Automated WebAssembly Function Purpose Identification

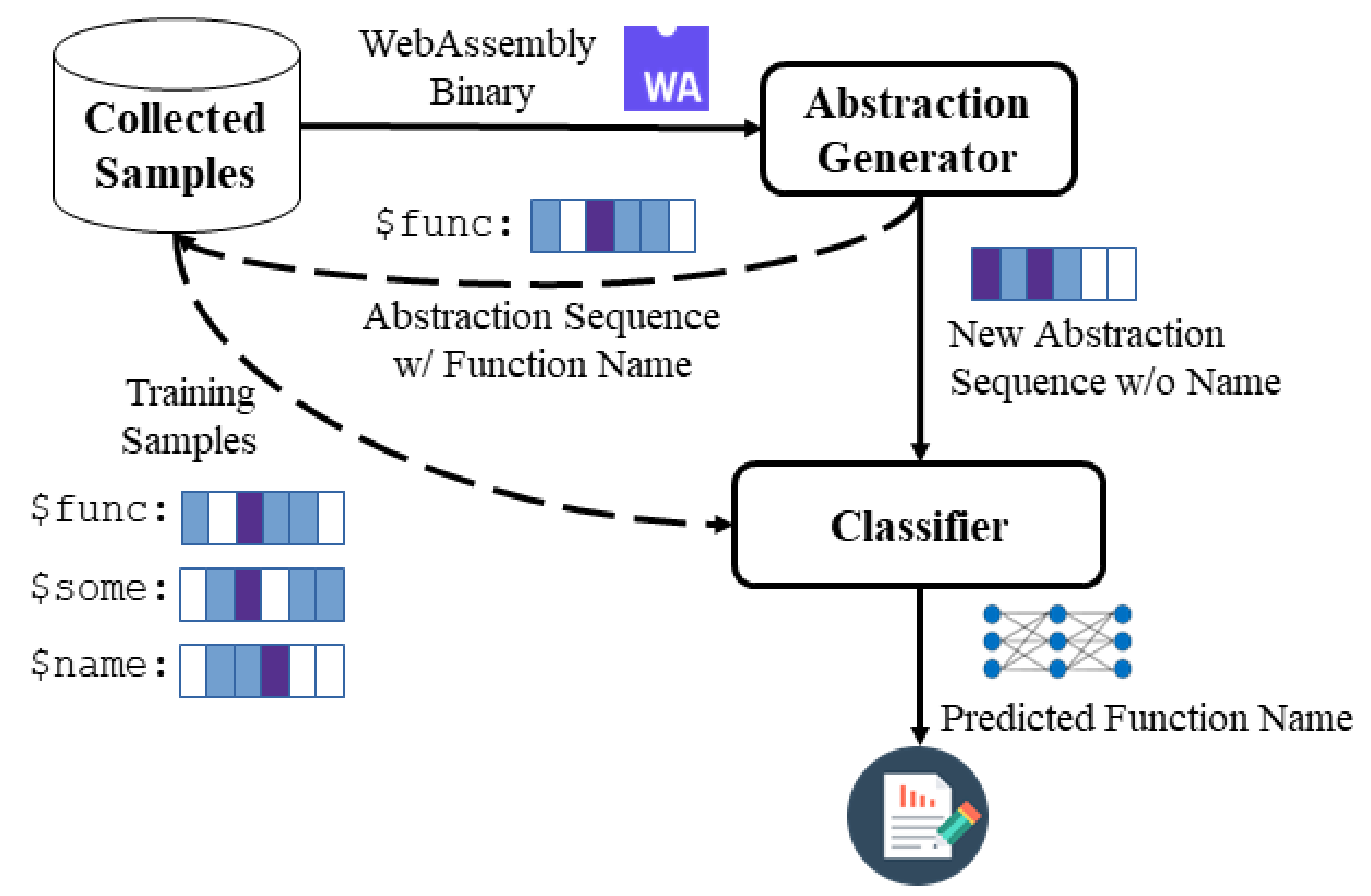
Alan Romano and Weihang Wang
University of Southern California

