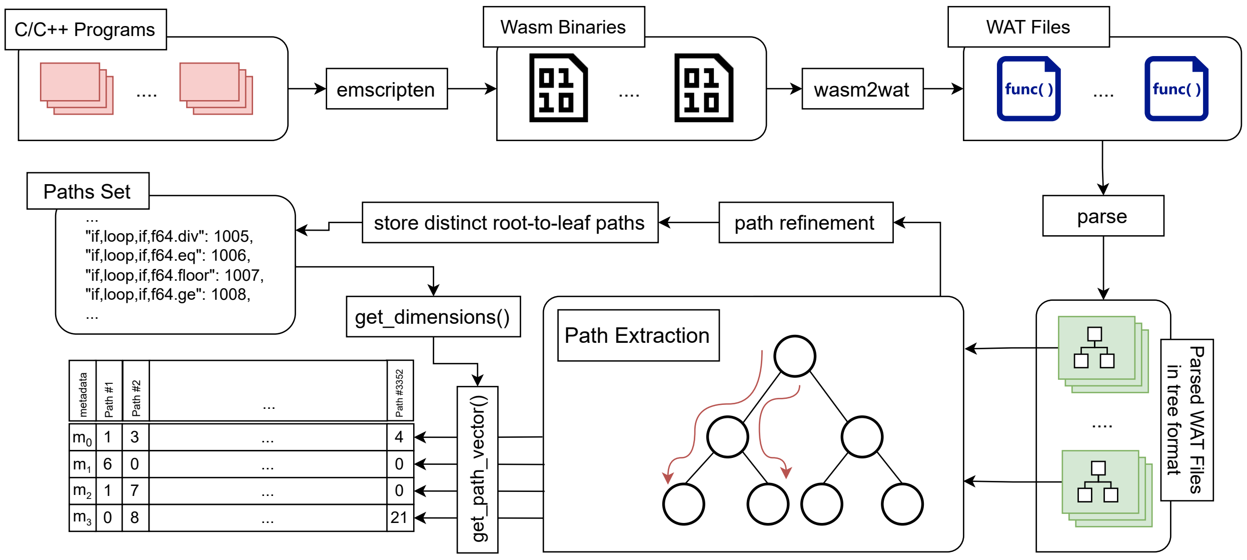
Methodology

Our objective is to encode information about a complete Wasm function to an interpretable fixed-sized vector. Similar to Feng et al. [15], a naive approach could be to build a feature vector for a function, where the features are general metadata about the function (e.g. number of variables). The problem with that approach is that it does not contain any information about code structure. We take a different approach: we exploit information encoded in the AST structure of Wasm programs, and hence what the program does, to form feature vectors.

Wasm binaries are generally being produced by tools that leverage LLVM compiler infrastructure: in addition to supporting WebAssembly, LLVM provides a high-performance and well-documented toolchain for producing WebAssembly binaries.

However, to study the AST form of the Wasm binaries, we need to convert the binaries to WebAssembly Text (WAT) format, which provides a more structured representation of Wasm binaries. For that goal, we use the wasm2wat module of WABT [43], a binary toolkit for Wasm. The output of wasm2wat is a textual representation of the WebAssembly module that closely mirrors the binary format, but with additional annotations and tree-like formatting to make the module structure more clear.

To process the nested structure of WAT files, we have to linearize the paths somehow so we can study their content. code2vec [3] extracts all leaf-to-leaf paths in the AST. That yields a $\Theta(n^2)$ space explosion given n terminals; for further discussion, see Section 6. Unlike code2vec, we instead record all root-to-leaf paths for two primary reasons: (1) to avoid the quadratic explosion, and (2) it is not clear that terminals in WAT ASTs contain useful information. The terminals are memory offset values and can easily change with slight modifications to the original program, even when the semantics stay the same. For



```
...
"if,loop,if,f64.div": 1005,
"if,loop,if,f64.eq": 1006,
"if,loop,if,f64.floor": 1007,
"if,loop,if,f64.ge": 1008,
...
```

metadata	Path #1	Path #2	...	Path #3352
m_0	1	3	...	4
m_1	6	0	...	0
m_2	1	7	...	0
m_3	0	8	...	21

Figure 3.1: Overview of the WasmWalker pipeline

instance, if in an alternative implementation of a subroutine, we use an extra variable, the compiler will allocate memory for variables based on their type and size. This may cause the memory locations of other variables to shift to make room for the new variable. This will not happen in high-level programming languages, where the terminals are usually variable names.

Figure 3.1 shows an overview of how we collect the common paths and store them in our paths set and leverage these paths to generate path vectors. This figure illustrates the use of emscripten and wasm2wat for transforming C/C++ code to WAT files. Then, we parse the WAT files and extract the raw paths. The raw paths are then refined (we present the refinement process in Section 3.1.2 and 3.1.3) and stored in an ordered paths set with unique numbers associated with each path.

Our pipeline, WasmWalker, traverses a WAT file's AST using Depth-First Search and extracts paths inside the subtree of a target function, as the WAT file can contain multiple functions. We refer to internal nodes in our AST as nonterminals and the leaves as terminals. The paths start with the nonterminal "`func`" and end with a terminal (see subfigure 2.1e). We drop the keyword "`func`", since it is the same for all paths, and also the terminal at the end. The resulting path is a sequence of nonterminals. Note that our use of "terminal" and "nonterminal" does not match their use in parsing.

WasmWalker is first applied to the training data to populate the paths set. The pipeline is then used again to generate path vectors, which contain the frequency of each path at its corresponding location in the set. The paths set is queried to obtain the index associated with a path. It should be noted that any path within the Wasm function that is not included in our paths set will be discarded. This is because we do not have a corresponding dimension in our path vector that is associated with that path. Each path vector also includes metadata, such as the function's name and high-level return type, collected from DWARF debugging information.

3.1 Common paths within WAT ASTs

We conducted an empirical study to find the common root-to-leaf paths that appear in WAT ASTs. Our dataset is comprised of training, validation and test data. To carry out a more accurate evaluation and avoid inflating the accuracy scores of our models, we extracted the common paths only from the training portion of our dataset. We extracted a total number of 807,972 raw paths. Since this number is so high, we reduce it through a path refinement process.

Three nonterminals can cause a nested structure in WAT ASTs: (1) `block`, (2) `if-else` conditionals, and (3) `loop`. Here, we describe how these instructions are used in Wasm code:

3.1.1 Nested Structure in WebAssembly Text (WAT) Format

In WebAssembly Text (WAT) format, three instructions can create nested structures: (1) `block`, (2) `if`, and (3) `loop`. These instructions are called *structured* instructions. They create nested sequences of instructions, called *blocks*, terminated with, or separated by, `end` or `else` pseudo-instructions. Defining an output type for structured instructions is optional: the output type specifies the type of value put on the operand stack after the execution of the block is finished. The structured instructions also carry implicit label that can be used in branching instructions (`br` and `br_if`) to jump between instructions. The implicit labels are introduced based on the depth of the nested structure. The innermost nested instruction get label 0, and increasing label numbers would refer to those further out. Next, we provide an example for each of structured instructions:

Wasm instruction: *block*

```
(func $add-sub-block (param i32 i32 i32) (result i32)
  (block $b1 (result i32)
    (i32.add
      (local.get 1)
      (local.get 2)
    )
    local.get 0
    i32.const 0
    i32.eq
    br_if $b1
    drop
    (i32.sub
      (local.get 1)
      (local.get 2)
    )
  )
)
```

Figure 3.2: An example of a control flow "block" in WebAssembly

Blocks bundle together a sequence of instructions that manifest a logical workflow, which can increase readability of code. Blocks can also be used for creating labels that can later be branched out of with branching instructions (`br`,`br_if`). Figure 3.2 illustrates an example of how `block` instruction is used in WebAssembly. Function `add-sub-block` takes in three parameters. The first parameters determines the operator (0 for subtraction and other numbers for addition). The next two parameters determine the operands. For instance, `add-sub-block(0,4,3)` computes $4-3$ and returns 1, and `add-sub-block(1,4,3)` computes $4+3$ and returns 7. The code first computes the addition and then checks whether the first parameter is 0. If it is 0 then it drops the top element in the stack (the addition result), and computes the subtraction. If the first parameter is not 0 then it branches out to the end of the block at line `br_if $b1`. The label `$b1` specifies which block we branch out from. If we had not used `block`, we would not have been able to branch out and avoid executing the subtraction portion of the code.

