Finding the Dwarf: Recovering Precise Types from WebAssembly Binaries

Daniel Lehmann
mail@dlehmann.eu
University of Stuttgart
Germany

Michael Pradel
michael@binaervarianz.de
University of Stuttgart
Germany

Abstract

The increasing popularity of WebAssembly creates a demand for understanding and reverse engineering WebAssembly binaries. Recovering high-level function types is an important part of this process. One method to recover types is data-flow analysis, but it is complex to implement and may require manual heuristics when logical constraints fall short. In contrast, this paper presents SnowWhite, a *learning-based* approach for recovering precise, high-level parameter and return types for WebAssembly functions. It improves over prior work on learning-based type recovery by representing the types-to-predict in an *expressive type language*, which can describe a large number of complex types, instead of the fixed, and usually small type vocabulary used previously. Thus, recovery of a single type is no longer a classification task but sequence prediction, for which we build on the success of neural sequence-to-sequence models. We evaluate SnowWhite on a new, large-scale dataset of 6.3 million type samples extracted from 300,905 WebAssembly object files. The results show the type language is expressive, precisely describing 1,225 types instead the 7 to 35 types considered in previous learning-based approaches. Despite this expressiveness, our type recovery has high accuracy, exactly matching 44.5% (75.2%) of all parameter types and 57.7% (80.5%) of all return types within the top-1 (top-5) predictions.

CCS Concepts: • **Software and its engineering → Software notations and tools**; • **Security and privacy → Software reverse engineering**.

Keywords: WebAssembly, type prediction, type recovery, reverse engineering, debugging information, DWARF, neural networks, machine learning, dataset, corpus.

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523449

ACM Reference Format:
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 '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523449

1 Introduction

WebAssembly is a portable bytecode language, with unmanaged, linear memory, low-level instructions, and near-native performance [25]. Originally designed for in-browser execution, it has expanded to many other application domains, including server-side code with Node.js, function-as-a-service cloud computing, standalone WebAssembly VMs, and smart contract systems. WebAssembly binaries are often compiled from C and C++, and more recently also from a multitude of other languages, such as Rust or Go [30].

Because WebAssembly programs are in a low-level, binary format, understanding them is all but trivial. At the same time, due to its increasing popularity and the multitude of application domains, there is ample demand for reverse engineering WebAssembly code. For example, a developer integrating a third-party WebAssembly module may want to better understand its exported functions, or audit it to prevent supply-chain attacks [75]. Security experts may need to analyze malicious WebAssembly binaries, which, e.g., try to escape from the browser sandbox [3, 65], or perform unsolicited cryptocurrency mining [37, 51, 63, 68, 70]. Finally, good reverse engineering tools are even more important when malicious JavaScript code is intentionally hidden inside or compiled to WebAssembly for obfuscation [50, 62].

An important first step toward understanding a WebAssembly binary is to understand the type signatures of its functions. Because types are highly relevant for understanding low-level code, they are targeted by existing reverse engineering tools for native binaries [12, 14, 57]. Developer studies also show that static types help to understand code [26, 49].

Unfortunately, the types available in a WebAssembly binary are only of very limited help. WebAssembly code *is* statically typed, but there are only four low-level primitive types: integers and floats of 32 bits and 64 bits. To a reverse engineer, those are not very informative. E.g., an `i32` could be a signed or unsigned integer in the application domain, a size in bytes, an array, a pointer to a struct, or one of many

other source types. Thus, in addition to WebAssembly's four low-level types, it would be beneficial to recover precise, high-level types similar to those used in the programming language the binary was compiled from.

One avenue to recover high-level types is based on "classical" data-flow analysis or type inference, which collects constraints based on how values are used in the program [12]. However, this is complex to implement and often builds on heavy analysis frameworks, such as BAP or CodeSurfer [43, 54]. Supporting WebAssembly, especially with its slightly unusual stack machine [25], would be a non-trivial undertaking. More fundamentally, not all information can be expressed as logical constraints, so manual heuristics are often still employed to present simplified, intuitive types in the end [54].

In contrast, we pursue a *data-driven, learning-based approach*, in line with general guidelines on when neural software analysis is beneficial [58]. E.g., some sequences of instructions may only provide a statistical hint, but no guarantee about the type of a parameter. Similarly, there is often no single correct solution in type recovery, e.g., both `class` and `struct` might be valid in terms of constraints, yet convey different intuitions to a human reverse engineer. Finally, there is, at least in principle, ample data to learn from, as arbitrary code can be compiled to WebAssembly and debug information can provide type labels for supervised training.

Various learning-based approach for predicting types in other languages have been proposed in recent years. They consider either binaries for native architectures [14, 27, 47] or dynamically typed source languages, e.g., Python [5, 59] and JavaScript [28, 48, 61]. These approaches explore different input representations, e.g., token sequences [28], data flow graphs [5], and natural language associated with code [48], and different model architectures and ways of training them, e.g., recurrent neural networks [59], transformers [2], graph neural networks [5], and unsupervised pre-training [57].

Unfortunately, current learning-based approaches suffer from two key limitations. On the practical side, to the best of our knowledge, no existing approach predicts high-level types for WebAssembly. More fundamentally, prior work focuses either on how to represent the code for which types are predicted, or on what machine learning model is most suitable for this task. In contrast, another important aspect of type prediction is currently understudied: *How to represent the to-be-predicted types themselves?* Almost all existing papers, with one noteworthy exception [5], formulate type prediction as a classification problem. As classification scales poorly to a large number of classes, this formulation typically implies a small number of types to choose from. For example, recent binary-level type prediction models [14, 27, 47, 57] support only 7, 11, 17, and 35 types, respectively.

This paper addresses both the lack of a type prediction approach for WebAssembly in particular, and the limitations of previous learning-based approaches to represent types for binary type recovery in general. We present SnowWhite, a learning-based approach for predicting high-level function types for a given WebAssembly binary. The core technical contribution is using an expressive language for describing the types SnowWhite can predict. The language supports primitive types, named types, complex types, such as pointers, arrays, and enums, as well recursive combinations of all the above. Because different source languages compile to WebAssembly [30], the type language is derived from the DWARF debugging format [17], which is widely supported by several compilers for different source languages.

Given the type language, SnowWhite trains a model to predict types as a sequence of tokens. That is, we formulate the type prediction problem as a sequence prediction, and not a classification task. An important advantage of sequence prediction is that we do not have to artificially limit the number of types the model can choose from, but instead support (at least in principle) infinitely many types.

To train and evaluate SnowWhite, we also gather the first large-scale dataset of WebAssembly binaries with debugging information. Based on the DWARF information provided by the compiler, we can associate each WebAssembly function with its return type and parameter types. Our dataset consists of 6.3 million types in 300,905 WebAssembly object files compiled from over 4,000 C and C++ Ubuntu source code packages. The dataset is two orders of magnitude larger than datasets considered in previous work on WebAssembly compiled from source code, which consider only tens of programs [32, 44]. Beyond this paper, we envision the dataset to provide a basis for other learning-based work on WebAssembly, e.g., to predict the names of program elements or to decompile WebAssembly code back to source code.

Our evaluation shows that the type language expresses 1,225 different types, i.e., many more than prior work on binary type prediction [14, 27, 47, 57], while also offering a more uniform type distribution. Despite this expressiveness, the type prediction model exactly predicts 44.5% (75.2%) of all parameter types and 57.7% (80.5%) of all return types within the top-1 (top-5) predictions, clearly outperforming a statistical baseline approach based on the data distribution.

In summary, this paper contributes the following:

- Addressing the practical problem of predicting high-level types of WebAssembly functions, which is important for understanding WebAssembly binaries.
- A type language for binary type recovery that is much more expressive than the small number of labels used in prior learning-based approaches (Section 3).
- Formulating the type prediction as a sequence prediction task, which facilitates accurate predictions across a large number of types to choose from (Section 4).
- Creating and sharing the so-far largest dataset of WebAssembly binaries with debug information (Section 5).

Our dataset, code, and results are publicly available at https://github.com/sola-st/wasm-type-prediction.

```
1  void amd_control(double Control[]) {
2    double alpha;
3    int aggressive;
4    if (Control != (double *) NULL) {
5      alpha = Control[DENSE];
6      aggressive = Control[AGGRESSIVE] != 0;
7    } else {
8      alpha = DEFAULT_DENSE;
9      aggressive = DEFAULT_AGGRESSIVE;
10   }
11   if (alpha < 0) {
12       printf("no rows treated as dense");
13   }
14   [...]
15 }
```

(a) Excerpt of the original source code in C.

```
1  0073: Code section start
2  0076: function $1: ;; Numerical indices
3          type (param $0 i32) (result)
4          [...]       ;; ^ Low-level type
5  0091:   block
6  0093:     local.get $0   ;; Push param.
7  0095:     br_if 0   ;; Jump if non-zero
8            [...]
9  00e9:   end    ;; Jump past this instr.
10 00ea:   local.get $0     ;; Push param.
11 00ec:   f64.load offset=8  ;; Mem. read
12         [...]
13 00f7:   local.get $0     ;; Push param.
14 00f9:   f64.load offset=0  ;; Mem. read
```

(b) Compiled WebAssembly binary from (a), with byte offsets into the binary shown on the left.

```
1  0026: DW_TAG_pointer_type
2          DW_AT_type @ 002b
3  002b: DW_TAG_base_type
4          DW_AT_name: "double"
5          DW_AT_encoding: float
6          DW_AT_byte_size: 8
7  0033: DW_TAG_subprogram
8          DW_AT_name: "amd_control"
9          DW_AT_low_pc: 0x0003
10 0047:   DW_TAG_formal_parameter
11           DW_AT_name: "Control"
12           DW_AT_type @ 0026
```

(c) DWARF debugging information.

```
pointer primitive float 64
```

(d) High-level type to predict.