Introduction

WebAssembly (abbreviated Wasm) is the newest web standard and defines compact bytecode format to serve as a compilation target for other languages such as C, C++, and Rust. WebAssembly also defines a text format meant to ease understanding for debugging, but the low-level structure and terse syntax of the language make it challenging to understand. To aid developers in understanding the functionalities of WebAssembly modules, we construct WAspur, a tool to automatically identify the purposes of WebAssembly functions. Specifically, we construct and analyze an extensive and diverse collection of WebAssembly modules. We then construct semantics-aware intermediate representations (IR) of the functions. Finally, we encode the function IR for use in a machine learning classifier. We hope our tool will enable efficient inspection of optimized and minified WebAssembly modules.

WAspur

WAspur leverages a semantics-resilient intermediate representation (IR) designed to capture the effects produced by WebAssembly instructions. WASPur classifies the functions in a WebAssembly module using two main components. The **Abstraction Generator** collects the abstractions for all functions within the module to represent each function in our IR. The **Classifier** inputs the sequence of IR units into a neural network classifier.



Data Collection Methods

In order to automatically identify and classify WebAssembly function purposes, we construct a dataset of WebAssembly modules from various sources to train the classifier.

Source	# of Samples Inspected	# of Wasm Samples
Websites	1,000,000	4,520
Chrome Extensions	17,862	90
Firefox Add-ons	16,385	43
GitHub	112,663,634	2,116
Total	**113,697,881**	**6,769**