Wasm instruction: *if*

In Wasm code, `if` instruction is used to execute a block of code based on a boolean condition. Figure 3.2 shows an example of how `if` instruction is used in WebAssembly. Function `add-sub-if` is similar to `add-sub-block`, except it is implemented using `if` instruction instead of `block`. Based on the value of the first parameter (`local.get 0`) we would either execute the then or else block, which include addition and subtraction of operands, respectively.

```
(func $add-sub-if (param i32 i32 i32) (result i32)
  (if (result i32)
    (local.get 0)
    (then
      (i32.add
        (local.get 1)
        (local.get 2)
      )
    )
    (else
      (i32.sub
        (local.get 1)
        (local.get 2)
      )
    )
  )
)
```

Figure 3.3: An example of a control flow “if” in WebAssembly

```
(func $count-to-ten (param i32) (result i32)
  (local $counter i32)
  (loop $increment
    (local.set $counter (i32.add (local.get $counter) (i32.const 1)))
    (local.get 0)
    (local.get $counter)
    (i32.add)
    (i32.const 10)
    (i32.lt_s)
    (br_if $increment)
  )
  (local.get $counter)
)
```

Figure 3.4: An example of a control flow “loop” in WebAssembly

Wasm instruction: *loop*

`loop` instruction in Wasm is similar to `block`, except once the program branches out within the body of a loop, it jumps to the beginning of the loop, as opposed to `block`, where it jumps to the end. Figure 3.4 shows an example of how `loop` instruction is used in WebAssembly. Function `count-to-ten` returns the difference between 10 and the input using the `loop` instruction (we assume the input is less than 10).

Each raw path is a sequence of these nonterminals and then a final terminal which reflects one of the Wasm instructions. Two examples of extracted raw paths are:

- `if,loop,block,if,f32.const`

- `block,if,block,block,block,if,f64.ne`

We carry out two refinement steps to reduce the number of paths:

3.1.2 Collapsing repeating nonterminals

When a nonterminal appears multiple times in a row in a sequence of nonterminals, we collapse it into a single occurrence. For instance, the second example given above will be transformed into `block,if,block,if,f64.ne`. This process applies to nonterminals `block`, `if`, and `loop`. This change reduces the number of paths to 40,654.

3.1.3 Dropping blocks

One way of reducing paths even further is to drop all instances of one of the three nonterminals `block`, `if`, and `loop`. We decided to drop `block`, as we believe the other two nonterminals convey more semantic information. After this refinement step, the total number of paths plummets to 3,352. This number is low enough for our purposes in devising a code representation, therefore we choose this number as the number of dimensions for our feature vectors.

As a separate study, we also dropped `if` and `loop` subtrees. That makes our paths analogous to the instructions themselves, without any nested structure. This would lead to 185 paths, i.e. there would only be 185 instructions used in the training portion of our dataset.

Table 3.1 shows the most common paths. The most common path is `local.get`, which appears when the program retrieves a variable's value.

Also, Figure 3.5 demonstrates the accumulative number of paths after visiting the files of our dataset, sorted in alphanumerical order. As the figure depicts, the number of paths quickly increases to 1000. Then, it keeps on going up until it reaches a total of 3,352 paths at the end. Also, there are sudden jumps in the accumulative plot, suggesting that a small number of files can cause significant changes to our paths set. That indicates our data-driven path extraction approach requires a large dataset to produce reliable paths. To overcome this, we use the largest available WebAssembly dataset to obtain the most reliable paths set.

As discussed earlier, 3,352 is an acceptable number to set as the number of dimensions for our feature vectors. Each vector represents a function in a Wasm program and each

Table 3.1: Top most common root-to-leaf paths

Rank	Path	#	%
1	`local.get`	31,886,923	19.67
2	`loop,local.get`	17,887,693	11.03
3	`local.set`	17,397,255	10.73
4	`i32.const`	11,478,749	7.08
5	`loop,local.set`	9,959,227	6.14
6	`i32.load`	5,695,009	3.51
7	`loop,i32.const`	5,644,941	3.48
8	`i32.add`	3,524,522	2.17
9	`loop,i32.load`	3,514,771	2.16
10	`i32.store`	3,501,192	2.16

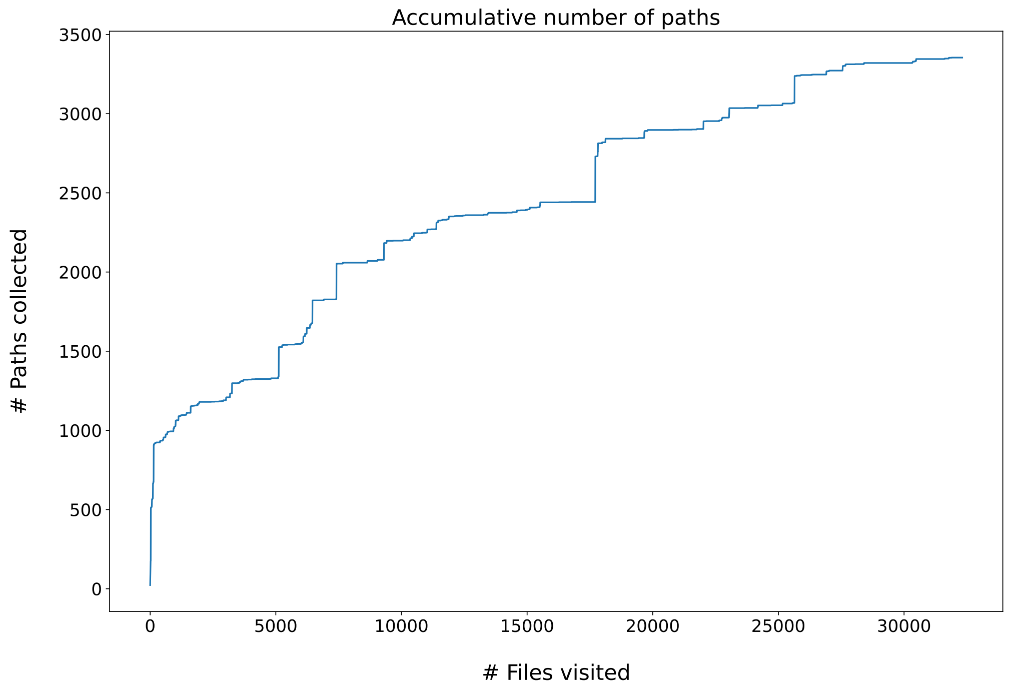


Figure 3.5: Accumulative number of paths

element in the vector is an integer indicating the number of times the path associated with that element appears in the Wasm function's AST.

3.2 Code representation

Using our path vectors, we devise two different code representations. Our goal for devising these code representations is to use them as inputs to deep learning models—Section 4 discusses applications extensively. In the following sections, we describe each of these representations in detail.

3.2.1 Path Sequence

We can use a simple function s to transform a path vector v to a sequence $s(v)$ which contains all the necessary AST information. We define the function s as follows:

$$s : \mathbb{N}^{3352} \mapsto \{\langle n, m\rangle\}^*, 1 \leq n \leq 3352, 1 \leq m \leq D,$$

$$s(v) := \{\langle e.index, \lceil \frac{eD}{\sum\limits_{e \in v} e} \rceil\rangle \mid e \in v, e \neq 0\}.$$

As shown in the equation above, each non-zero element in v maps to a tuple $\langle n, m\rangle$. The first argument n is the path's index in our indexed paths set, and the second argument m shows the number of times the n^{th} path appeared in the AST. In tuple $\langle n, m\rangle$, a high value for m indicates that the n^{th} path appeared many times in the AST. Note that we normalize the values of m between 1 and D. We empirically found out that setting $D = 30$ is enough for our purposes. A high value for D can create redundant symbols that could have formed a single symbol. To give an example a vector like $v = [1, 0, 204, ..., 2]$ can be transformed to $s(v) = \langle 1, 1\rangle\ \langle 3, 30\rangle\ ...\ \langle 3352, 1\rangle$. It is worth mentioning that the average number of unique paths that exist in a Wasm function is 21.5. That is, a path sequence has on average around 21 tuples.