Figure 1. Real-world example of a function's source code, compiled binary, DWARF information, and high-level type to predict.

2 Overview

This section gives an overview of SnowWhite, starting with a motivating example, then defines the problem more precisely, and presents the main components of the approach.

Motivating example. Figure 1a shows the source code of a function from `libglpk`, a linear-programming library written in C, contained in the Ubuntu repositories. The function has one parameter, which is declared as an array of `double`s (line 1). If it is non-`NULL`, the function reads two values from the array, and otherwise uses defaults (lines 4–10).

Compiling the function to WebAssembly yields the code in Figure 1b. The comments are for illustration only and not part of the actual binary. On the right, Figure 1c shows the type of the parameter as represented in the DWARF debugging format [17]. DWARF data is embedded in binaries when compiling with debug information (`-g`), but not present in stripped binaries a reverse engineer typically encounters. It is a hierarchical binary format, with single-tag entries (**`blue`**) that have multiple attributes (**`teal`**), and potentially multiple other entries as children (e.g., the parameter in line 10 is a child of the function in line 7). As attributes can refer to other entries (lines 2, 12), the information forms a directed, possibly cyclic graph. In the example, the parameter entry refers to a `pointer` type entry (line 1), which in turn refers to its element type, a primitive 64-bit float type (line 3).

Problem definition. The goal of SnowWhite is to predict precise, high-level types from WebAssembly binaries. Because understanding functions and their types are valuable first steps to a reverse engineer understanding the functionality of a binary, we focus on function parameter and return types. More formally:

Definition 2.1. The *type prediction problem* is to find a mapping of the form

$$f, e, t_{low} \longrightarrow t_{high}$$

where

- f is the body of a given WebAssembly function with k parameters and zero or one return value,
- $e \in \{param_1, ..., param_k, return\}$ is the desired element from the function signature to predict a type for,
- $t_{low} \in \{\texttt{i32}, \texttt{i64}, \texttt{f32}, \texttt{f64}\}$ is the low-level WebAssembly type of e, already present in the binary, and
- $t_{high} \in \mathcal{L}_{types}$ is a type defined by a high-level type language $\mathcal{L}_{types}$.

SnowWhite predicts the type of each parameter and the return type separately. As WebAssembly is statically typed, the low-level type of each program element is known, which we exploit by providing it as an input to the approach. The main novelty of SnowWhite is how to represent the output t_{high} of the type prediction task. One option would be to predict one out of a fixed set of types, as done in prior work on binary type prediction [14, 27, 47, 57]. For example, we could aim to predict simply that the function parameter in Figure 1 is a pointer. While providing a relatively easy prediction task, that approach misses many details relevant for understanding the functions in a binary. Another option would be to predict the full DWARF type (Figure 1c). However, full DWARF types contain various details that are irrelevant for a reverse engineer, making the prediction task unnecessarily hard.

Instead of the above two extremes, SnowWhite predicts types that are sentences in a high-level type language. For example, the type in Figure 1d expresses the fact that the parameter is a pointer to a memory location that stores a primitive 64-bit float. Our high-level type language is derived from DWARF, and hence can express types across multiple source languages that are commonly compiled to WebAssembly, such as C and C++. In contrast to DWARF, the type language omits details that are not crucial to a reverse engineer but that would make prediction harder.

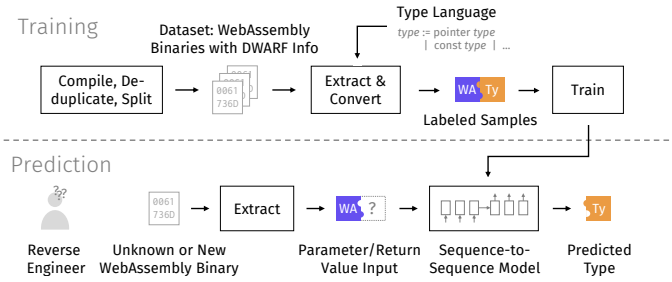


Figure 2. Overview of SnowWhite's components.

SnowWhite in a nutshell. To address the type prediction problem, SnowWhite uses a neural sequence-to-sequence model and a large-scale dataset of WebAssembly binaries with DWARF type information. Figure 2 shows the two major phases of the approach: In the *training* phase, the sequence-to-sequence neural network model is trained from labeled data. As there exists no suitable large, real-world dataset of WebAssembly functions with high-level type information, we create a large-scale dataset by compiling 4,081 Ubuntu source packages to WebAssembly, resulting in more than 6.3 million labeled samples. In the *prediction* phase, a reverse engineer can query the trained model with previously unseen WebAssembly functions, to obtain a high-level type prediction for parameters and return values.

The following sections present the type language (Section 3), the neural prediction model (Section 4), and the collected dataset (Section 5) in more detail.

3 High-Level Type Language

Prior work has explored different input representations and model architectures for binary type prediction [14, 27, 47, 57]. Much less focus has been on the representation of types for the prediction task itself. One core contribution of this work is to describe the types to predict using a type language that includes precise information about primitive types, nested types, const and signed-ness, and type names. Figure 3 gives a BNF grammar of the language. Table 1 compares our type language against those used in prior learning-based binary type prediction and against full DWARF information [17].

3.1 Representing the Types to Predict

There are two extremes in terms of how types can be represented: On the one end of the spectrum, types can be represented as a small, *fixed set* of choices. This is the case for prior work, as shown in the first four rows of Table 1. The $|\mathcal{L}|$ column shows the number of unique types as reported in the respective papers. Even though a type grammar is sometimes presented, this is only for illustration and the sets of types these grammars describe are all finite and small.

One virtue of a fixed set of types is simplicity, both in terms of data extraction and the model architecture, as classification tasks are simple to train and evaluate. The downside

$$
\begin{aligned}
\mathit{type} ::= {} & \texttt{primitive}\ \mathit{primitive} && \text{(primitive types)}\\
\mid {} & \texttt{pointer}\ \mathit{type} \mid \texttt{array}\ \mathit{type} && \text{(pointers and arrays)}\\
\mid {} & \texttt{const}\ \mathit{type} && \text{(const-ness)}\\
\mid {} & \texttt{name}\ \mathit{name}\ \mathit{type} && \text{(nominal types and typedefs)}\\
\mid {} & \texttt{struct} \mid \texttt{class} \mid \texttt{union} \mid \texttt{enum} && \text{(aggregates)}\\
\mid {} & \texttt{function} && \text{(for function pointers)}\\
\mid {} & \texttt{unknown} && \text{(uninformative type)}\\
\mathit{primitive} ::= {} & \texttt{bool} && \text{(booleans)}\\
\mid {} & \texttt{int}\ \mathit{bits}_{\texttt{int}} \mid \texttt{uint}\ \mathit{bits}_{\texttt{int}} && \text{(integers)}\\
\mid {} & \texttt{float}\ \mathit{bits}_{\texttt{float}} \mid \texttt{complex} && \text{(floating-point)}\\
\mid {} & \texttt{cchar} \mid \texttt{wchar}\ \mathit{bits}_{\texttt{wchar}} && \text{(characters)}\\
\mathit{name} \in {} & \{\texttt{"size\_t"}, \texttt{"FILE"}, \texttt{"string"}, \ldots\} && \text{(names)}
\end{aligned}
$$

$$\mathit{bits}_{\texttt{int}} \in \{8,16,32,64\},\ \mathit{bits}_{\texttt{float}} \in \{32,64,128\},\ \mathit{bits}_{\texttt{wchar}} \in \{16,32\}$$

Figure 3. Grammar of the high-level type language $\mathcal{L}_{\text{SW}}$.

is that there is a mismatch between the fixed set of types and the infinite types in the source program (e.g., in C and C++), where more complex types can be built up from simple ones by composition. As such, the source types need to be heavily simplified to map to the target set, which loses information and equates many potentially different types.

On the other end of the spectrum stands the full type language of DWARF. Types therein are directed, possibly cyclic graphs, allowing DWARF to capture recursive types. A downside of this representation is that predicting graphs with neural networks is challenging; recent work we know of only encodes graphs, but does not predict them [6, 13, 29]. Full DWARF types also contain constructs that are unlikely to be recoverable from binaries, e.g., optimization hints, language-specific constructs, and domain specific names.

With our type language, we aim to strike a balance between those two extremes. The grammar in Figure 3 produces types that are represented as a *linear sequence of type tokens*. The set of all possible types is infinite, and while we do not represent the fields of aggregates, we do allow nested types for pointers, arrays, `const` and names. Besides describing a larger set of types than prior work on learning-based type prediction, the fact that each type in our language is a sequence of type tokens allows us to formulate type prediction as a sequence prediction task.

To produce a type sequence in our language from the DWARF information in a binary, we recursively traverse the DWARF type graph, pattern match on the type constructor (e.g., `DW_TAG_pointer_type` in Figure 1c) and convert it to a type constructor of Figure 3 or remove it. Figures 1c and d show an example of a DWARF type represented in our language. We break cycles to prevent generating infinite type sequences. We now describe the features of our grammar in more detail.