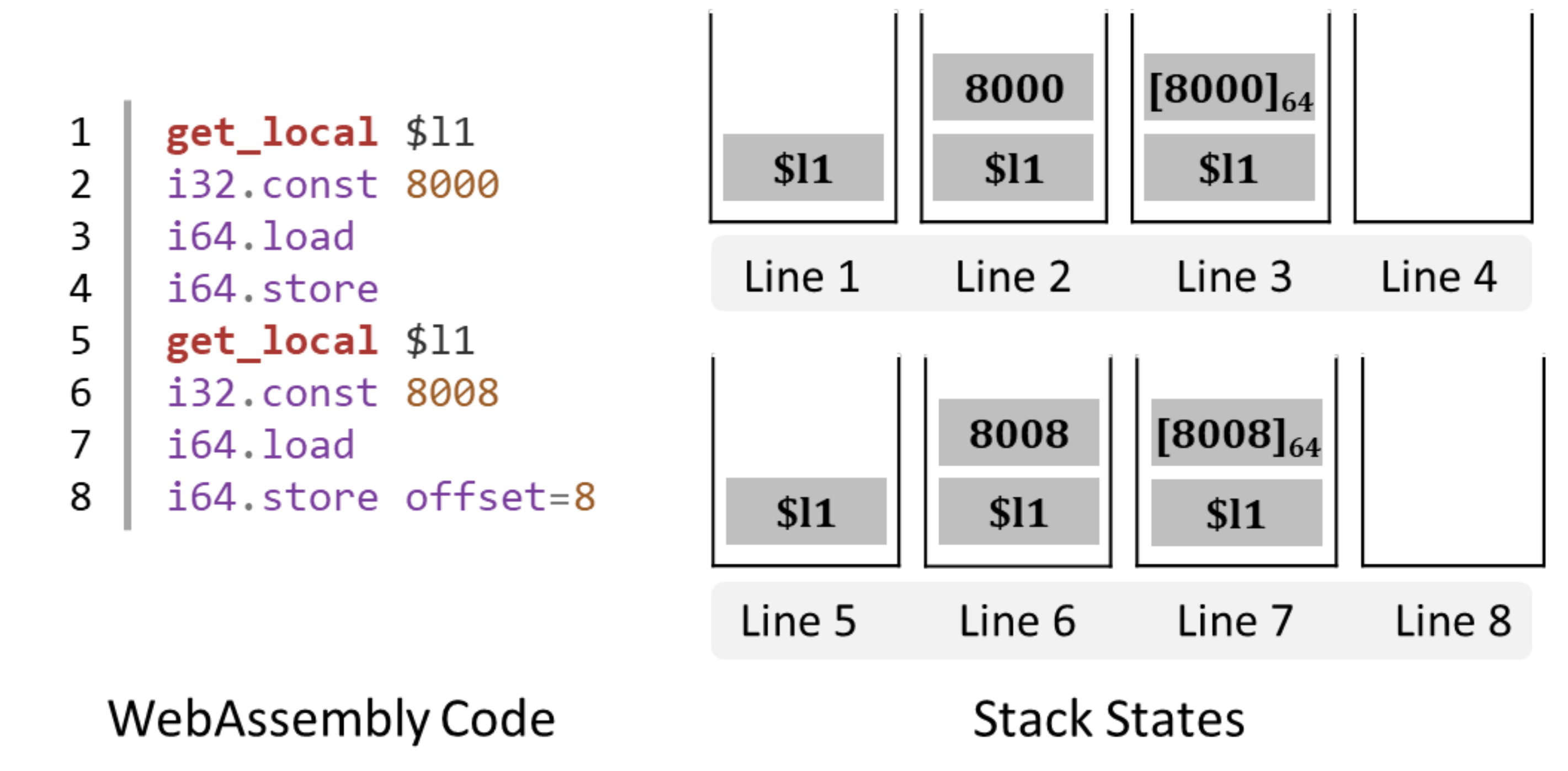
Abstraction Generator

Our abstractions model WebAssembly stack, linear memory, and control flow.

Variable Instructions We model the instructions that store values into variables abstractions that record the value and location stored. We model the instructions that load values from variables through their effect on the virtual stack. Symbolic execution models the effects as parametric expressions with variable names and referenced memory locations as parameters.

Control Instructions We use separate abstractions to model the instructions that impact the control flow, such as if, loop, block, and call. We also model the change in scope caused by the control flow constructions by storing subsequent abstractions within them.

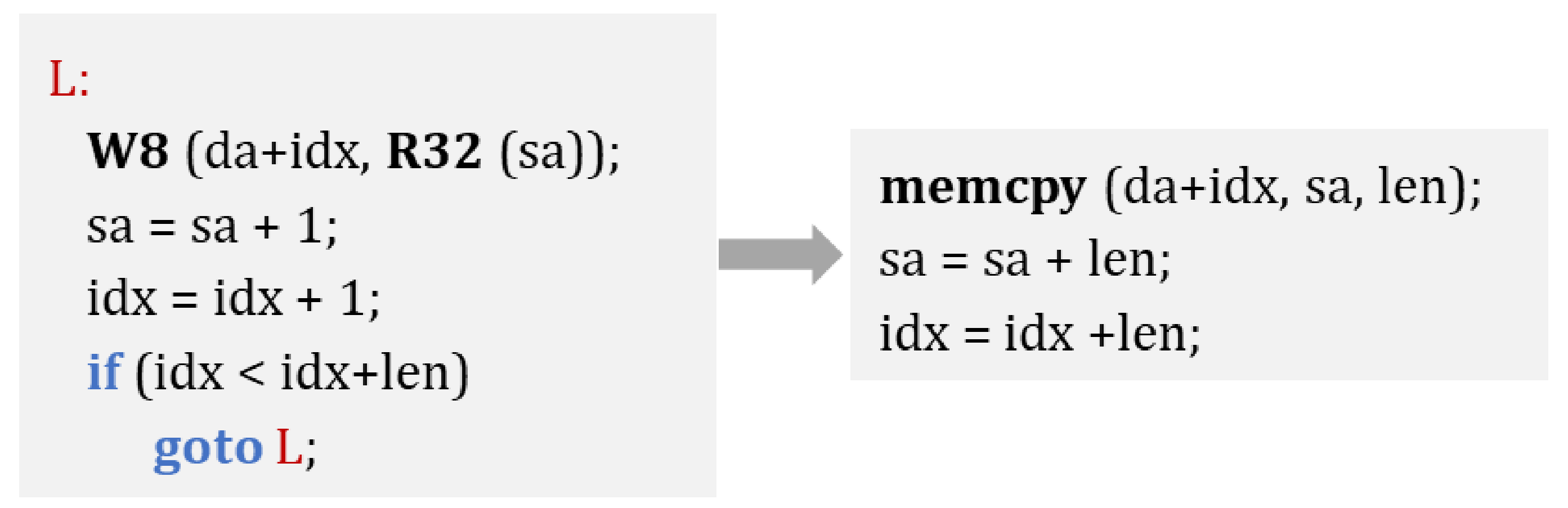
Memory Instructions We model the instructions that load and store into linear memory using similar abstractions as the variable instructions. Since WebAssembly applications frequently use consecutive memory copies and accesses, we merge adjacent store abstractions to highlight this functionality.



```
1   get_local $l1
2   i32.const 8000
3   i64.load
4   i64.store
5   get_local $l1
6   i32.const 8008
7   i64.load
8   i64.store offset=8
```