3.2.2 Code Embedding

An embedding is a mapping from objects to vectors of real numbers. It makes it possible to work with textual data in a mathematical model. It also has the advantage that fundamentally discrete data (words) is transformed into a continuous vector space. In natural languages, embeddings can be created at different granularity levels, such as words, sentences, or documents. Similarly, in programming languages, we can create embeddings for program tokens, statements, or functions [12].

One of the goals of embedding is to make similar objects have similar vectors. For instance, in the Skip-gram model, the words that are used in similar contexts will have similar embeddings. This similarity, however, must be defined based on our objective. For example, two functions can be similar in "method name" but not similar in "containing security vulnerability"

Neural networks are often used for creating embeddings because of scalability and the fact that they are capable of learning relationships between large numbers of words, making them well-suited for processing large-scale text datasets. Also, neural networks can model complex, non-linear relationships between words, allowing them to capture subtle relationships and associations.

Similar to word2vec and code2vec, we use simple feedforward neural networks to create code embeddings. However, unlike word2vec, we do not use the hot-one encoding, because hot-one encoding provides no measure of similarity between similar objects (e.g. words like "apple" and "orange"). Our inital vectors encode AST paths, so similar objects might share similar paths. We use a similar network to the one used in code2vec to create our code embeddings. The only difference is that our method does not require having an attention weights vector, as each element in our input vector represents exactly one path. In code2vec, by contrast, each leaf-to-leaf path is encoded with a few hundred entries of the input vector. The attention weights vector is a vector in code2vec's neural architecture that contains a weight for each leaf-to-leaf path that corresponds to the path's impact in determining the output label.

Chapter 4

Evaluation

In this section, we evaluate our method on two tasks: (1) prediction of method names and (2) recovery of precise return types. We used the same dataset for both evaluations. First, we provide an overview of the general characteristics of the dataset. Then we introduce the models that we created and used for evaluation. Finally, we present our results for method name prediction in Section 4.3 and for precise return type recovery in Section 4.4.

4.1 Dataset

We used the same dataset introduced in SnowWhite [27] for three main reasons: (1) the dataset has been filtered and each data point is linked to a sequence of instructions, which are refined for more accurate type recovery. This provides us with a good opportunity to examine if our insights regarding common AST paths and our path sequence can enhance the accuracy results. (2) The dataset contains DWARF debugging information, including function names and precise types, properties that our experiments are designed to predict. (3) To the best of our knowledge, this is the largest dataset for WebAssembly and is significantly larger than the datasets used in previous works [16, 18, 26, 21].

The dataset used in this study contains 6.3 million samples of WebAssembly code and type information obtained from 300,905 object files that were compiled from 4,081 C and C++ packages for the Ubuntu operating system. To prevent artificial inflation of results, the SnowWhite dataset has undergone some pre-processing steps, such as deduplication and discarding data points where the number of return types in WebAssembly does not match the number in C/C++ code. In addition to that, to use the wasm2wat module as part of deriving ASTs, we also discarded datapoints that take more than 30 seconds to translate to WAT files. The dataset has already been divided, based on the number of packages, into three portions, with 96% for training, 2% for early stopping and evaluating hyperparameters, and 2% reserved for a held-out test set. For a more accurate comparison with SnowWhite, we adopted the same dataset divisions and conducted a hold-out cross-validation to evaluate our approach. We did not use more complex cross-validation techniques, such as k-fold, due to time and space limitations. More specifically, a 10-fold cross validation would require near 100GB of storage and approximately 120 hours of training time for each configuration.

4.2 Models

We trained five different models to assess whether AST paths enable us to create better models for predicting properties of Wasm programs. To carry out a more accurate comparison with SnowWhite, we use the same sequence-to-sequence model as SnowWhite for predicting the types of function parameters and return values in code. The only difference between our approach and SnowWhite is that we did not use subword tokenization because it had zero impact on our model's performance. SnowWhite uses a bidirectional LSTM model [40] with global attention [5] and dropout [41] for regularization. The model is optimized with backpropagation-through-time gradient descent using the Adam optimizer [22]. The SnowWhite authors chose hyperparameters including hidden vector dimension, number of layers, learning rate, dropout rate, and embedding dimension through experimentation. To conduct an accurate comparison, we use the same hyperparameters. The authors also experimented with the Transformer architecture but found that the LSTM model was more effective. The model has a total of 5.5 million learnable parameters.

We use the same sequence-to-sequence architecture for our five models, with each model receiving a different input. The first model (seq2seq-INP) receives a concatenation of the last 20 Instructions and Nested Path sequences (we will justify our choice of the last 20 instructions later). The second model (seq2seq-ISP) is similar to the first model, except the path sequence does not reflect the 3,352 nested paths but 185 Simpler Paths after dropping `if` and `loop` subtrees, as discussed in Section 3. The third model (seq2seq-I) only receives the last 20 Instructions sequences, which is the model used in SnowWhite. The fourth model (seq2seq-NP) only receives the path sequence, reflecting the 3,352 Nested Paths. The fifth model (seq2seq-SP) is similar to the fourth model, but it uses the 185 Simpler Paths instead of the nested ones.

We chose these five models to control the type and amount of information being input to our models and to enable accurate comparisons between them. The seq2seq-INP model receives the most information, as it takes in both the instruction and path sequences, with the path sequence being nested.

Like SnowWhite, we used OpenNMT [23] to train our models. We trained our models using Google Colaboratory Pro+. Our training setup was an Intel(R) Xeon(R) CPU with 3.7GHz clock speed and 16 cores, 85GB of RAM, and an NVIDIA P100 GPU with dedicated memory. Training and testing each model took around 15 minutes.

4.3 Method name prediction

First, to avoid manual inspection after model prediction, we carry out a preprocessing step to simplify method names. This preprocessing step includes: (1) removing generic types that a method name may include, (2) removing initial underscores that usually indicate private/protected methods, (3) converting name letters to lowercase. The preprocessing step also prevents artificial inflation of our accuracy scores and enables us to conduct a more meaningful analysis of our results. Remember that these preprocessing steps are specifically designed for C/C++ naming conventions, which are the relevant ones for our training set. Other languages might require different steps, which are straightforward to carry out. Next, we create different datasets parameterized by m, the minimum number of datapoints associated with a method name for it to be included in the dataset. For instance, if a dataset is created with $m = 50$, that means each method name in the dataset has at least 50 datapoints associated with itself. We create four datasets for $m = 10/20/50/100$. The datasets include 345,417 / 279,262 / 223,903 / 192,905 datapoints and 7,560 / 2,768 / 874 / 412 unique method names, respectively.

Table 4.1 presents the prediction accuracy results after training each model on each dataset. **As shown in the table, our seq2seq-INP model offers the best accuracy scores across the board.** Moreover, both seq2seq-INP and seq2seq-ISP models offer better accuracy than seq2seq-I, which was the model used in SnowWhite.

Moving on to the models that were trained without being given instruction sequences, seq2seq-NP and seq2seq-SP usually resulted in less accuracy than seq2seq-I. Under some circumstances, omitting the instruction order does not affect the model accuracy.

Table 4.1: Method name prediction accuracy results: The seq2seq-INP model provides the highest accuracy scores overall. Additionally, seq2seq-INP and seq2seq-ISP models outperform seq2seq-I, which was used in SnowWhite. Models without instruction sequences, seq2seq-NP and seq2seq-SP, generally yield lower accuracy compared to seq2seq-I.

	m=10		m=20		m=50		m=100	
	top-1 acc.	top-5 acc.	top-1 acc.	top-5 acc.	top-1 acc.	top-5 acc.	top-1 acc.	top-5 acc.
seq2seq-INP	**75.97%**	**92.79%**	**77.03%**	**94.67%**	**76.82%**	**96.01%**	**76.62%**	**96.68%**
seq2seq-ISP	74.80%	92.68%	74.71%	93.59%	75.95%	95.59%	75.76%	96.07%
seq2seq-I	74.19%	90.72%	73.68%	91.41%	71.46%	84.70%	74.25%	92.48%
seq2seq-NP	71.54%	89.25%	72.11%	91.51%	71.53%	79.53%	72.75%	92.65%
seq2seq-SP	71.13%	88.75%	72.70%	86.61%	70.96%	79.64%	64.21%	86.61%

4.3.1 Code embedding

As a separate study, we created code embeddings for method names. Our goal is to depict similar method names close to each other, providing evidence that the embeddings have the correct understanding of code semantics. Figure 4.1 shows a 2D visualization of our embeddings using t-SNE [30], a technique used for dimensionality reduction and visualization of high-dimensional datasets. The point associated with each method name shows the location of the average of all embeddings with that name. This visualization effectively conveys the relationships between method names and their semantics.