Table 1. Comparing different type languages of learning-based binary type prediction. ✓ means a feature is supported, ✗ means not. (✓) in the 'Prim. Size' column means C type names are used, instead of an exact, unambiguous representation.

	$\lvert\mathcal{L}\rvert$	Type Structure	`int` / `char`	`bool`	`float`	`int` Sign	Prim. Size	`struct`	`class`	`union`	`enum`	`array`	`const`	Pointer Pointee Type	Names	Fields	Opt. Hints	Lang.-specific
Eklavya [14]	7	Fixed set	✓	✗	✓	✗	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗
Debin [27]	17	Fixed set	✓	✓	✗	✓	(`int`)	✓	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗
TypeMiner [47]	11	Fixed set	✓	✓	✓	✗	(`int`)	✗	✗	✗	✗	✓	✗	`struct`, `char`, `func`	✗	✗	✗	✗
StateFormer [57]	35	Fixed set	✓	✗	✓	✓	(✓)	✓	✗	✓	✓	✓	✗	Single level	✗	✗	✗	✗
SnowWhite	∞	Sequence	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Recursive	Top k	✗	✗	`class`
Full DWARF	∞	Full graph	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Recursive	✓	✓	✓	✓

3.2 Primitive Types

Primitive types in binaries, as represented in DWARF, are surprisingly complex. All type prediction approaches have some representation of integers, but they differ in their precision and how they handle other primitives, which can lead to ambiguous or incorrect type labels (columns 4–8 of Table 1).

First, our language has an explicit boolean type. Even though on a machine-level, boolean values are represented as integers, we believe the type distinction is important and instructive for reverse engineers, e.g., `bool` is more telling than just `int`. Eklavya and StateFormer do not distinguish the two and map booleans to integers instead.

Second, we support floats of different width (single, double, and quad precision) and the C built-in `complex` type. In contrast, floating point types are not handled by Debin at all, and Eklavya and TypeMiner do not capture their width.

Third, our language represents integer types precisely. Eklavya does not distinguish different sizes or signed-ness of integers, e.g., `short` and `long long` are mapped to the same type. A better, but still naive approach is to represent integer types by their name in the source code, e.g., `int` or `unsigned long`. This is problematic, because the relation between the source code name and the machine representation of integers is both ambiguous and not injective. The mapping of name to representation is not injective because different names can map to the same type, e.g., `short`, `short int`, and `signed short int` (and even permutations thereof) are all the same type in C. In other languages, the same type is called differently again, e.g., `i16` in Rust. Using the source code name for identification would thus introduce distinct classes for what is really the same type. Additionally, the mapping from name to representation is ambiguous, e.g., `long` can be both 32 bits wide or 64 bits wide, depending on the compiler's data model (ILP32 vs. LP64). That is, a reverse engineer cannot tell the bit-width from `long` alone. Thus, unlike prior work, which uses C and C++ type names to identify integers, we choose an unambiguous and language-independent representation based on bit-width and signed-ness.

Finally, our type language models character types precisely. In C and C++, `signed` and `unsigned char` are just 8-bit integers, and encoded in our approach as such. A "plain" `char` in C is different from both, and used only for character data, not in arithmetic operations. It commonly appears in string-handling functions. We represent it as `cchar`. *Wide char* types, e.g., `wchar16_t` in C++, are used for 16 and 32-bit unicode characters and modeled in our language as well. Prior approaches do not distinguish between those types.

In summary, our encoding normalizes all primitive types appearing in the dataset to an unambiguous representation of 16 types. It does not conflate different types with each other, and assigns exactly one type name per unique underlying primitive type. Notably absent are high-level data structures such as strings, lists, or dictionaries, as they are not built-in in systems-level languages, unlike, e.g., in Python or JavaScript.

3.3 Pointers and Aggregate Types

More complex aggregate types can be built up from constituents, e.g., in array, pointer, or struct types. Our type language supports both arrays and pointers (unlike Eklavya, which maps the former to the latter) and also captures the nested type of their elements and pointee, respectively. This is unlike Eklavya and Debin, where all pointers are `*void`, regardless of what they actually point to. TypeMiner and StateFormer discern certain classes of pointers from each other, e.g., pointers-to-structs from function pointers, but are not recursive, and as such cannot represent, e.g., the C type `*char[]` (`array pointer char` in our type language).

We do not capture individual fields of aggregate types like structs and unions, which is where we lose information compared with full DWARF types. As not every field of a struct or union is used in a given function, prediction of field types is a challenge left for future work. To model function pointers, our language includes a `function` constructor.

3.4 Type Attributes and Language-Specific Types

DWARF information includes type attributes, e.g., `const`. We include `const` as a type constructor into our language. This allows a reverse engineer to discern between a pointer to constant data (`pointer const` t), a pointer whose value is constant (`const pointer` t), and a mutable `pointer` t, and thus gives useful information about the invariants of the source program. This is similar to the constraint-based Retypd [54],

but unlike all prior learning-based approaches. DWARF types also contain the attributes `volatile` and `restrict`, but since those are optimization hints for the compiler and likely hard to recover, we remove them when traversing the input type.

We also aim to recover some language specific types, notably the distinction between a `class` and a `struct`. We believe this is useful to reverse engineers because classes point to object-oriented programming, frequently have methods, and implicitly identify the source language as C++, whereas structs are idiomatic for plain old data. Neither learning-based nor constrained-based approaches so far aimed to recover this distinction, instead equating classes and structs into a single concept. The distinction between C++ references and C pointers is less instructive and more difficult to recover, so we map those to a single `pointer` constructor.

3.5 Unknown and Unspecified Types

In some cases, the type information in DWARF can be incomplete. One common cause are forward declarations in C and C++, such as `struct name;`. While a forward declared type cannot be used directly in parameters or return types, they frequently appear behind pointers. Similar are `void` pointers, e.g., in the return type of `malloc` and other generic functions. A third case is the C++ `nullptr` expression, which has an unnameable type `decltype(nullptr)`. In all three cases the element type is unknown, but we still know that this is a generic pointer. We thus feature an `unknown` constructor, similar to an uninformative type $\top$ in other type systems, and encode all three mentioned cases as `pointer unknown`.

3.6 Names and Typedefs

The type language so far is precise, but still purely structural and fairly low-level, thus capturing little "human intuition" about the high-level semantics of types. Type names can convey such semantics and are included in DWARF, so we would like to recover them (at least partially) in our prediction task. This sets us apart from prior work on learning-based type prediction from binaries, that never went beyond primitive types and simple aggregates. Constraint-based approaches also either ignore type names fully, or rely on manually written rules for some well-known functions [54]. There are several challenges when representing names in types.

Names in DWARF types appear in two places, namely in typedefs and in named aggregate type definitions, such as `struct`, `class`, or `union`. In both cases, names are used to ascribe meaning, but only the latter introduces a nominal type with strict typing discipline. Typedefs are merely aliases, i.e., can be freely exchanged with the underlying type. Should we thus remove typedefs? We argue not, because their names still convey useful information, e.g., the type `size_t` is more instructive than just integer. We thus map names in typedefs and names in datatype definitions to a single `name` constructor that also contains the underlying structural type. E.g., `name "size_t" uint 32` captures both the name and the underlying structural type.

Next, what to do with nested names? Those can appear either because of the previous conversion, or simply because of repeated typedefs. Consider the frequent example of a typedef in conjunction with a struct definition:

```
typedef struct sname { /* fields... */ } tname;
```

Which name should be used for the type: `tname` or `sname`? We solve this by filtering the names (see below), and then keeping only the outermost (i.e., first) `name` constructor in a type sequence, as this is most likely the user-visible name. I.e., the example would be represented as `name "tname" struct`.

Finally, out of all names in the DWARF information, we keep only a subset in our language. First, because of the age and low-level nature of C and C++, there is less of a shared type vocabulary than there is, e.g., in Python, and many programs define their own data structures and even names for primitives. We do not want to predict such domain or project specific names. Second, very infrequent type names cannot be realistically predicted, and unlike in type-prediction approaches for high-level languages, the input binary contains no natural language data that a copy mechanism [22] could use to generate type names from the input.

We thus extract a set of k common names (from typedefs and named datatypes), where we define common as all names that appear at least once in 1% of all compiled packages in our dataset. Additionally, we filter out names that start with an underscore (as those are likely internal) or where the typename is equal to what we already capture in our primitive type representation, such as the name `uint32_t`. For generic types, e.g., the C++ template `std::vector<int>`, type arguments are included in the name. An alternative would be to abstract over the argument, but for simplicity and to retain more information, we keep the name as-is.

3.7 Type Language Variants

To evaluate the effect of different type languages on the type distribution and the accuracy of type prediction, we also define two variants of our language. We call the type language described so far $\mathcal{L}_{\text{SnowWhite}}$ (or $\mathcal{L}_{\text{SW}}$ for short). First, we define a variant of $\mathcal{L}_{\text{SW}}$ that contains all type names in the dataset. That is, it has the same grammar as in Figure 3, and performs the same mapping of typedef and datatype names to a `name` constructor, and keeps only the outermost name as described in Section 3.6, but it does not restrict the set of names based on the number of packages they appear in. Consequently, many more types are named in this language. Second, we define a simplified version of $\mathcal{L}_{\text{SW}}$, which removes the following constructors from Figure 3: `const`, `class`, and `name`. Consequently, types in this language are never named, classes are represented as `structs`, and `const` constructors are flattened away. This makes the language considerably simpler and closer to prior binary type prediction work. We discuss the effect of those variants on prediction in Section 6.