WebAssembly Code

Stack States



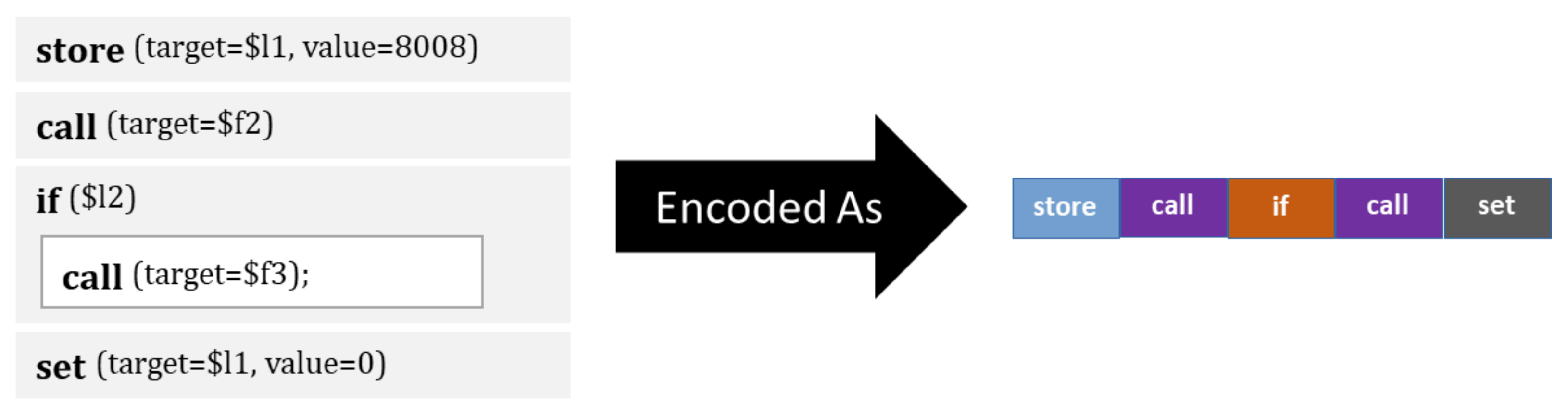
```
L:
  W8 (da+idx, R32 (sa));
  sa = sa + 1;
  idx = idx + 1;
  if (idx < idx+len)
     goto L;
```

```
memcpy (da+idx, sa, len);
sa = sa + len;
idx = idx +len;
```