To produce these embeddings, we used a feedforward network with the following characteristics: four hidden layers, which employ dropout and L2 kernel regularization to prevent overfitting; ReLU and softmax activation functions in the hidden and output layers, respectively; and the Adam optimizer. This model takes path vectors as input. The last hidden layer has 50 nodes that contain our embedding values. We chose $m = 10$.

With these embeddings, we are able to cross-check correlations between method names—names have independent semantic meanings. By examining these correlations, we aim to further validate our model's effectiveness in capturing underlying semantics. This additional analysis shows the quality of the embeddings we generate. We discuss some method name similarities in the figure:

A Closer Look: "lzwdecodecompat" and "logluvdecode32" both decode compressed data. "lzwdecodecompat" decodes LZW (Lempel-Ziv-Welch) data compression, which is a lossless compression algorithm commonly used in the past for images, text, and other types of data. "logluvdecode32", on the other hand, decodes LogLuv-encoded image data.

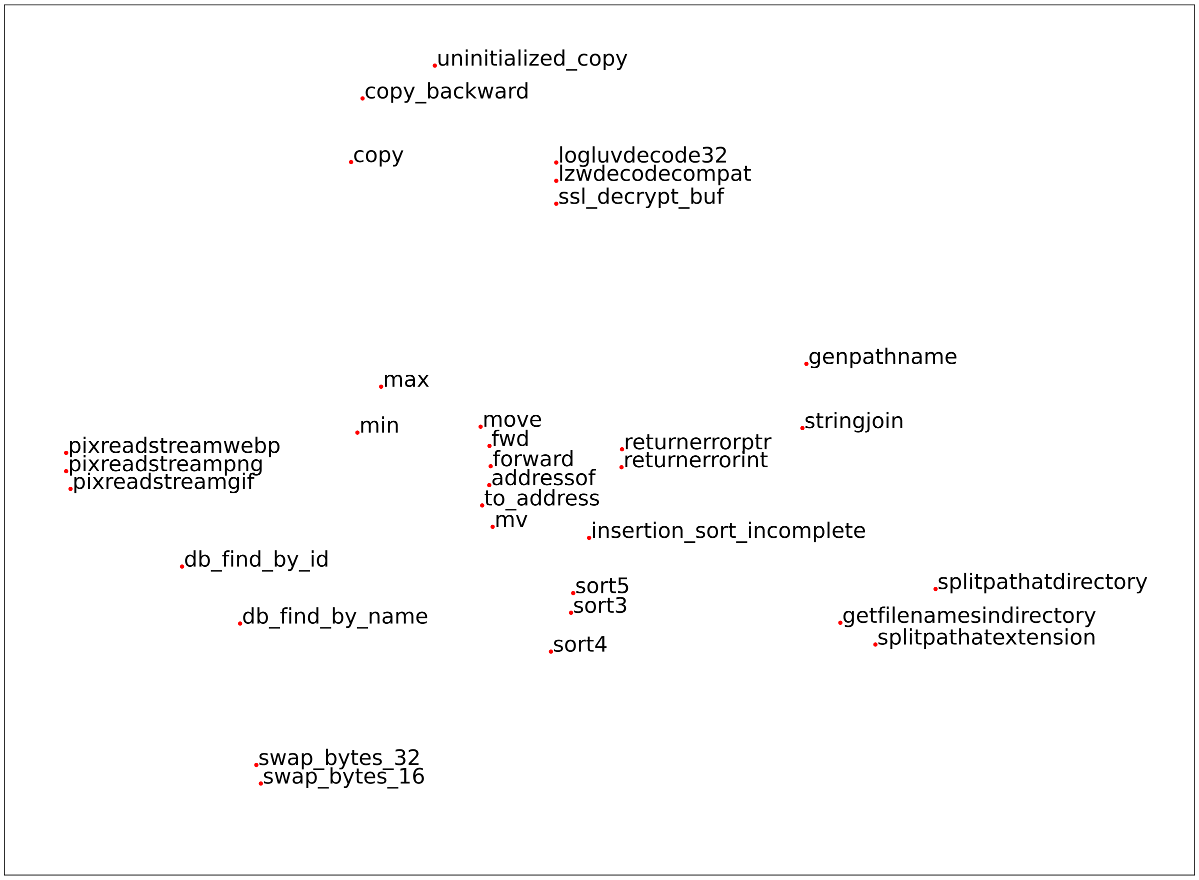


Figure 4.1: A t-SNE 2D plot of code embeddings generated using our proposed code embedding approach. The plot shows the spreading of method names in the 2D plane, with similar method names closer to each other, highlighting the effectiveness of our embedding approach in capturing the semantic similarities among Wasm program methods. To prevent label overlapping, we manually adjusted the vertical position of points by up to 3.1%. To see the original plot, refer to Appendix A

.

LogLuv is a non-linear color space that was developed for representing high-dynamic-range (HDR) image data, and it is used in many professional graphics applications. While these methods are used for different types of data compression, they are similar in that they are both used for decoding compressed data.

The figure also shows "ssl_decrypt_buf" close to "lzwdecodecompat" and "logluvdecode32". "ssl_decrypt_buf" is selected from a package related to SSL/TLS protocol, a cryptographic protocol that is used to provide secure communication over the internet. As the name suggests, "ssl_decrypt_buf" refers to decrypting SSL traffic, which is similar to "lzwdecodecompat" and "logluvdecode32", as all of these methods aim to retrieve the original encoding of a data stream that has been transformed to a different encoding.

Other method name pairs are "db_find_by_name"/"db_find_by_id", "min"/"max", and "swap_bytes_16"/"swap_bytes_32".

4.4 Recovering precise return types

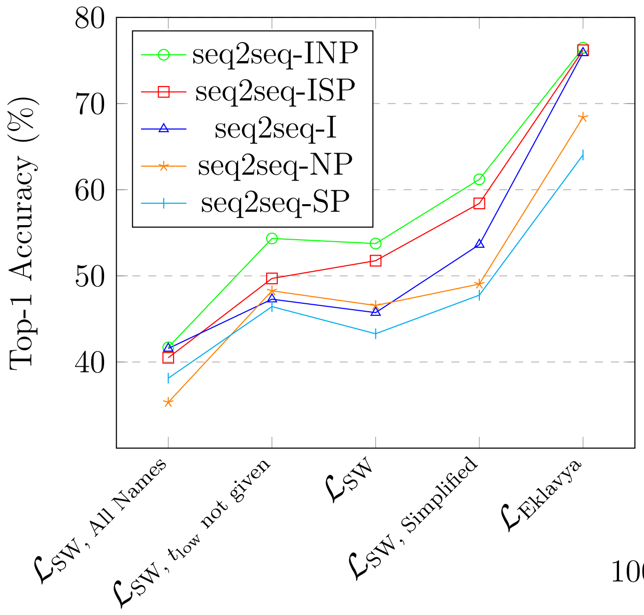
As we described earlier, Lehmann et al. [27] proposed SnowWhite, the first tool for recovering precise types from Wasm binaries. The WebAssembly type system only includes four primitive types: "i32", "i64", "f32", and "f64". SnowWhite successfully showed that it is possible to recover precise high-level C/C++ types from Wasm primitive types using a seq2seq model that gets a portion of Wasm instructions as input. Their work used two different models to recover precise types for (1) method parameters, and (2) return values. Our initial assumption was that providing the model with AST knowledge would improve accuracy. Since the primary objective of this paper is not precise type recovery, but rather offering a novel code representation, we only evaluate recovering precise return types and not parameter types.

For evaluating SnowWhite, the authors defined a high-level type language that governed the to-be-predicted type sequences. Additionally, they defined five different type variants and, based on those variants, carried out five different evaluations: (1) $\mathcal{L}_{\text{SW}}$: The default version that applies filtering on types (e.g. omiting static arguments of instructions that are likely unhelpful and unnecessarily), (2) $\mathcal{L}_{\text{SW, All Names}}$: Disables all filters, (3) $\mathcal{L}_{\text{SW, Simplified}}$: Removes names, constness and the distinction between classes and structs from $\mathcal{L}_{\text{SW}}$, (4) $\mathcal{L}_{\text{SW, } t_{\text{low}} \text{ not given}}$: Similar to $\mathcal{L}_{\text{SW}}$, except the low-level WebAssembly type is not given, (5) $\mathcal{L}_{\text{Eklavya}}$: This variant is based on a prior work [13] that has only seven primitive types.