4 Type Prediction Model

We now present how SnowWhite predicts high-level types for WebAssembly functions with a neural model. Our work is the first to formulate prediction of a single type as a sequence prediction task p

$$p : (i_1, ..., i_m) \rightarrow (t_1, ..., t_n)$$

where $(i_1, ..., i_m)$ are instruction tokens extracted from the WebAssembly function body (Section 4.1) and $(t_1, ..., t_n)$ are type tokens as described by our type language. We address the prediction task p with a state-of-the-art, sequence-to-sequence neural network model (Section 4.2) trained in a supervised manner to minimize the difference between the predicted type and the known actual type.

4.1 WebAssembly Input Representation

We have described the type language and hence the type tokens t_i in Section 3, but we also need to represent the WebAssembly input as tokens i_i.

Extracting instruction tokens. To predict the parameter types and the return type of a function f, SnowWhite creates a sequence $(i_1, ..., i_m)$ of instruction tokens from f's representation in the WebAssembly binary. First, we disassemble the binary into a sequence of instructions for each function. In contrast to native binary formats, e.g., x86, static disassembly is well-specified and robust for WebAssembly. Then, we represent each instruction as in the WebAssembly text format (e.g., `i32.const 42` for the instruction that pushes 42 on the stack), and delimit individual instructions by '`;`'. We omit static arguments of instructions that are likely unhelpful and unnecessarily increase the number of tokens, namely alignment hints for memory accesses and the function index of the callee in `call` instructions. For predicting the type of a parameter p, we replace the index of p in `local.get`, `local.set`, and `local.tee` instructions with the special token `<param>`, to indicate to the model which parameter to focus on. Finally, we also add the low-level type (e.g., `i32`) of the parameter or return value to predict at the beginning of the sequence, delimited by a `<begin>` token.

Handling long functions. Binary code can have very long functions, which results in even longer token sequences. In our dataset, 10% of the functions have more than 1,000 tokens and 1% even more than 5,500 tokens. Because recurrent neural networks have trouble handling long sequences [55], and to facilitate efficient training in mini-batches, sequences are truncated and padded to a fixed length (here: 500 tokens) during training. Prior learning-based approaches on code often completely filter out long samples from the dataset, both during training and evaluation [14, 28, 42]. As such filtering may unrealistically inflate accuracy, we do not follow this practice. Instead, SnowWhite extracts windows of instructions around instructions related to the to-be-predicted type.

For predicting a parameter type, the approach extracts fixed-size windows around all instructions that use the parameter (`local.get`, `local.set`, and `local.tee`), and concatenates the windows, omitting the instructions in between. For prediction of a return type, SnowWhite extracts all windows ending in a `return` instruction. Windows are delimited by a `<window>` token from each other.

Example. For illustration, consider the following sequence of i_i tokens, extracted for predicting a parameter type:

('`i32`', '`<begin>`',
'`i32.const`', '`42`', '`;`', '`local.get`', '`<param>`', '`;`', '`call`', '`<window>`',
'`i32.add`', '`;`', '`local.set`', '`<param>`', '`;`', '`i32.eqz`')

It starts with the low-level type `i32` of the parameter to predict and then contains two windows of $w = 3$ instructions each, extracted around usages of the parameter. By default, we extract windows of size $w = 21$, i.e., 10 instructions to the left and right of parameter usages, and 20 instructions before a return instruction.

Token-level embedding. Finally, the tokens, both for the WebAssembly code and the type language, need to be converted to real-valued vectors for the neural network. We jointly train embedding layers that map each individual token to a dense vector of dimension e, with one embedding for WebAssembly tokens and one for type tokens. If we would naively embed all WebAssembly tokens, one issue is the very large number of unique, but infrequent tokens in code [33]. In particular, our dataset contains more than $v = 427,000$ unique WebAssembly tokens. The majority of those are numbers in the instructions, such as memory offsets, or integer and floating point constants. Using a very large vocabulary is undesirable due to increasing the number of model parameters (and thus memory usage), so instead we first build up a subword model based on byte-pair encoding (BPE) [64] that re-tokenizes the input into only $v' \ll v$ subword tokens. This breaks down infrequent source tokens into multiple subword tokens, which are then embedded with a much smaller embedding matrix, at the cost of slightly increased sequence length. We employ subword tokenization both for WebAssembly and for the type language.

4.2 Sequence-to-Sequence Model Architecture

To address the sequence-to-sequence prediction task, we reuse state-of-the-art results from neural machine translation (NMT). As the model architecture is standard, we keep this description short and refer to the available implementation and literature for details. The model is queried separately for every parameter of a function and its return type, i.e., only a single type (which is itself a sequence) is generated per prediction. For each type-to-predict, we present the model with a separate input sequence. For example, to predict the types of a function of two arguments, we would query the same model twice, but with slightly different inputs.

We train two separate models, one for parameter and one for return type prediction. We use the same configuration for both models, namely a bidirectional LSTM model with global attention [10, 46] and dropout for regularization [66], as implemented in the OpenNMT framework [40]. The network's weights are optimized by standard backpropagation-through-time gradient descent with the Adam optimizer [39]. As an alternative sequence-to-sequence architecture, we also explored Transformers [69], but did not find it improving accuracy, so we select the computationally much cheaper LSTM model. More experimentation with other model architectures is orthogonal to our work; there is ample work on alternative architectures and representations of code [6, 8, 19, 23, 29].

As hyperparameters, we choose after experimentation: $h = 512$ as the dimension of the hidden vectors, $l_e = 2$ two layers for the encoder and $l_d = 1$ a single layer for the decoder, $lr = 0.001$ as the initial learning rate and default momentums for Adam, $d = 0.2$ as the dropout rate, $e = 100$ as the dimension of the embeddings, and $v' = 500$ as the subword vocabulary size. The models have about 5.5 million learnable parameters in total.

5 Dataset

Training the models in SnowWhite for predicting types requires a dataset (i) from a diverse set of source programs, (ii) compiled to WebAssembly binaries, (iii) including the appropriate DWARF information, and (iv) resulting in an overall dataset size that is conductive to training a deep neural network. Previous work on studying WebAssembly performance [25, 32] and security [44] uses small datasets in the order of tens of programs. A recent dataset provides 8,400 WebAssembly binaries [30] but contains neither source code nor DWARF information. We thus collect our own large-scale dataset, which comprises 6.3 million WebAssembly code and type samples, extracted from 300,905 object files, which were compiled from 4,081 C and C++ Ubuntu packages. Beyond type prediction, we envision the dataset to also serve other purposes, e.g., for work on recovering names from stripped binaries, decompilation, or finding compilation issues.

Compiling to WebAssembly binaries. We start from all 70,065 source packages in the Ubuntu 18.04 repositories. Filtering out Linux kernel modules, which are unlikely to be compilable to WebAssembly, duplicates of applications for different locales, and fonts, 61,261 packages remain. We download their source code and keep all packages with at least one C or C++ source file. To compile the packages to WebAssembly, we modify the build scripts to use Emscripten, which is based on LLVM and the currently most popular compiler for WebAssembly. We add the `-g` flag to add debugging information in the DWARF format, but leave all other compilation options unchanged. In particular, all packages are compiled with their original optimization level (e.g., `-O2` or `-O3`), reflecting the options used for realistic binaries found in the wild and which a reverse engineer would encounter in practice. In total, 4,081 packages can be partially built to at least one object file, producing a total of 300,905 object files with WebAssembly code and DWARF information.[1]

Deduplication. Following the advice by Allamanis [4], we deduplicate the dataset to avoid artificially inflating our results. One option is to deduplicate individual functions or type samples. However, Allamanis [4] notes that in some tasks, duplication is part of the true data distribution. Functions are frequently duplicated across different binaries when they stem from the same statically linked library, which is the case in WebAssembly. Thus, instead of deduplicating on the level of functions or samples, we deduplicate at the level of binaries. As a first step, we remove exact duplicates of binaries, identified by hashing the full file contents.

To also remove near-duplicates, e.g., because of strings like build time included in the binary, we also compute an approximate signature of each binary. The signature takes only the function bodies into account, where each function gets a hash based on its *abstracted* instructions. The abstraction removes immediate arguments from the instructions, i.e., `local.get $0` is mapped to `local.get`, or `i32.load offset=8` to just `i32.load`. The function hashes are then concatenated (i.e., function order is taken into account) and the result is hashed again to obtain an approximate signature for the whole binary. Out of multiple binaries with the same signature, only one is finally retained in the dataset.

Overall, deduplication reduces the dataset from 3.8 billion instructions and 31 million functions in 300,905 object files, down to 866 million instructions and 7.9 million functions in 46,856 object files. Even after this strict deduplication, our dataset is much larger than prior binary type prediction datasets: It is 1.8× the size, in terms of number of instructions, of the four-architecture dataset in [57], 21× of [14], 32× of [27], and 1,059× of [47]. With an average function length of 109 instructions, we also believe our dataset is representative of real-world code, whereas an average length of ≈8 in [57] might indicate a dataset biased towards very short functions. Our deduplicated set comprises 21 GiB of WebAssembly binaries, which is 3.2× the size of the previously largest WebAssembly corpus [30], which additionally lacks debug information.