Merge Consecutive Writes in a Loop

Classifier

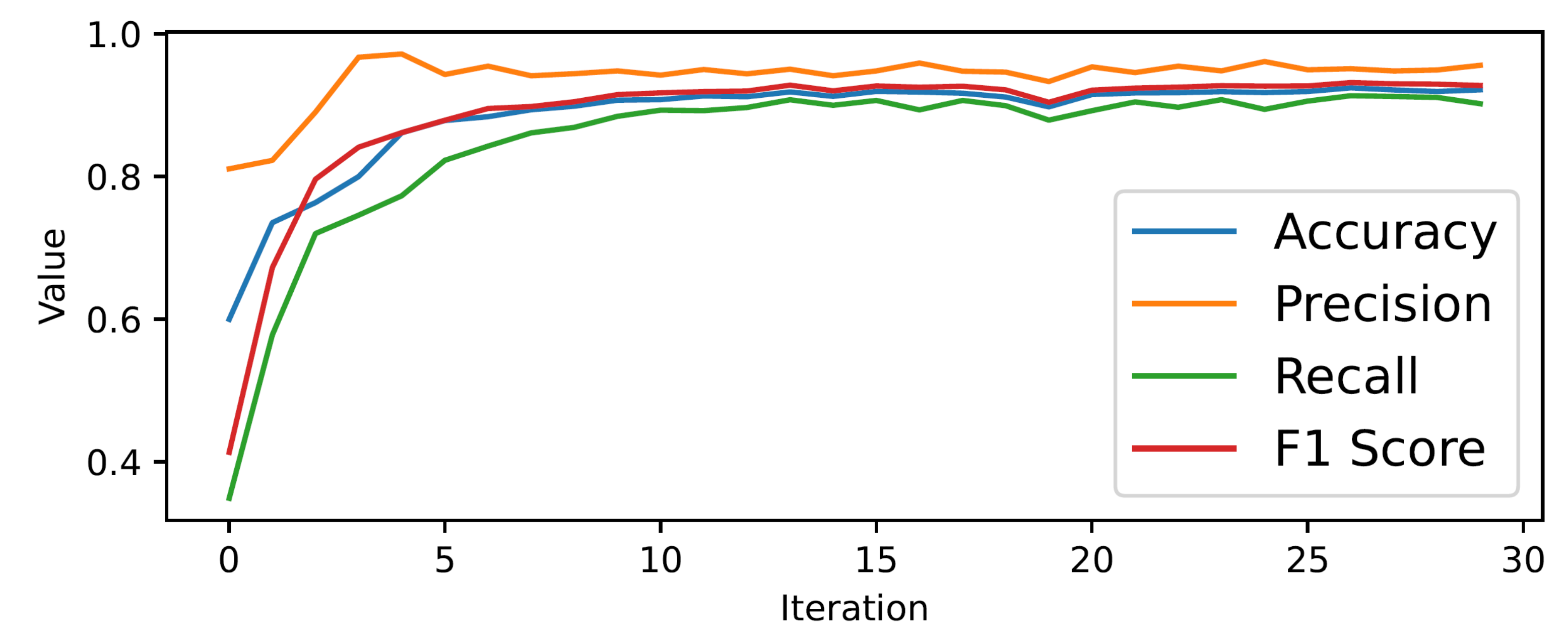
We use the abstractions obtained for each WebAssembly function to train a neural network classifier on the module purposes. To input into the neural network, we encode the abstraction as a sequence produced when traversing the abstractions for a function.



```
store (target=$l1, value=8008)
call (target=$f2)
if ($l2)
  call (target=$f3);
set (target=$l1, value=0)
```

Classification Results

The classifier can predict the similarity of given function against known names with an accuracy rate of 87.4%.



Using the classifier, a WebAssembly function can be used to categorize the originating module from one of 12 use case categories:

Use Case	Purpose
Compression	Performs data compression operations.
Cryptography	Performs cryptographic operations (e.g., hashing).
Game	Implements stand-alone online games.
Text Processing	Performs text or word processing.
Image Processing	Analyzes or edits images.
Numeric Processing	Provides mathematical or numeric functions.
Support Test Stub	Probes environment for WebAssembly support.
Standalone Apps	Independent standalone programs.
Auxiliary Library	Provides data structures or utility functions.
Cryptominer	Performs cryptocurrency-mining operations.
Code Carrier	Stores JavaScript/CSS/HTML payloads.
Unit Test	Ensures conformance to language specification.

Future Work

Currently, WAspur only uses the IR of the instructions defined within WebAssembly functions. We can later include other information from the remaining WebAssembly module sections as well. For example, including the function types within the abstraction sequence may improve predictions.

Conclusion

- We propose an intermediate representation to abstract semantics of WebAssembly applications that enables syntax-resilient analysis.
- We construct a diverse dataset of WebAssembly samples from real-world websites, Firefox add-ons, Chrome extensions, and GitHub repositories.
- We perform a comprehensive analysis of the collected WebAssembly samples to classify them into 10 categories.
- We develop an automated classification tool, WAspur, that can accurately label a given WebAssembly function with a function name indicative of to its functionality and find it can achieve an 87.4% accuracy rate.