Figure 4.2 compares the accuracy scores of our five models (to find the actual accuracy numbers, see Appendix A). **Like we found with method name prediction, our concatenation of instructions and path sequences resulted in the highest accuracy scores both in Top-1 and Top-5 accuracy plots.** More specifically, seq2seq-INP resulted in better accuracy than seq2seq-I by 8.02% top-1 accuracy and 7.92% top-5 accuracy in $\mathcal{L}_{\text{SW}}$. Similarly, seq2seq-ISP offers better accuracy than seq2seq-I by 6.02% top-1 accuracy and 6.14% top-5 accuracy in $\mathcal{L}_{\text{SW}}$. The accuracy scores displayed for seq2seq-I are close to those reported by the SnowWhite. Minor variations may exist due to missing training data points caused by the 30-second timeout during WAT generation.

As with method name prediction, seq2seq-NP, a model that was not given the instruction sequences as input, offers better accuracy than seq2seq-I in top-1 accuracy in $\mathcal{L}_{\text{SW, }t_{\text{low}}\text{ not given}}$ and $\mathcal{L}_{\text{SW}}$, and in top-5 accuracy in $\mathcal{L}_{\text{SW, All Names}}$ and $\mathcal{L}_{\text{SW, }t_{\text{low}}\text{ not given}}$.

For both method name prediction and precise return type recovery, nested paths models (seq2seq-INP and -NP) give better accuracy than their simple paths counterparts (seq2seq-ISP and -SP).



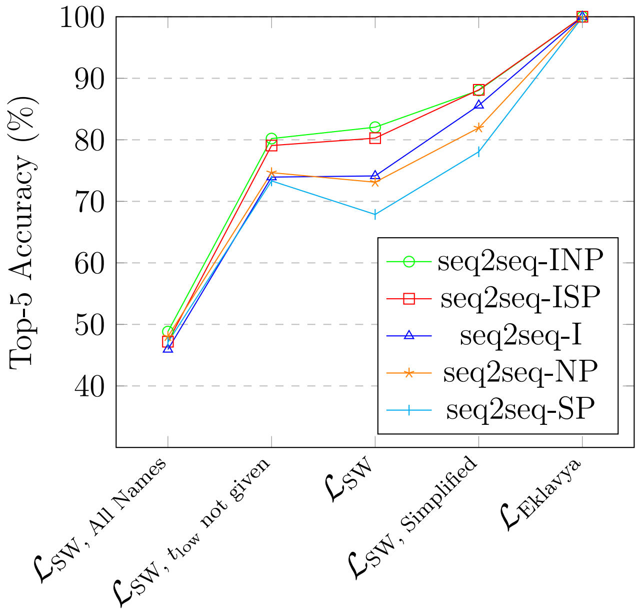


Figure 4.2: Model Accuracies for precise return type recovery

Chapter 5

Discussion

Our hypothesis was that information about AST paths in Wasm can improve the effectiveness of our models. We tested this hypothesis over two different high-level tasks: (1) method name prediction, and (2) recovering precise return types. There are fundamental differences in tool requirements for these two tasks.

Our two tasks benefit from different types of information about the program under analysis. Method name prediction usually requires analysis of the program's control flow, while data flow analysis might be more helpful for recovering precise high-level types. Both tasks can be helpful for reverse engineering, detecting security vulnerabilities, and understanding the internal structures used inside the program.

Another key difference between these tasks is that method name prediction requires all of the binary code of the target function, i.e. a method name is associated with the whole method body. On the other hand, recovering precise return types can be done with a selected portion of the binary code of the target function.

We showed that both tasks benefit from knowledge of the AST. More specifically, we observed that using the seq2seq-INP model, which includes nested AST paths information in the code representation, we achieve a higher accuracy compared to the seq2seq-I model, which does not. This improvement in accuracy was observed across a diverse set of programs in our dataset, containing Ubuntu packages with a wide range of software, including networking programs and high-performance security and cryptographic packages. **Therefore, incorporating AST paths information can be beneficial for both method name prediction and recovering precise high-level types, and this holds true for a wide variety of programs.**

We incorporated concrete knowledge of Wasm binaries into the seq2seq-INP, -ISP and -I models by using the last 20 instructions sequence. There were two main reasons for choosing 20. Firstly, this was the approach used in previous work (SnowWhite) for recovering precise return types: the assumption was that the last 20 instructions are most closely related to the return type. Secondly, we needed to limit the number of instructions used in our models: using all instructions can result in overfitting and long training times. Giving more instructions to seq2seq models would lead to a linear increase in the additional space ($\Theta(n)$) proportional to the number of instructions n. The benefits of adding path sequences is independent of the selected number of instructions: our path sequence imposes a constant additional space complexity ($\Theta(1)$). It's important to note that the average number of unique paths in a path sequence is 21.5, and that the path sequence represents an entire function, not just the last 20 instructions. We argue that incorporating structured AST knowledge is an economical and efficient method of enhancing our analysis models with more valuable information versus other linear approaches. Although we designed WasmWalker for WebAssembly, the fundamental methodology is flexible and would apply to related technologies.

It is uncertain whether providing all paths to our model would be universally beneficial, as it is possible that including certain paths might not improve model performance, or may even reduce its effectiveness for certain objective functions. However, for the two tasks studied in our research, we found that including all paths led to better models. It is worth noting that our path extraction method did not include concrete terminals, as we aimed to reduce the number of paths and keep them simple. Additionally, the use of terminals can vary significantly for semantically similar programs. However, further investigation of how the terminals, mainly memory locations, change in Wasm binaries with slight variations in the high-level source code could improve our method. Such a study would provide more insight into how our model can leverage terminal information in its path extraction process.

5.1 Threats to Validity

The potential threats to the validity of our research must be taken into account to assess the efficacy of our strategy.

An overarching threat is that our dataset is not representative. The number of AST paths that we derived and used as the size of our feature vectors was based on an empirical study. Arbitrary Wasm programs may contain paths that are not included in our paths set. We mitigate this threat by using a large dataset. And, given a new dataset, it is a straightforward task to recompute paths; while the number 3,352 might change, we do not expect this to affect the viability of our representations.

Another threat to the validity of our approach is the composition of our dataset. Specifically, our dataset consists of Wasm binaries that were compiled from C/C++ programs using emscripten. While we believe that all Wasm compilers that use LLVM as part of their compilation pipeline (as does emscripten) would create the same paths, it is unclear if compilers that use a different process for creating binaries would yield the same set of paths. However, it should be noted that the set of instructions that can be used in Wasm is limited by the language, and the order of instructions in a nested structure is restricted by the three structured instructions (`block`, `if`, and `loop`). Therefore, we have reason to believe that even if a different compiler toolchain were employed, the resulting paths would be similar.

Finally, WasmWalker embodies a dependency on the current version of WebAssembly. If WebAssembly gains new features causing changes in AST strucures, we would need to recollect paths and update our paths set accordingly—a routine non-research task.

Chapter 6

Related Works

The analysis of binary code is a widely used method for studying closed-source programs. This approach has a wide range of applications, including identifying plagiarism, predicting performance, detecting malware, and discovering vulnerabilities. Researchers have developed many techniques for analyzing binary code.

6.1 Program analysis for WebAssembly binaries

Lehmann et al. [27] proposed SnowWhite, a neural approach for recovering precise high-level parameter and return types of WebAssembly functions. To represent high-level types, SnowWhite uses an expressive type language that describes a large number of complex types, and builds on the success of neural sequence-to-sequence models for sequence prediction. SnowWhite is the first work that evaluates its performance on a large-scale dataset of 6.3 million type samples extracted from 300,905 WebAssembly object files. The results highlight that SnowWhite's type language is expressive and accurately matches a significant portion of parameter and return types with the top-1 and top-5 predictions. To conduct a reliable evaluation, SnowWhite defines five different dataset variants, each of which with a different target type precision. Our work uses the same dataset with the same variants. However, for our first experiment (method name prediction), we changed the target to method names. To have a meaningful and accuracte comparison with SnowWhite, we used the same sequence-to-sequence models that were used in this study. We also took advantage of the instruction sequences that our work provides to create a more effective code representation for WebAssembly functions.