Matching WebAssembly to DWARF and filtering. To train SnowWhite in a supervised manner and as a ground truth for the evaluation, we associate WebAssembly functions with DWARF type information about the parameters and the return value. Function bodies are in the `code` section of a WebAssembly binary, whereas debug information is split over several custom sections `.debug_info`, `.debug_str`, etc.

[1] Final linking frequently fails because Emscripten uses `musl` libc instead of `glibc`. As pre-linking object files still contain WebAssembly and DWARF information, those can still be used to train and evaluate our approach.

We match each WebAssembly function with the corresponding DWARF information via the function's offset in the binary.

The number of parameters of a function and also whether a function has a return value, may differ between the source code and the compiled binary, e.g., due to optimizations. If the number of function parameters in the DWARF information and the number of parameters in the WebAssembly bytecode is the same, then we extract one parameter type sample for each parameter. Likewise, if a function has a non-`void` return type in DWARF and returns a value in WebAssembly, then we extract a return type sample. Because of this, we do not extract type samples for 6% of the 7.9 million functions in the deduplicated dataset, which we believe does not introduce a significant bias. Finally, to avoid that one Ubuntu package with many samples biases our dataset, we limit the number of samples per package to at most the number of samples in the second most frequent package.

The final dataset after all deduplication and filtering comprises 5.5 million parameter type samples and 796 thousand return type samples. The lower number of return type samples can be attributed to many C and C++ functions returning `void`, where no type needs to be predicted.

Splitting the dataset. We split the dataset into three portions: one for training, one for early stopping and evaluating hyperparameters, and a held-out test set. Randomly assigning each function or type sample to one of the three portions could cause (i) functions from the same binaries, or (ii) binaries from the same Ubuntu package ending up in two different portions of the dataset. Issue (i) is definitely unrealistic compared with the usage scenario of our approach, as the reverse engineer encounters a previously unseen binary, and (ii) means information from related binaries can leak from test to training data. To avoid both issues, we hence split the dataset by original Ubuntu packages. Since the total number of samples in our dataset is in the order of millions, the validation and test sets can be a relatively small portion of the overall dataset [9]. We choose 96% of the packages for training, and 2% for validation and testing, respectively.

6 Evaluation

We evaluate SnowWhite on the previously described dataset. In Section 6.2, we focus on the expressiveness of our type language and the resulting distribution of realized types. We show that the default variant of our language distinguishes 1,225 unique types and provides a more uniform type distribution than the small set of types predicted by prior learning-based approaches. In Section 6.3, we evaluate the accuracy of the type prediction model. We show that SnowWhite predicts 44.5% (75.2%) of all parameter types and 57.7% (80.5%) of all return types exactly within the top-1 (top-5) predictions. Finally, Section 6.4 qualitatively discusses the strengths and weaknesses of our approach with some representative examples of predicted and ground truth types.

6.1 Implementation, Setup, and Runtime

Our implementation, the dataset, and all scripts required to reproduce our work are publicly available at

`https://github.com/sola-st/wasm-type-prediction`.

The implementation consists of about 500 lines of Python and Bash for gathering the dataset, about 4,000 lines of Rust for extracting and processing DWARF types and WebAssembly code, and about 1,700 lines of Python for preparing the data and the neural model. We use the *gimli* and *wasmparser* libraries for parsing DWARF and WebAssembly, respectively. The neural model is built on top of *OpenNMT-py* for the neural model and *SentencePiece* for the subword tokenization.

We run all experiments on a server with with two Intel Xeon 12-core 24-thread CPUs running at 2.2 GHz, using 256 GiB of system memory, and Ubuntu 18.04 LTS as the operating system. For training the neural networks and during inference, we also use two NVIDIA Tesla T4 GPUs with 16 GiB of GPU memory each.

As usual, training neural networks takes orders of magnitude more time than prediction, but needs to be done only once. In our case, the fastest training run took about 1 hour 25 minutes and the slowest 11 hours 55 minutes on a single GPU. As a general rule, training a return type model takes less time than a parameter type model (due to fewer samples), and training with a simple type language takes less time than with a complex language (due to shorter type sequences). Prediction takes on average between 3ms and 40ms per input sample, including beam search to produce multiple predictions. Such near instantaneous results are another advantage of learning-based approaches, as no complex constraint solving is required.

We train all models on the training portion of the data set. During training, we check the accuracy on the validation set and stop early if it regresses. Due to the large dataset, the models converge after one to four epochs. We then take the best model from validation for final evaluation. All final type predictions are obtained on the test data, which the model has never seen and was not used to select the best model.

6.2 Type Language

The following evaluates the expressiveness of our type language and the type distribution that results from it.

Most common types. Table 2 shows the ten most common types in the dataset expressed in our type language. We observe that several features make the language distinguish large groups of types from each other that would otherwise be merged into imprecise labels. First and most importantly, we see that 7 out of the 10 most common types are some kind of pointer. Without tracking their pointee type, all of those labels would collapse into one, so the recursive nature of our type language is essential for informative predictions. Second, if we did not distinguish `classes` from `structs`, the

Table 2. Most common types in $\mathcal{L}_{\text{SnowWhite}}$ in our dataset. Short explanations for select types in italics.

Rank	Type	Sample Count	% Total
1	`pointer class` *a pointer to a class*	1,307,617	20.5%
2	`pointer struct`	918,332	14.4%
3	`primitive int 32` *a 32-bit signed integer*	771,690	12.1%
4	`pointer const class`	468,184	7.3%
5	`pointer const struct`	185,635	2.9%
6	`pointer const primitive cchar` *a pointer to constant character(s)*	184,586	2.9%
7	`name "size_t" primitive uint 32` *a 32-bit unsigned integer, named "size_t"*	181,204	2.8%
8	`primitive uint 32`	144,519	2.3%
9	`pointer unknown` *a pointer of unspecified pointee type*	114,139	1.8%
10	`pointer primitive int 32`	101,947	1.6%
Total Samples in Dataset		6,376,307	100%

largest two types would be merged into a single type accounting for 35% of all data, instead of only 20% and 14%, respectively. Third, `const`-ness is also useful, in particular in the pointee type of pointers. Without it, the types with rank four and five would be merged into the first two. Finally, we see that type names are useful to distinguish `size_t` from other, non-specified integers.

Most common names. Table 3 lists the eight most common type names as defined in Section 3.6, ordered by in how many packages they appear. In total, we extract 239 commonly used names from the dataset. Those names are well-known, semantic types of C and C++ programs, and they are not domain or project specific. The most common name `size_t` appears in almost two thirds of all projects, followed by `FILE` handles in a bit less than half of all projects. The distribution levels off quickly, with ranks three to six containing common types from the C++ standard library related to strings and I/O. All of these names are more useful to a reverse engineer than just the underlying structural type, e.g., `FILE` instead of `pointer struct`. Of the 239 names in our dataset, 141 (59%) also appear in the test data, so this feature is sufficiently exercised during testing.

When comparing the name distribution to type distributions from high-level languages [5, 59], complex data structures are notably absent, e.g., lists or maps. This shows that there is much less of a shared "type vocabulary" in binaries compiled from C and C++ than there is, e.g., in Python.

Expressiveness. To quantify how expressive our type language is, e.g., compared to a fixed set of types as considered in prior work [14, 27, 47, 57], we measure how many different types it describes in our dataset. The underlying assumption is that a larger set of types provides more precise type information to users of type prediction, e.g., during reverse engineering. Table 4 compares our language $\mathcal{L}_{\text{SW}}$ (short for $\mathcal{L}_{\text{SnowWhite}}$), against the two variants described in Section 3.7 and the type language of Eklavya [14]. Column $|\mathcal{L}|$ shows the number of unique types in the dataset, if expressed in the respective language. For that, we re-extract samples from the binaries with different configuration settings that map DWARF types to the respective languages.

$\mathcal{L}_{\text{SW}}$ can distinguish 1,225 unique types, far more than the 7 of Eklavya, 11 of TypeMiner [47], 17 of Debin [27], or 35 of StateFormer [57]. Just by removing names, `const`-ness and the distinction between classes and structs, the simplified variant in the third row results in only 120 unique types in the dataset, so our aforementioned type language features are clearly necessary to express many types more precisely. In the "All Names" variant, the large amount of project and domain specific names increases the set of types than 100-fold to 146,883 unique types.

We also check that recursion in our type language is useful and even necessary for many types. Of all type samples expressed in $\mathcal{L}_{\text{SW}}$, only 20.7% do not make use of recursion (e.g., primitive types), 48.3% have one nested type constructor (e.g., `pointer` from Figure 3), and 31% have an even deeper nesting depth of up to six nested type constructors.

Type distribution. As another measure of how informative the type language is, we also inspect the resulting distribution of realized types when converting the samples in the dataset to the language. For brevity, we do not show the full distributions, but summarize two key aspects in Table 4.