Stievenart et al. [42] introduced Wassail, a static analysis library designed for WebAssembly. The toolkit offers a range of reusable modules that can be utilized to develop static analysis tools for Wasm, and is characterized by its hybrid functionality. These modules can be used for a variety of purposes, from lightweight tasks such as counting instructions, to more sophisticated tasks like taint analysis. Wassail takes a Wasm binary and transforms it into an intermediate file format, which makes it easier to access the necessary information for static analysis. This intermediate file can then be read by different modules, including the call graph module, which is designed to construct the call graph of a Wasm binary based on the `call` and `call_indirect` instructions. Wassail also includes a module for counting instructions, which provides a lightweight way to extract the frequency of instructions, and a control flow graph module, which can be used to construct the control flow graph of a Wasm binary. The control flow graph module enables heavyweight static analysis tasks, such as data flow analysis and taint analysis, to be performed more easily. While Wassail offers a suite of tools for static analysis of WebAssembly binaries, we did not utilize it for our purposes as we simply needed to parse the abstract syntax tree of WAT files.

Bastys et al. [6] presented SecWasm, a method for enforcing information-flow security in Wasm. Wasm provides security through the same-origin policy and a sandboxed execution environment for memory safety. Its structured control flow ensures control-flow integrity by preventing malicious jumps to arbitrary locations. However, this does not guarantee a secure information flow. SecWasm uses Wasm's structured control flow and type system to provide a general approach for controlling information flow, leveraging Wasm's operand stack to prevent sensitive inputs from leaking to public outputs. While our project has a completely different goal from SecWasm, both projects are similar in terms of using Wasm's nested code structure to achieve better results.

6.2 Program analysis for other binaries

Feng et al. [15] proposed Genius, a graph embedding pipeline. Genius splits native code into a collection of basic blocks and creates an attributed control flow graph (ACFG); ACFG nodes are basic blocks. These nodes contain two types of attributes: block-level attributes (e.g. the number of instructions) and inter-block attributes (e.g. the number of offspring). To transform an ACFG to an embedding, Genius trains a codebook using a clustering algorithm. Then it leverages a bipartite graph matching algorithm to measure the similarity between an ACFG and the representative of a cluster. These similarity measures then map to fixed-sized feature vectors.

The main drawback of the Genius work is that generating the codebook is costly, and the quality of the codebook is limited by the size of the training dataset. Xu et. al [46] improved on Genius with neural network-based embedding generation instead of bipartite graph matching. They tested their method on three different datasets, including the one from Genius, and achieved better accuracy and training time. Our work is similar to these two studies: we also transform block-level information into fixed-sized vectors. Like Xu et. al, we use a neural approach that yields a reduced training time compared to traditional methods like Genius codebooks. However, there are key differences. First, their ACFG graph traversal algorithm for generating embeddings does not involve path extraction, whereas our method is based on gathering paths from ASTs. Second, their dataset did not include Wasm binaries.

To capture code semantics, Ben-Nun et al. [8] follow a hypothesis for computer programs akin to the linguistic Distributional Hypothesis [38]. This hypothesis states that statements used in the same context often have similar semantics. They thus devise a representation for LLVM IR statements called conteXtual Flow Graphs (XFG) that incorporates the relative data- and control-flow of a statement. To build a XFG, they use LLVM IR statements in Static Single Assignment format. After preprocessing, they used the skip-gram model [32] to derive embeddings. The authors evaluated the derived XFG embeddings in three high-level classification and prediction tasks and found that they outperformed all manually extracted features. The results were comparable to or better than results from two inherently different specialized DNN solutions.

Similarly, Redmond et al. [39] introduced an instruction embedding model. They created a joint model combining the context information found in instruction sequences within the same architecture and the semantic similarities found in instruction sequence pairs from different architectures. They used word2vec's CBOW algorithm and evaluated the model on three tasks: (1) determining instruction similarity within the same architecture, (2) determining instruction similarity across different architectures, and (3) comparing basic block similarity across different architectures.

Our embeddings, unlike prior work, maps functions to vectors, rather than instructions to vectors. Furthermore, our results on Wasm ASTs provide not just an embedding, but also an interpretable sequence representation of a Wasm function, which is not offered by the prior studies. Our path sequence is interpretable as we draw each path from an indexed paths set (shown in Figure 3.1).

6.3 Code embedding

Alon et al. [3] proposed code2vec as a method to embed code in high-level programming languages into fixed-sized vectors. Their approach involved extracting paths from the AST, focusing specifically on leaf-to-leaf paths. They asserted that these paths contain more semantic information, as they include all the information between two terminals such as variable names in the AST. As discussed in Section 3, we believe that leaf-to-leaf paths do not contain additional useful semantic information in our context. They assigned fixed-sized vectors to paths and program tokens, which were then fed into a neural network to learn how to combine the vectors into a single embedding. The authors trained the network with an attention mechanism for predicting method names in Java programs and achieved promising results. However, code2vec faces challenges in creating input for the network, such as the quadratic explosion of leaf-to-leaf paths and the variable number of paths in a program. These challenges can be addressed by adding padding or using RNNs to obtain a compressed representation of paths. In contrast, our method avoids these challenges as it is based on the limited number of refined paths in WAT ASTs and the fact that we use root-to-leaf paths to avoid the quadratic explosion.

Chapter 7

Conclusion

In this study, using our pipeline, WasmWalker, we conducted the first empirical investigation of the common paths within the AST of Wasm programs over a large dataset of Ubuntu 18.04 packages. These paths are obtained from Wasm programs in WebAssembly Text (WAT) format, which offers a nested structured representation of the Wasm binary. We initially extracted over 800,000 raw paths from all the Wasm functions within our dataset. To reduce the number of paths, we carried out two refinement steps, with the goal of preserving the semantics of the program as much as possible. After refining the collected paths, our analysis revealed that there are only 3,352 refined root-to-leaf paths within the ASTs of Wasm programs. We represent each Wasm function using a feature vector with 3,352 dimensions, which we call the path vector. The i^{th} element of the path vector corresponds to the number of times path i occurs in the AST of a Wasm file associated with that path vector.

Our novel contribution lies in the development of two code representations based on our empirical findings about these 3,352 root-to-leaf paths that facilitate program analysis over WebAssembly binaries, providing researchers with a valuable tool for this purpose. The first code representation is a path sequence that shows the paths inside a Wasm function and the number of times they occurred in the AST. Our path sequence is essentially the sequence of nonzero values in our path vector after normalizing the values. The second code representation is a code embedding that encapsulates all the information within a Wasm function into a compact vector of real numbers. To obtain the embeddings we used off-the-shelf embedding techniques, providing our path vectors as input.

To demonstrate the utility of our proposed code representations for WebAssembly program analysis, we evaluated them using five sequence-to-sequence deep learning models across two tasks: (1) method name prediction and (2) recovering precise return types. When concatenated with a portion of actual instructions, our path sequence representation led to improved prediction accuracy in both of these tasks. Specifically, it resulted in up to 5.36% (11.31%) improvement in Top-1 (Top-5) accuracy in method name prediction and 8.02% (7.92%) improvement in recovering precise return types, compared to the state-of-the-art. Also, our code embedding approach successfully provides similar embeddings for method names that have semantically similar method bodies. Although we tailored WasmWalker specifically for WebAssembly, the underlying approach can be readily adapted to other technologies.

Our experiments were conducted over two inherently-different tasks, yet we successfully showed that knowledge about AST paths in Wasm files can improve the effectiveness of the deep learning models. Our novel code representations that were used as input to our deep learning models enable us to surpass the former state-of-the-art, SnowWhite, where the inputs were a sequence of instructions, not considering the nested structure of Wasm files. In contrast to SnowWhite, we took advantage of the nested structure that Wasm offers to add knowledge about AST paths to the selected instructions. The result was better accuracy with only a constant additional cost.

References

[1] Emscripten: A C/C++ to JavaScript compiler. `https://emscripten.org/`, 2023. Accessed: March 3, 2023.

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th joint meeting on Foundations of Software Engineering*, pages 38–49, 2015.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[4] Amy Aumpansub and Zhen Huang. Learning-based vulnerability detection in binary code. In *2022 14th International Conference on Machine Learning and Computing (ICMLC)*, pages 266–271, 2022.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[6] Iulia Bastys, Maximilian Algehed, Alexander Sjösten, and Andrei Sabelfeld. SecWasm: Information flow control for webassembly. In *Static Analysis: 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5–7, 2022, Proceedings*, pages 74–103. Springer, 2022.

[7] Canan Batur Şahin and Laith Abualigah. A novel deep learning-based feature selection model for improving the static analysis of vulnerability detection. *Neural Computing and Applications*, 33(20):14049–14067, 2021.

[8] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. *Advances in Neural Information Processing Systems*, 31, 2018.

[9] Tiago Brito, Pedro Lopes, Nuno Santos, and José Fragoso Santos. Wasmati: An efficient static vulnerability scanner for WebAssembly. *Computers & Security*, 118:102745, 2022.

[10] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[11] Lutz Büch and Artur Andrzejak. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–104. IEEE, 2019.

[12] Zimin Chen and Martin Monperrus. A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061*, 2019.

[13] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *USENIX Security Symposium*, pages 99–116, 2017.

[14] Jacob Devlin, Jonathan Uesato, Rishabh Singh, and Pushmeet Kohli. Semantic code repair using neuro-symbolic transformation networks. *arXiv preprint arXiv:1710.11054*, 2017.

[15] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491, 2016.

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

[17] Quinn Hanam, Fernando S de M Brito, and Ali Mesbah. Discovering bug patterns in JavaScript. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 144–156, 2016.

[18] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. An empirical study of real-world WebAssembly binaries: Security, languages, use cases. In *Proceedings of the Web Conference 2021*, pages 2696–2708, 2021.

[19] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[20] The LLVM Compiler Infrastructure. LLVM: a compilation framework. `https://llvm.org/`, 2022. [Online; accessed 3 March 2023].

[21] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *USENIX Annual Technical Conference*, pages 107–120, 2019.

[22] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[23] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M Rush. Opennmt: Open-source toolkit for neural machine translation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 67–72, 2017.

[24] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1714–1730, 2018.

[25] Anders Krogh and John A Hertz. A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pages 950–957, 1992.

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

[27] Daniel Lehmann and Michael Pradel. Finding the Dwarf: Recovering precise types from WebAssembly binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 410–425, 2022.

[28] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. CCLearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260. IEEE, 2017.

[29] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[30] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[31] C Manjula and Lilly Florence. Deep neural network based hybrid approach for software defect prediction using software metrics. *Cluster Computing*, 22(Suppl 4):9847–9863, 2019.

[32] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[33] Mozilla. Binaryen. `https://github.com/WebAssembly/binaryen`, 2021. Accessed: March 3, 2023.

[34] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. New kid on the web: A study on the prevalence of WebAssembly in the wild. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, pages 23–42. Springer, 2019.

[35] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. Swivel: Hardening WebAssembly against spectre. *arXiv preprint arXiv:2102.12730*, 2021.

[36] Bobby Powers, John Vilk, and Emery D Berger. Browsix: Bridging the gap between UNIX and the browser. *ACM SIGPLAN Notices*, 52(4):253–266, 2017.

[37] Lei Qiao, Xuesong Li, Qasim Umer, and Ping Guo. Deep learning based software defect prediction. *Neurocomputing*, 385:100–110, 2020.

[38] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 419–428, 2014.

[39] Kimberly Redmond, Lannan Luo, and Qiang Zeng. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. *arXiv preprint arXiv:1812.09652*, 2018.

[40] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[41] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[42] Quentin Stiévenart and Coen De Roover. Wassail: a webassembly static analysis library. In *Fifth International Workshop on Programming Technology for the Future Web*, 2021.

[43] The Wasm Team. WebAssembly Binary Toolkit. https://github.com/WebAssembly/wabt, 2021.

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[45] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE, 2020.

[46] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 363–376, 2017.

[47] Yutian Yan, Tengfei Tu, Lijian Zhao, Yuchen Zhou, and Weihang Wang. Understanding the performance of WebAssembly applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 533–549, 2021.

[48] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. Vuldeepecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2224–2236, 2019.

[49] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706*, 2018.

APPENDICES

Appendix A

Results (Extra)

In this section, we include two extra results: (1) Figure A.1 illustrates the original code embeddings plot that has overlapping labels, and (2) Table A.1 provides the numerical accuracy results of the recovering precise return types evaluation.

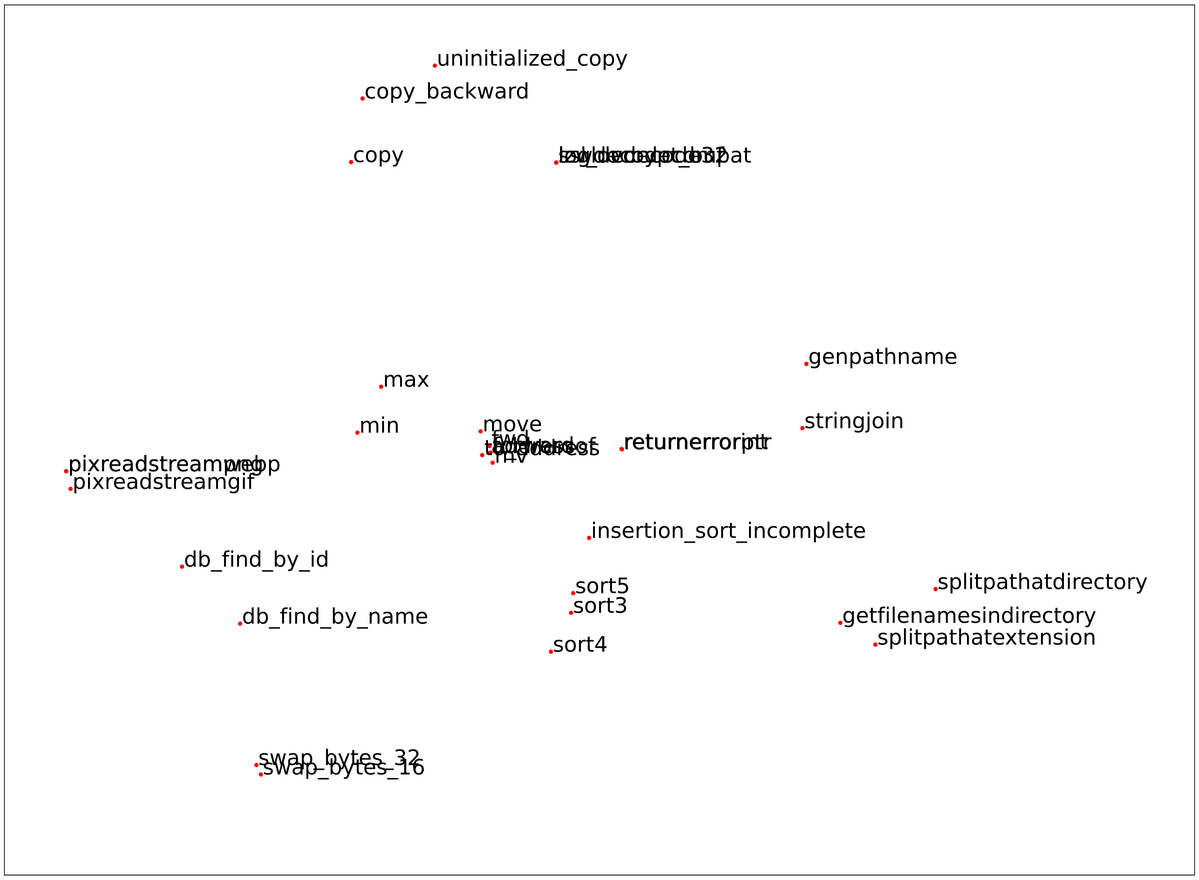


Figure A.1: A t-SNE 2D plot of code embeddings generated using our proposed code embedding approach. This is the original plot, and the labels are overlapped.