First, column H/H_{max} gives the normalized entropy of the type distribution, where $H_{max} = \log_2 |\mathcal{L}|$ is the entropy of a uniform distribution of the same size. If a type distribution is very non-uniform, e.g., if one type is extremely common, less information can be gained from a single predicted type, and H becomes smaller compared to a more ideal, uniform distribution of types. With the normalized entropy, we can compare the entropy of distributions of different size. Evidently, more expressive type languages have not only more types, but are more uniformly distributed as well, as the entropy increases towards the maximum of 1.

Table 4 also shows the most frequent type, separately for parameter and return types, and how much of the overall distribution that type accounts for. For $\mathcal{L}_{\text{Eklavya}}$, the `pointer` label makes up for almost 80% of the data, a very biased type distribution! Simplified $\mathcal{L}_{\text{SW}}$ without names, `const`, and `classes` is similar to the type language used in StateFormer [57], and is only slightly better, as the most common parameter type already accounts for 57% of the data. $\mathcal{L}_{\text{SW}}$ is much more uniform with 22% for the most common parameter type. Interestingly, the return type distribution is less affected by the different languages than the parameter types. Regardless of the type language, the most common label is a primitive integer, accounting for 30% (most expressive language) to 51% (least expressive) of all return types. This

Table 3. Most common extracted type names.

Name	Sample Count	Packages
`size_t`	516,451	63.8 %
`FILE`	20,949	45.2 %
`basic_string<char, ...>`	135,900	17.2 %
`basic_ostream<char, ...>`	35,460	16.3 %
`ios_base`	10,002	16.1 %
`ostreambuf_iterator<char, ...>`	7,801	15.8 %
`va_list`	2,470	15.8 %
`string`	45,081	15.5 %

Table 4. Different type distributions compared.

Type Language	$\lvert\mathcal{L}\rvert$	$\frac{H}{H_{max}}$	Most Frequent Type			
			Parameter		Return	
$\mathcal{L}_{\text{SW}}$, All Names	146,883	0.69	`primitive int 32`	5%	`primitive int 32`	30%
$\mathcal{L}_{\text{SW}}$	1,225	0.49	`pointer class`	22%	`primitive int 32`	39%
$\mathcal{L}_{\text{SW}}$, Simplified	120	0.42	`pointer struct`	57%	`primitive int 32`	41%
$\mathcal{L}_{\text{Eklavya}}$	7	0.38	`pointer`	78%	`int`	51%

may be an artifact of C and C++, where complex results are often written via pointers instead of being returned by value.

Summary. We conclude that all features of our type language, in particular type names, recursion, `const`, and the distinction of `class` vs. `struct` help to distinguish types and avoid a biased type distribution. The extracted names convey useful intuitions and are project and domain independent. In general, parameter types are more sensitive to the type language than return types. While more names can be added to the language to increase the number of unique types, the next section shows that accurate prediction also becomes much harder in that case.

6.3 Type Prediction Model

We now evaluate the accuracy of our type prediction model.

Metrics. To compare a predicted type against the ground truth type, we use two metrics. One is *perfect match accuracy*, i.e., the percentage of all predicted types that exactly match the ground truth. We report perfect match accuracy within the top-1 and the top-5 predictions, where the latter retrieves the five most likely type predictions via beam search.

Perfect match accuracy does not consider partially correct predictions. For example, if the ground truth is `pointer struct`, a prediction of `pointer class` is intuitively better than `primitive int 32`, but since neither are exact matches, they do not count towards accuracy. We thus introduce a metric for type accuracy based on the longest common prefix of the prediction and the ground truth. The *type prefix score* of a prediction t' and ground truth t is the length of the common prefix $TPS(t', t) = |commonPrefix(t', t)|$. That is,

$$TPS(\texttt{pointer struct}, \texttt{pointer class}) = 1, \text{ but}$$
$$TPS(\texttt{pointer struct}, \texttt{primitive int 32}) = 0.$$

Computed over the whole test set, *TPS* gives the average number of type tokens that are correct until the predicted sequence diverges from the ground truth.

Baseline. As there is no existing learning-based type prediction for WebAssembly binaries, we compare our model against a statistical baseline. This baseline exploits that the low-level WebAssembly type t_{low} is available in the binary for each parameter and return sample. Given only the t_{low} of an input, we can "generate" top-k predictions by copying the k most likely high-level types for a given t_{low} from the conditional probability distribution $P(t_{high} \mid t_{low})$ that was empirical observed on the training data. For example, the most common type for t_{low} = `i32` is `pointer class`, and for t_{low} = `f32` it is `primitive float 32`.

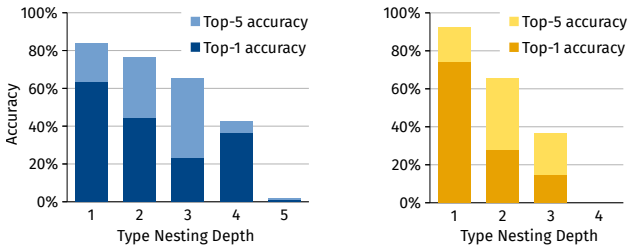
Results. Table 5 shows the model accuracy for parameter and return type prediction separately in the left and right half. For our proposed type language $\mathcal{L}_{\text{SW}}$, we see that the model predicts the exactly correct parameter type in 44.5% of the cases, even though there is a large choice out of 1,225 unique types. If we accept any of the model's top five predictions, the model is right for 75.2% of the test samples. This is also not just because of a skewed data distribution, as the baseline exploiting the underlying data distribution achieves only 28.7% top-1 exact match accuracy, significantly less than the neural model. The model accuracy is even better for return type prediction with a top-1 (top-5) accuracy of 57.7% (80.5%). On average, the type prediction model gets the first 1.47 (1.37) tokens of the type sequence correct for parameter (return) types. We expect the first tokens to be likely the most relevant to a reverse engineer, so this is a good result.

We also judge the hardness of type prediction with different languages. Without filtering names ($\mathcal{L}_{\text{SW}}$, All Names), the task becomes much too difficult, with a top-1 accuracy of only 18.6% on parameter types. This motivates our restriction to a vocabulary of common type names. At the other end, for the simple language $\mathcal{L}_{\text{Eklavya}}$ we achieve a top-1 accuracy of 87.9%, compared with an accuracy of around 81% on native binaries reported in [14]. However, the statistical baseline also casts doubt on whether this is really an achievement of the model, or simply a very easy task, as even the baseline achieves 77.1% top-1 accuracy without any neural network. The model for simplified $\mathcal{L}_{\text{SW}}$ sits between $\mathcal{L}_{\text{SW}}$ and $\mathcal{L}_{\text{Eklavya}}$, with a top-1 accuracy of 65.1% (60.6%) on parameter (return) types. In general, we can conclude that the neural model accuracy is consistent with the complexity of the type language. A good type language for learning-based type prediction must balance the trade-off between being precise and allowing for accurate predictions.

Ablation study and type depth. The rightmost model for both parameter and return type prediction in Table 5 is an ablation study as to how much passing the WebAssembly low-level type t_{low} helps the model to predict high-level types. For that, we take the same language as in $\mathcal{L}_{\text{SW}}$, but

Table 5. Model accuracy on different type prediction tasks, compared with a simple conditional probability baseline.

Task	Parameter Type Prediction					Return Type Prediction				
Type Language	$\mathcal{L}_{\text{SW}}$	$\mathcal{L}_{\text{SW}}$, All Names	$\mathcal{L}_{\text{SW}}$, Simplified	$\mathcal{L}_{\text{Eklavya}}$	$\mathcal{L}_{\text{SW}}$, t_{low} not given	$\mathcal{L}_{\text{SW}}$	$\mathcal{L}_{\text{SW}}$, All Names	$\mathcal{L}_{\text{SW}}$, Simplified	$\mathcal{L}_{\text{Eklavya}}$	$\mathcal{L}_{\text{SW}}$, t_{low} not given
Seq-to-seq Model, see Section 4										
Top-1 Accuracy	44.5%	18.6%	65.1%	87.9%	42.4%	57.7%	40.6%	60.6%	76.3%	50.7%
Top-5 Accuracy	75.2%	27.1%	86.2%	100.0%	73.4%	80.5%	47.3%	87.9%	100.0%	81.2%
Type Prefix Score	1.47	1.31	1.62	0.88	1.45	1.37	1.00	1.38	0.76	1.02
Statistical Baseline, based on $P(t_{high} \mid t_{low})$										
Top-1 Accuracy	28.7%	13.0%	47.1%	77.1%		49.9%	41.7%	50.7%	64.6%	
Top-5 Accuracy	61.4%	20.8%	78.1%	99.9%	N/A	74.2%	48.5%	81.4%	100.0%	N/A
Type Prefix Score	1.05	0.28	1.24	0.77		1.14	0.92	1.16	0.65	



(a) Parameter types. **(b)** Return types.

Figure 4. Prediction accuracy of $\mathcal{L}_{\text{SW}}$ by type nesting depth.

remove the low-level type from the beginning of the input sequence. For parameter types, the low-level type seems to help only marginally (a difference of ≈2% in accuracy), possibly because there are enough cues from the parameter usage even without explicitly passing the low-level type. For return type prediction, the low-level type seems more useful, with an accuracy difference of ≈9%.

Finally, Figure 4 shows the prediction accuracy of $\mathcal{L}_{\text{SW}}$, separately for different type nesting depths. The general trend is that accuracy decreases with more deeply nested types, as expected. However, even for types with three (four) nested levels, parameters can still be predicted exactly with a top-5 accuracy of 65% (43%). Return types are less deeply nested in general and prediction accuracy is also worse beyond types with a single or two nested levels.