Table A.1: Recovering precise return types numerical accuracy results

	$\mathcal{L}_{\text{SW, All Names}}$		$\mathcal{L}_{\text{SW, }t_{\text{low}}\text{ not given}}$		$\mathcal{L}_{\text{SW}}$		$\mathcal{L}_{\text{SW, Simplified}}$		$\mathcal{L}_{\text{Eklavya}}$	
	top-1 acc.	top-5 acc.	top-1 acc.	top-5 acc.	top-1 acc.	top-5 acc.	top-1 acc.	top-5 acc.	top-1 acc.	top-5 acc.
seq2seq-INP	**41.70%**	**48.83%**	**54.34%**	**80.20%**	**53.75%**	**82.05%**	**61.20%**	**88.03%**	**76.47%**	**100.00%**
seq2seq-ISP	40.49%	47.19%	49.70%	79.07%	51.76%	80.27%	58.41%	88.12%	76.19%	100.00%
seq2seq-I	41.57%	45.93%	47.29%	73.94%	45.73%	74.13%	53.63%	85.60%	75.88%	100.00%
seq2seq-NP	35.32%	48.03%	48.26%	74.66%	46.56%	73.11%	49.05%	81.94%	68.41%	99.95%
seq2seq-SP	38.12%	47.08%	46.43%	73.32%	43.28%	67.87%	47.76%	78.06%	64.04%	99.91%

Appendix B

Code Repository

In this section we will provide a guideline on how to work with our replication package. Our replication package includes the following:

- *Dataset*: Our study utilized two path-based code representations generated from WebAssembly (Wasm) binaries compiled by SnowWhite. We processed the original dataset using our pipeline and created a new dataset for model training. Our replication package includes both the original dataset and our new dataset.
- *Pipeline*: We designed a pipeline in Rust and Python to extract path sequences from Wasm binaries.
- *Data cleaning*: We provide scripts that allow the dataset to be split into different variants and input sequences to be created.
- *Training notebooks*: Two Jupyter notebooks are included for training a feed-forward neural network to generate code embeddings for method names and for training seq2seq models with five input sequence variants.
- *Models*: This section includes the weights of the seq2seq models trained by OpenNMT and the feedforward neural network used to generate the code embeddings.
- *Results*: The log files in this section contain the evaluation results of our models, including prediction accuracy scores, BLEU scores, and other evaluation metrics.

We welcome contributions to improve our method. Please open an issue or submit a pull request.

B.1 Dataset

You can find our dataset in `./data/`. This folder includes the following files:

- `./data/binaries.7z.link`: We used the same Wasm binaries that SnowWhite provided. For more info see SnowWhite [1].
- `./data/dataset.7z.link`: We used the same dataset that SnowWhite provided.
- `./data/combined.7z.link`: This file contains a more compact version of the dataset introduced in SnowWhite. Bascially, we combined all the necessary information including labels for return type recovery and method name prediciton together, so we can only work with one training/test/dev file. This file is 23.2 MB and after decompression would be around 800 MB.
- `./data/seqs.7z.link`: This file includes the different input sequences for different dataset variants for both return type recovery and method name prediciton. These sequences will be used directly as inputs to seq2seq models. There are 5 sequences per dataset variant:
  - seq1: Nested-Paths only (seq2seq-NP)
  - seq2: Instructions only (seq2seq-I)
  - seq3: Nested-Paths and Instructions concatenated (seq2seq-INP)
  - seq4: Simple-Paths only (seq2seq-SP)
  - seq5: Simple-Paths and Instructions concatenated (seq2seq-ISP)

  This file is 300 MB and after decompression would be around 8.65 GB.
- `./data/embedding.7z`: This file contains feature vectors and their corresponding method name labels that we used to create Wasm code embeddings. The size of the file is 14 MB and after decompression would be around 4.5 GB.

  We used this command for compression:
  ```
  > 7z a <name>.7z <folder>
  ```
  Use the following command for decompression:
  ```
  > 7z x <name>.7z
  ```

[1] https://github.com/sola-st/wasm-type-prediction/

B.2 Pipeline

You can find the code for our path extraction and processing pipeline (WasmWalker) in `./implementation/extraction`. This folder includes the following:

- `./paths_set/`: This folder includes the empricial results of our path extraction step. `./paths_set/loop_if_collapsed.log` shows the 3,352 refined paths mentioned in our paper.
- `./accumulate.py`: The single-threaded program we used to collect the accumulative disribution of refined paths within our dataset.
- `./collect.py`: The multi-threaded program we used to collect the raw paths within our dataset.
- `./run.py`: The multi-threaded program we used to convert wasm binaries in our dataset to path sequences. To reproduce the preprocessing step, you would need to extract `./data/combined.7z.link` and `./data/binaries.7z.link` and change the paths accordingly.
- `./to_wat.py`: This file runs the Rust project at the root folder to use WABT::wasm2wat and store wat files in `./__logs__/`.
- `./path.py`: DFS for collecting paths and the refinement algorithm are implemented here.

B.3 Data Cleaning

The scripts we used to clean the data and split them into dataset variants and also provide different input sequences for each of them for method name prediction and return type recovery can be found at the following paths, respectively:

- `./implementation/scripts/method-name`
- `./implementation/scripts/type-pred`

B.4 Training Notebooks

To faciliate the process of reproducing our results, we provide the Jupyter Notebooks that we used on Google Colab to train our models. The notebook that we used to create code embeddings can be found at `./implementation/scripts/training/embedding.ipynb`. For training our seq2seq models for both method name prediction and return type recovery, we used the notebook provided at `./implementation/scripts/training/OpenNMT.ipynb`. These files include all the necessary config information that can be helpful for replicating our results.

B.4.1 Steps for reproducing the embedding t-SNE plot

- Extract `./data/embedding.7z`, then copy `./names.txt` and `./vectors.txt` to your working directory.
- Extract `./models/models.7z`, then copy `./embedding.h5` to your working directory. Alternatively, you can not use this file, and retrain the feedforward-NN. The code for training is provided in the notebook (it is commented out).
- Change the paths in `./implementation/scripts/training/embedding.ipynb` accordingly.
- Run code blocks in the jupyter notebook. The versions of the used packages are as follows: sklearn (1.2.2), keras (2.11.0), pandas (1.4.4), numpy (1.22.4).
- The generated t-SNE plot will have overlapped labels, to separate them, uncomment the commented lines in the last code block.

B.4.2 Steps for reproducing the seq2seq models

- Extract `./data/seqs.7z.link`, then copy everything to your working directory.
- Extract `./models/models.7z`, then copy everything to your working directory.
- Change the paths in `./implementation/scripts/training/OpenNMT.ipynb` accordingly. The main path that you have to change is in the fifth code block:

```
import torc
torch.cuda.is_available()
torch.cuda.get_device_name(0)
folder = "path/to/sequences"
```

For instance, one example for path can be `folder = "./seqs/type_pred/eklavya/seq1"`— to reproduce results for seq2seq-NP model for eklavya variant.

- If you want to retrain the seq2seq network then keep the code block containing the following command:

```
# Train the NMT model
!onmt\_train -config config.yaml
```

- If not, then comment the previous block and run the next blocks. Just remember to replace `--model='./model_best.pt'` with the path to the model you want from `./models/models.7z`.

B.5 Models

You can find the weights for neural models that we trained in `./models/models.7z`. The size of the file is 4.41 GB and after decompression would be around 5.07 GB. After decompression you would find the weights of seq2seq models for different variants and different input sequences for both method name prediction and return type recovery in `./models/method_name/` and `./models/type\_pred/`. The models are generated using OpenNMT (.pt files). The naming of the file indicates the dataset variant and input variant. For method name prediction, first token is the value of m and second token is the chosen input sequence. For instance, `20_seq3.pt` is when $m = 20$ and seq3 is chosen. m is the minimum number of datapoints associated with a method name for it to be included in the dataset. For example, if a dataset is created with $m = 50$, that means each method name in the dataset has at least 50 datapoints associated with itself. For return type recovery, first token is the variant of dataset, similar to SnowWhite and second token is the chosen input sequence.

In addition to the seq2seq weights, you would find `./embedding.h5` which is the weights of the keras feedforward NN we used for creating code embeddings.

B.6 Results

You can find the results files of our evaluating our models for method name prediction and return type recovery in `./results/method_name` and `./results/type_pred`. Each log file includes prediction accuracy scores, BLEU scores, and other evaluation metrics. We used the script provided by SnowWhite authors to generate these metrics. You can find the script at `./results/eval.py`. The naming of the file indicates the dataset variant and input variant. For method name prediction, first token is the value of m and second token is the chosen input sequence. For instance, `20_seq3.log` is when $m = 20$ and seq3 is chosen. For return type recovery, first token is the variant of dataset, similar to SnowWhite and second token is the chosen input sequence.