6.4 Case Studies of Predictions

We show representative examples of predictions produced by the model, to get an intuition how useful it is in practice.

***Example:* `libgdal`.** For predicting the first parameter sample in the shuffled test set, the $\mathcal{L}_{\text{SW}}$ model is given a WebAssembly input of 8 instruction windows, containing 168 instructions and 453 tokens in total. The input starts with:

```
i32 <begin> global.get 1 ; i32.const 294552 ; i32.add ;
i32.const 3 ; i32.const 1 ; local.get <param> ; call ...
```

From just this input, the model's top five predictions are:

```
pointer name "FILE" struct
pointer struct
primitive int 32
pointer primitive cchar
pointer const primitive cchar
```

The sample corresponds to the fourth parameter[2] `fp` of a class method in `libgdal`, a geospatial library written in C++. The method declaration in the source code reads:

```
void DDFSubfieldDefn::DumpData(
    const char * pachData, int nMaxBytes, FILE * fp ) ...
```

As we can see, the top-most prediction of the model is exactly correct, the parameter is indeed a pointer to a file handle. For a reverse engineer, we believe this prediction in our type language is much more useful than, e.g., `pointer struct` by StateFormer or just `pointer` by Eklavya. The top-2 prediction would not have been incorrect either, just not as precise, justifying our type prefix metric for evaluation.

Staying with the example, the model's predictions for the parameter `nMaxBytes` of the same function are as follows:

```
pointer const primitive uint 8
primitive int 32
pointer const primitive cchar
pointer struct
pointer primitive uint 8
```

Here, the top-1 prediction is not correct, but the top-2 prediction is. It is unclear how the model came up with the top-1 prediction; we can only speculate that it got confused by instructions related to the other parameter, whose string type is closer to the predictions.

***Example:* `libtiff`.** Taking the first return type sample in the test set from the next library, the model attempts to predict the return type of the following function in `libtiff`:

```
int JPEGVGetField(TIFF* tif, uint32 tag, va_list ap) ...
```

The model's top five return type predictions are:

```
primitive int 8
primitive uint 32
primitive uint 32
pointer name "Exception" class
primitive int 32
```

[2]We regard the methods' receiver object as the implicit first parameter.

The correct prediction is on place five. As the raw model is not constrained to generate five unique predictions, we also see two duplicate predictions. In a production-grade type prediction tool, the raw model outputs could be filtered to only include unique types. Interestingly, the body of the function (not shown) returns a literal 1, so the other primitive predictions would have actually been type compatible in C, showing the difficulty in getting accurate training data.

Finally, we inspect the top-most prediction for the first parameter of the same function, where the model returns `pointer struct`. As a domain specific name, `TIFF` is not shared among enough projects to be in our list of common type names. The predicted type is thus correct and as precise as possible as per our type language $\mathcal{L}_{\text{SW}}$. Future work could explore to predict information about the struct fields as well.

7 Related Work

Binary type recovery. Recovering types from binaries has received significant attention, mostly for x86 binaries. Several approaches aim to recover class hierarchies [35, 56] and other kinds of type information [18, 34, 43, 52]. Caballero and Lin [12] provide a good overview of approaches until 2016. More recently, several data-driven and learning-based type prediction approaches have been proposed [14, 27, 47, 57], which we compare against in terms of the set of types that can be predicted (Section 6.2). Like SnowWhite, TypeMiner [47] and StateFormer [57] also discern types of pointers, e.g., "pointer to array", but they do not define a recursive type language, and hence, their types are strictly less expressive than ours. Beyond the expressiveness of our type language, SnowWhite contributes by being the first type prediction approach for WebAssembly binaries.

Type prediction for source languages. Dynamically typed languages also benefit from type prediction, e.g., to automatically add type annotations to source code. There are several approaches for JavaScript [28, 48, 61, 72], Python [5, 59, 73], and Ruby [36]. Most of them focus on how to represent and process the input to a prediction model, e.g., with an RNN over a token sequence [28] or a GNN over a graph representation of code [72]. TypeWriter [59] combines neural type prediction and type checking-based validation. Almost all the above approaches predict types from a fixed set, with the exception of Typilus [5], which represents types as points in a continuous type space. SnowWhite differs by representing types as sentences in a type language, which turns type prediction into a sequence prediction task.

Reverse engineering. Beyond types, other information about a binary can also be predicted to support reverse engineering. DIRE [42] and Nero [15] are neural models to predict names of variables and functions, respectively. Coda [20] is a trained model that predicts an AST for code given in a simple assembly language. For WebAssembly, `wasm-decompile` and `wasm2c` from the official WebAssembly Binary Toolkit (WABT) aim to decompile binaries to more readable pseudo code or proper C, respectively [1]. All the above tools and techniques are complementary to recovering types.

Neural models of code. Deep learning on code is receiving significant interest [58] beyond the work discussed above. One important question is how to represent a piece of code, e.g., using AST paths [8], control flow graphs [71], ASTs [74], or a combination of token sequences and a graph representation of code [29]. Instead of the input representation, this work focuses on how to represent the type *output* of a predictive model. Other neural models are applied to predicting code changes [11, 31] or program repairs [16, 24], complete partial code [7, 38], or detects bugs [60].

WebAssembly. Since its inception [25], WebAssembly has been subject to studies about its security [41, 44, 50], performance [32], and use in practice [30]. Techniques to analyze and improve WebAssembly code include a general purpose dynamic analysis framework [45], taint analyses [21, 67], and a compiler framework for hardening WebAssembly against Spectre attacks [53]. To the best of our knowledge, no prior work addresses the problem of recovering precise, high-level types in WebAssembly binaries. In particular, `wasm2c`, despite the name, does not recover high-level source types. Instead, the low-level WebAssembly types in the binary are merely translated to matching C typedefs.

8 Conclusion

This paper presents the first learning-based approach for recovering precise types in WebAssembly binaries. In contrast to prior work on learning-based binary type prediction, we represent types through an expressive type language. The language allows for thousands of different types, instead of the 7 to 35 types considered previously. Despite this increase of expressiveness, we find our type prediction model to be highly accurate, exactly predicting 44.5% (75.2%) of all parameter types and 57.7% (80.5%) of all return types within the top-1 (top-5) predictions. SnowWhite is an important first step toward reverse engineering WebAssembly binaries. Beyond our technique, we share a novel large-scale dataset of C and C++ code compiled to WebAssembly with debug information, which is two orders of magnitude larger than existing datasets for WebAssembly. Finally, we envision the idea of formulating type prediction as a sequence prediction task to be potentially useful also for other languages.

Acknowledgments

We thank the reviewers and our shepherd, Justin Gottschlich, for their helpful feedback on an earlier version of this paper.

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and Perf4JS projects.

References

[1] 2021. *WABT: The WebAssembly Binary Toolkit.* https://github.com/WebAssembly/wabt

[2] Toufique Ahmed, Vincent Hellendoorn, and Premkumar T. Devanbu. 2019. Learning Lenient Parsing & Typing via Indirect Supervision. *CoRR* abs/1910.05879 (2019). arXiv:1910.05879 http://arxiv.org/abs/1910.05879

[3] Georgi Geshev Alex Plaskett, Fabian Beterke. 2018. *Apple Safari – Wasm Section Exploit.* https://labs.f-secure.com/assets/BlogFiles/apple-safari-wasm-section-vuln-write-up-2018-04-16.pdf

[4] Miltiadis Allamanis. 2019. The Adverse Effects of Code Duplication in Machine Learning Models of Code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019)*. 143–153. https://doi.org/10.1145/3359591.3359735

[5] 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 2020)*. 91–105. https://doi.org/10.1145/3385412.3385997

[6] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations (ICLR 2018)*. https://openreview.net/forum?id=BJOFETxR-

[7] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *7th International Conference on Learning Representations (ICLR 2019)*. https://openreview.net/forum?id=H1gKYo09tX

[8] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[9] Shun-ichi Amari, Noboru Murata, Klaus-Rovert Muller, Michael Finke, and Howard Hua Yang. 1997. Asymptotic Statistical Theory of Overtraining and Cross-Validation. *IEEE Transactions on Neural Networks* 8, 5 (1997), 985–996. https://doi.org/10.1109/72.623200

[10] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations (ICLR 2015)*. http://arxiv.org/abs/1409.0473

[11] Shaked Brody, Uri Alon, and Eran Yahav. 2020. A structural model for contextual code changes. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 215:1–215:28. https://doi.org/10.1145/3428283

[12] Juan Caballero and Zhiqiang Lin. 2016. Type Inference on Executables. *ACM Comput. Surv.* 48, 4 (2016), 65:1–65:35. https://doi.org/10.1145/2896499

[13] Zimin Chen, Vincent Josua Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Manzagol, Daniel Tarlow, and Subhodeep Moitra. 2021. PLUR: A Unifying, Graph-Based View of Program Learning, Understanding, and Repair. In *Advances in Neural Information Processing Systems (NeurIPS 2021)*. https://openreview.net/forum?id=GEm4o9A6Jfb

[14] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural Nets Can Learn Function Type Signatures From Binaries. In *26th USENIX Security Symposium (USENIX Security 17)*. 99–116. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/chua

[15] Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural Reverse Engineering of Stripped Binaries Using Augmented Control Flow Graphs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 225:1–225:28. https://doi.org/10.1145/3428293

[16] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *8th International Conference on Learning Representations (ICLR 2020)*. https://openreview.net/forum?id=SJeqs6EFvB

[17] DWARF Committee. 2017. DWARF Debugging Information Format – Version 5. http://www.dwarfstd.org/doc/DWARF5.pdf

[18] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. Scalable Variable and Data Type Detection in a Binary Rewriter. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. 51–60. https://doi.org/10.1145/2491956.2462165

[19] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. EMNLP 2020 (2020), 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[20] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. Coda: An End-to-End Neural Program Decompiler. In *Advances in Neural Information Processing Systems*, Vol. 32. https://proceedings.neurips.cc/paper/2019/file/093b60fd0557804c8ba0cbf1453da22f-Paper.pdf

[21] William Fu, Raymond Lin, and Daniel Inge. 2018. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly. *CoRR* abs/1802.01050 (2018). http://arxiv.org/abs/1802.01050

[22] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. Incorporating Copying Mechanism in Sequence-to-Sequence Learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (ACL 2016)*. 1631–1640. https://doi.org/10.18653/v1/P16-1154

[23] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*. https://openreview.net/forum?id=jLoC4ez43PZ

[24] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. 1345–1351. http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[25] 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 2017)*. 185–200. https://doi.org/10.1145/3062341.3062363

[26] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. 2014. An Empirical Study on the Impact of Static Typing on Software Maintainability. *Empir. Softw. Eng.* 19, 5 (2014), 1335–1382. https://doi.org/10.1007/s10664-013-9289-1

[27] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting Debug Information in Stripped Binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. 1667–1680. https://doi.org/10.1145/3243734.3243866

[28] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. 152–162. https://doi.org/10.1145/3236024.3236051

[29] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global Relational Models of Source Code. In *8th International Conference on Learning Representations (ICLR 2020)*. https://openreview.net/forum?id=B1lnbRNtwr

[30] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference 2021 (WWW '21)*. 2696–2708. https://doi.org/10.1145/3442381.3450138

[31] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed Representations of Code Changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. 518–529. https://doi.org/10.1145/3377811.3380361

[32] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 107–120. https://www.usenix.org/conference/atc19/presentation/jangda

[33] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. 1073–1085. https://doi.org/10.1145/3377811.3380342

[34] Omer Katz, Ran El-Yaniv, and Eran Yahav. 2016. Estimating Types in Binaries Using Predictive Modeling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. 313–326. https://doi.org/10.1145/2837614.2837674

[35] Omer Katz, Noam Rinetzky, and Eran Yahav. 2018. Statistical Reconstruction of Class Hierarchies in Binaries. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. 363–376. https://doi.org/10.1145/3173162.3173202

[36] Milod Kazerounian, Brianna M. Ren, and Jeffrey S. Foster. 2020. Sound, Heuristic Type Annotation Inference for Ruby. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages (DLS 2020)*. 112–125. https://doi.org/10.1145/3426422.3426985

[37] Amin Kharraz, Zane Ma, Paul Murley, Charles Lever, Joshua Mason, Andrew Miller, Nikita Borisov, Manos Antonakakis, and Michael Bailey. 2019. Outguard: Detecting In-Browser Covert Cryptocurrency Mining in the Wild. In *Proceedings of the 2019 World Wide Web Conference (WWW '19)*. 840–852. https://doi.org/10.1145/3308558.3313665

[38] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code Prediction by Feeding Trees to Transformers. In *43rd IEEE/ACM International Conference on Software Engineering (ICSE 2021)*. 150–162. https://doi.org/10.1109/ICSE43902.2021.00026

[39] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations (ICLR 2015)*. http://arxiv.org/abs/1412.6980

[40] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander Rush. 2017. OpenNMT: Open-Source Toolkit for Neural Machine Translation. In *Proceedings of ACL 2017, System Demonstrations*. 67–72. https://www.aclweb.org/anthology/P17-4012

[41] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. 2018. 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 (CCS '18)*. 1714–1730. https://doi.org/10.1145/3243734.3243858

[42] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A Neural Approach to Decompiled Identifier Naming. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*. 628–639. https://doi.org/10.1109/ASE.2019.00064

[43] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2011)*. https://www.ndss-symposium.org/ndss2011/tie-principled-reverse-engineering-of-types-in-binary-programs

[44] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 2020)*. 217–234. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[45] Daniel Lehmann and Michael Pradel. 2019. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 1045–1058. https://doi.org/10.1145/3297858.3304068

[46] Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP 2015)*. 1412–1421. https://doi.org/10.18653/v1/D15-1166

[47] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. 2019. TypeMiner: Recovering Types in Binary Programs Using Machine Learning. In *Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2019. (Lecture Notes in Computer Science, Vol. 11543)*. 288–308. https://doi.org/10.1007/978-3-030-22038-9_14

[48] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering (ICSE 2019)*. 304–315. https://doi.org/10.1109/ICSE.2019.00045

[49] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. 2012. An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. 683–702. https://doi.org/10.1145/2384616.2384666

[50] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2019 (Lecture Notes in Computer Science, Vol. 11543)*. 23–42. https://doi.org/10.1007/978-3-030-22038-9_2

[51] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. Thieves in the Browser: Web-Based Cryptojacking in the Wild. In *Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES '19)*. Article 4, 10 pages. https://doi.org/10.1145/3339252.3339261

[52] Alan Mycroft. 1999. Type-Based Decompilation (or Program Reconstruction via Type Reconstruction). In *Proceedings of the 8th European Symposium on Programming Languages and Systems (ESOP '99)*. 208–223. https://link.springer.com/content/pdf/10.1007/3-540-49099-X_14.pdf

[53] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean M. Tullsen, and Deian Stefan. 2021. Swivel: Hardening WebAssembly against Spectre. In *30th USENIX Security Symposium (USENIX Security 2021)*. 1433–1450. https://www.usenix.org/conference/usenixsecurity21/presentation/narayan

[54] Matt Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic Type Inference for Machine Code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. 27–41. https://doi.org/10.1145/2908080.2908119

[55] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the Difficulty of Training Recurrent Neural Networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28 (ICML'13)*. III–1310–III–1318. https://proceedings.mlr.press/v28/pascanu13.pdf

[56] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. 2017. MARX: Uncovering Class Hierarchies in C++ Programs.. In *24th Annual Network and Distributed System Security Symposium (NDSS 2017)*. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/marx-uncovering-class-hierarchies-c-programs/

[57] 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 '21)*. 690–702. https://doi.org/10.1145/3468264.3468607

[58] Michael Pradel and Satish Chandra. 2021. Neural Software Analysis. *Commun. ACM* 65, 1 (Dec. 2021), 86–96. https://doi.org/10.1145/3460348

[59] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-based Validation. In *28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*. 209–220. https://doi.org/10.1145/3368089.3409715

[60] Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-Based Bug Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 147 (Oct. 2018), 25 pages. https://doi.org/10.1145/3276517

[61] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. 111–124. https://doi.org/10.1145/2676726.2677009

[62] Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. 2022. Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly. In *Proceedings of the 2022 IEEE Symposium on Security and Privacy (S&P 2022)*. 1101–1116. https://doi.org/10.1109/SP46214.2022.00064

[63] Jan Rüth, Torsten Zimmermann, Konrad Wolsing, and Oliver Hohlfeld. 2018. Digging into Browser-Based Crypto Mining. In *Proceedings of the Internet Measurement Conference 2018 (IMC '18)*. 70–76. https://doi.org/10.1145/3278532.3278539

[64] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1715–1725. https://doi.org/10.18653/v1/P16-1162

[65] Natalie Silvanovich. 2018. *The Problems and Promise of WebAssembly*. https://googleprojectzero.blogspot.com/2018/08/the-problems-and-promise-of-webassembly.html

[66] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 56 (2014), 1929–1958. http://jmlr.org/papers/v15/srivastava14a.html

[67] Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. 2018. Taint Tracking for WebAssembly. *CoRR* abs/1807.08349 (2018). arXiv:1807.08349 http://arxiv.org/abs/1807.08349

[68] Said Varlioglu, Bilal Gonen, Murat Ozer, and Mehmet Bastug. 2020. Is Cryptojacking Dead after Coinhive Shutdown?. In *2020 3rd International Conference on Information and Computer Technologies (ICICT)*. 385–389. https://doi.org/10.1109/ICICT50521.2020.00068

[69] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems (NIPS 2017, Vol. 30)*. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[70] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W Hamlen, and Shuang Hao. 2018. SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks. In *Computer Security - 23rd European Symposium on Research in Computer Security. ESORICS 2018 (Lecture Notes in Computer Science, Vol. 11099)*. 122–142. https://doi.org/10.1007/978-3-319-98989-1_7

[71] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning Semantic Program Embeddings with Graph Interval Neural Network. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 137, 27 pages. https://doi.org/10.1145/3428205

[72] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *8th International Conference on Learning Representations (ICLR 2020)*. https://openreview.net/forum?id=Hkx6hANtwH

[73] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python Probabilistic Type Inference with Natural Language Support. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. 607–618. https://doi.org/10.1145/2950290.2950343

[74] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. 783–794. https://doi.org/10.1109/ICSE.2019.00086

[75] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. 995–1010. https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman