Bachelor's Thesis Computing Science

Decompilation of Kasada's JavaScript Virtual machine

Patrick van den Bosch
s1072113

January 6, 2025

First supervisor/assessor:
dr. M. Lubbers (Mart)

Second assessor:
prof. dr. S.-B. Scholz (Sven-Bodo)

Radboud University

Abstract

This thesis presents a decompiler for Kasada's JavaScript VM, the part of Kasada's bot protection system that obfuscates client-side code through bytecode execution in a custom VM. While traditional JavaScript obfuscation techniques can be analyzed using existing deobfuscation tools, VM-based scripts require a different approach due to their additional layer of abstraction and dynamic components. These dynamic components include varying instruction sets, changing bytecode parsing mechanisms, and shifting control signal mappings between different VM instances. We develop a decompilation pipeline consisting of three main phases: parsing the VM's dynamic components, decompiling the bytecode, and generating equivalent JavaScript code. Our approach demonstrates that classical decompilation techniques can be successfully adapted to handle modern VM-based JavaScript obfuscation.

Contents

Acronyms **3**

1 Introduction **4**

2 Preliminaries **6**
2.1 JavaScript Obfuscation 6
2.2 Virtual Machines in JavaScript 7
2.3 Kasada's Bot Protection 9
2.4 Decompilation Concepts 10
2.5 Graph Theory in Decompilation 11

3 Parsing the Interpreter **13**
3.1 Structure of Kasada's Virtual Machine 13
3.2 Challenges in Parsing the Interpreter 13
3.3 Decoding the Bytecode String 14
3.4 Bytecode Parsing Function and Control Signals 15
3.5 Parsing Approach Using Speedy Web Compiler 18
3.6 Opcode Parsing 18

4 Disassembly: From Bytecode to Basic Blocks **20**
4.1 Intermediate Representation Design 20
4.2 Traversal Strategy 21
4.3 Basic Block and Control Flow 21
4.4 Expressions 23
4.5 Exception Handling 23

5 Constructing Control Flow Graphs **25**
5.1 Nodes 25
5.2 Edges 26
5.3 Example Control Flow Graph 26

6 Control Flow Analysis **29**
6.1 Structuring Loops 29
6.2 Structuring Exception Handlers 30
6.3 Structuring Conditionals 32

7 Code Generation **34**
7.1 Graph Traversal Strategy 34
7.2 Structure-Based Code Generation 35
7.3 Basic Block Translation 35
7.4 Scope Management 35
7.5 Code Generation Example 36

8 Related Work **38**
8.1 JavaScript Deobfuscation Tools 38
8.2 VM-Based Obfuscation in JavaScript 38
8.3 Existing Decompiler Research 39
8.4 Control Flow Analysis Techniques 39
8.5 Research Gaps and Novel Contributions 40

9 Conclusions **42**
9.1 Summary of Research Contributions 42
9.2 Evaluation of Research Objectives 42
9.3 Limitations and Future Work 43
9.4 Future Research Directions 43
9.5 Concluding Remarks 43

Acronyms

API Application Programming Interface. 9

AST Abstract Syntax Tree. 18, 19

BFS Breadth First Search. 21

CFA Control Flow Analysis. 5, 10

CFG Control Flow Graph. 5, 10, 11, 12, 25

DFA Data Flow Analysis. 10, 43

DFS Depth First Search. 34

DOM Document Object Model. 9

FFI Foreign Function Interface. 9

IR Intermediate Representation. 1, 10, 20, 22

SWC Speedy Web Compiler. 18

VM Virtual Machine. 4, 5, 6, 7, 8, 9, 13, 14, 18, 19, 20, 21, 23, 30, 35, 36, 38, 39, 40, 41, 42, 43

Chapter 1

Introduction

In recent years, protecting web applications from automated threats has become a significant challenge. Bot protection systems play an important role in distinguishing legitimate users from automated scripts. Kasada is one of these protection systems and employs sophisticated techniques to detect and prevent bot attacks and online fraud.

A key component of Kasada's protection mechanism is its use of JavaScript obfuscation through a custom Virtual Machine (VM). Unlike traditional JavaScript obfuscation techniques that rely on code transformation and minification, Kasada's approach involves implementing a complete VM that executes custom bytecode within the client's browser. This adds a significant layer of complexity for potential attackers, as understanding the protected code requires understanding both the VM's architecture and the bytecode it executes.

Kasada's VM-based approach makes traditional analysis techniques insufficient, as they cannot directly interpret the custom bytecode or understand the VM's execution model. The challenge is further complicated by the dynamic nature of the VM, where components like instruction sets and parsing mechanisms change between browser sessions.

The goal of this thesis is to develop a decompiler capable of transforming Kasada's VM bytecode into readable JavaScript code. This involves several steps: analyzing the structure of the VM to understand its components, developing methods to handle its dynamic aspects, implementing decompilation techniques for the bytecode, and generating equivalent JavaScript code.

While we present detailed technical analysis of the VM's architecture and decompilation techniques, we maintain responsible disclosure practices and avoid releasing sensitive implementation details that could be misused. The remainder of this thesis is organized as follows: chapter 2 provides the necessary background information on JavaScript obfuscation, virtual machines, and decompilation concepts. Chapter 3 details our analysis of

Kasada's VM structure. Chapter 4 and chapter 5 present our approach to disassembly and Control Flow Graph (CFG) construction. Chapter 6 covers Control Flow Analysis (CFA) techniques. Chapter 7 describes the code generation process. Chapter 8 discusses related work and chapter 9 presents our conclusions and future work.

Chapter 2

Preliminaries

This chapter introduces the fundamental concepts and background knowledge required to understand JavaScript VM decompilation. We begin by examining JavaScript obfuscation techniques and their evolution, followed by an exploration of virtual machines implemented in JavaScript. We then discuss Kasada's specific approach to bot protection using VM-based obfuscation. The chapter concludes with an overview of decompilation concepts and the graph theory principles that underpin our analysis techniques. These foundations are essential for understanding the decompilation approach presented in subsequent chapters.

2.1 JavaScript Obfuscation

JavaScript obfuscation is a technique that emerged shortly after the introduction of JavaScript in 1995. Rauti and Leppänen [2018] define obfuscation as the process of transforming source or machine code to obscure its intended meaning, deliberately making it difficult to understand and analyze. This approach aims to deter reverse engineers and preventing unauthorized tampering of code.

The main purposes of obfuscation include protecting intellectual property, enhancing security through obscurity, and enforcing software licenses. As JavaScript is typically executed on the client side in web browsers, the source code is visible to users, making obfuscation relevant for web-based software.

Collberg et al. [1997] outlines three main types of obfuscation transformations, namely: control obfuscation, data obfuscation, and layout obfuscation. Control obfuscation attempts to obfuscate the program's control flow, using techniques such as dead-code insertion, control-flow flattening and altering loop structures. Data obfuscation focuses on obfuscating data structures within the source code, including methods such as encoding variables and string concealing. Layout obfuscation, the simplest form, involves

techniques such as renaming identifiers and minification.

The effectiveness of obfuscation can be evaluated using several metrics. Rauti and Leppänen [2018] discuss three key properties: resilience, potency, and cost. Resilicience measures how well the obfuscated code resists automated deobfuscation tools. Potency refers to the degree of confusion experienced by developers attemping to decipher the obfuscated code. Cost considers the overhead added to the application as a result of obfuscation, such as increases in code size and execution time.

While obfuscation can hinder reverse engineering efforts, it does not provide absolute protection. As Rauti and Leppänen [2018] point out, code on the client side can always be deobfuscated with enough effort. Moreover, within the JavaScript engine, the code is executed in more or less plain form, making it vulnerable to dynamic analysis. Therefore, obfusation should be viewed as a mitigation technique that increases the difficulty and time required for attacks, rather than a bulletproof security measure.

Kasada employs a different approach to code obfuscation that goes beyond the aforementioned obfuscation techniques. Kasada has implemented virtualization obfuscation by developing its own VM that runs within the JavaScript environment of the browser. This VM executes custom bytecode, which represents the obfuscated form of the original application logic.

The use of a custom VM for obfuscation offers significant advantages in terms of resilience and potency. The resilience is high as the bytecode is resistant against automated deobfuscation attacks. Understanding and reversing the code requires not only deciphering the bytecode but also comprehending the entire VM architecture. That makes the VM virtually indecipherable by human readers without knowledge of the VM's inner workings.

The cost of virtualization obfuscation is high. The implementation of a full VM that runs within the JavaScript environment adds significant overhead in terms of code size and execution time. The original application logic is not only obfuscated but also interpreted by an additional layer, the VM, which impacts performance.

2.2 Virtual Machines in JavaScript

2.2.1 Concept of JavaScript-based Virtual Machines

JavaScript-based VMs are software implementations that emulate a computer system within the JavaScript runtime environment. These VMs allow for the execution of code that is not native to the JavaScript language, providing a layer of abstraction between the code and the underlying system.

JavaScript VMs can be designed with different architectures, commonly either stack-based or register-based. While stack-based VMs operate by pushing and popping values from a stack, register-based VMs use a set of registers for storing and manipulating data. Register-based VMs often

provide more efficient execution as they require fewer instructions to perform operations, reducing the overhead of stack management.

2.2.2 Use of Virtual Machines for Code Obfuscation

VMs in JavaScript can be used as a powerful obfuscation method. By implementing a custom VM, developers can compile their original code into bytecode that can only be executed within the context of their VM. This approach adds a layer of complexity for anyone attempting to reverse engineer or understand the code.

The obfuscation process typically involves several steps: A custom instruction set for the VM must be designed, then the original code must be translated into custom bytecode, and finally the VM must be implemented in JavaScript so that it can interpret and execute the bytecode.

This method of obfucation is effective because it not only hides the original code but also requires an attacker to understand the entire VM architecture to make sense of the obfuscated code. Bhansali et al. [2022] note that this approach, talking specifically about WebAssembly, scores exceptionally well in terms of resilience against automated deobfuscation attacks and potency in confusing human readers.

2.2.3 General Structure and Components of a JavaScript Virtual Machine

A JavaScript-based VM typically consists of several key components:

- **Bytecode**: The low-level code that represents the program to be executed.
- **Interpreter**: Reads and executes the bytecode instructions. It maintains the state of the VM and performs operations based on the bytecode instructions.
- **Memory Management**: The VM needs to manage its own memory space, separate from the JavaScript heap, including allocating and deallocating memory as needed by the executing program.
- **Instruction Set**: The set of operations that the VM can perform. Each instruction in the bytecode corresponds to one of these operations.
- **Register Set or Stack**: Depending on the VM architecture, it will either use a set of registers or a stack for temporary storage and passing arguments to functions, or both.

- **Interface with JavaScript**: The VM needs to provide methods for JavaScript to initialize the VM, load bytecode, and interact with the executing program.

- **Foreign Function Interface (FFI)**: A mechanism that allows the VM to call native JavaScript functions. This is crucial for integrating the VM's functionality with the host JavaScript environment and leveraging existing JavaScript capabilities.

In the case of a register-based VM, the execution engine manipulates data primarily through a set of virtual registers. Shi et al. [2008] show that this approach leads to more compact bytecode and potentially faster execution compared to stack-based alternatives.

The VM's interface with JavaScript and its FFI are required to integrate the VM within the broader JavaScript environment, allowing for bidirectional interaction between the VM and the host JavaScript context. The FFI in particular enables the VM to utilize JavaScript's built-in functions and access the Document Object Model (DOM) or other Web Application Programming Interface (API)s when necessary, bridging the gap between the isolated VM environment and the full capabilities of the JavaScript runtime.

2.3 Kasada's Bot Protection

Kasada's bot protection mechanism relies on a JavaScript VM script that operates in two main stages:

2.3.1 Signal Collection and Token Generation

When a user visits a protected website, Kasada serves a VM script that executes in the user's browser. This script performs two critical functions:

1. It collects various signals from the browser. These signals are used to assess the likelihood of the visitor being human versus an automated system.

2. The script then posts a payload containing these signals to a specific endpoint. Upon successful verification, this endpoint returns a token.

2.3.2 Request Interception and Token Usage

Kasada employs an additional script that intercepts outgoing requests from the VM. This interception script:

1. Extracts the token received from the previous step.

2. Includes this token in future requests, either as a header or a cookie.

This token allows Kasada's backend to assess each request and determine whether to allow or block the associated action based on the token's validity. Through this two-stage process, Kasada creates a system for distinguishing between legitimate users and potential bots or automated systems, providing protection for their clients' web applications.

2.4 Decompilation Concepts

2.4.1 Definition and Goals of Decompilation

Decompilation is the process of transforming low-level language programs, such as bytecode, into high-level language representations. Cifuentes [1994, §1.1] defines the primary goal of decompilation as recovering a program's source code or a similar high-level representation from its executable form. This process aims to enhance program understanding, facilitate software maintenance, enable code reuse, and assist in malware analysis.

2.4.2 General Steps of Decompilation

Cifuentes outlines a general framework for decompilation, which consists of several key steps:

Parsing and Initial Analysis

The decompilation process begins with parsing the low-level code and performing an initial analysis. This step involves disassembling the machine code or bytecode and creating an Intermediate Representation (IR) of the program. The IR serves as an intermediate step between the low-level and high-level representations of the code [Cifuentes, 1994, Chp. 1].

Control Flow Analysis

CFA is required for understanding the program's structure. It involves identifying basic blocks, determining the relationships between these blocks, and constructing a CFG. Cifuentes emphasizes the importance of CFA in recovering high-level control structures such as loops and conditional statements.

Data Flow Analysis

Data Flow Analysis (DFA) focuses on tracking the flow of data through the program. This step helps in understanding how variables are used and modified throughout the code. Cifuentes [1994] describes various DFA techniques, including reaching definitions analysis and live variable analysis, which are essential for accurate decompilation.

Code Generation

The final step involves generating high-level code from the analyzed and transformed intermediate representation. Cifuentes discusses various strategies for producing readable and semantically equivalent high-level code, including the reconstruction of complex data types and control structures.

2.4.3 Challenges Specific to JavaScript Decompilation

While Cifuentes' work primarily focuses on code that is designed to run on bare metal, JavaScript decompilation presents unique challenges due to its interpreted nature and dynamic features:

1. **Dynamic Typing**: JavaScript's dynamic typing system makes it challenging to infer precise type information during decompilation, a problem not typically encountered in the compiled languages Cifuentes studied.

2. **Closures and Scope**: JavaScript's lexical scoping and closure mechanisms add complexity to data flow analysis, requiring adaptations to the techniques described by Cifuentes.

These challenges require adaptations and extensions to the decompilation techniques outlined by Cifuentes to effectively handle the complexities of JavaScript and modern obfuscation methods.

2.5 Graph Theory in Decompilation

Graph theory plays an important role in decompilation techniques, particularly in the analysis and representation of program structure. In our decompiler, we leverage graph theory concepts to build and analyze Control Flow Graphs (CFGs), which are required for understanding the structure and flow of the decompiled program.

2.5.1 Control Flow Graphs

A CFG is a directed graph representation of a program, where nodes represent basic blocks of code, and edges represent the possible flow of control between these blocks. In our implementation, we use the `petgraph` library in Rust to construct and manipulate these graphs petgraph [2024].

Each node in our CFG corresponds to a `BasicBlock`, which contains a sequence of instructions. The edges between these nodes are labeled to indicate the type of control flow, such as `consequent` and `alternate` for conditional jumps, or `try`, `catch`, and `finally` for exception handling structures.

2.5.2 Dominator Trees

An important concept in our graph analysis is the dominator tree. A node d dominates a node n if every path from the entry node to n goes through d. The dominator tree provides a hierarchical view of the program's control flow and is required for identifying conditional structures.

2.5.3 Cycle Detection

Identifying cycles in the CFG is used for identifying loop structures. We employ Johnson's algorithm for finding all bounded-length simple cycles in a directed graph [Johnson, 1975]. This algorithm is useful for detecting various types of loops, including while loops, do-while loops, and infinite loops.

Chapter 3

Parsing the Interpreter

Before we develop a decompiler for Kasada's VM, we must first understand and parse the dynamic components of the interpreter. This task is complex due to the VM's design, which includes dynamic elements that change between different versions of the interpreter.

3.1 Structure of Kasada's Virtual Machine

The interpreter has several important components, these components include the bytecode itself, which is an array of 32-bit signed integers containing the low-level instructions representing the program to be executed, the bytecode parsing function, responsible for interpreting different types of values from the bytecode, the control signal array, which determines how different types of values are read from the bytecode, and the opcode array, which defines the set of operations the VM can perform. The VM's design includes dynamic elements, particularly in the bytecode parsing function, which changes between different instances of the VM.

3.2 Challenges in Parsing the Interpreter

Unlike static interpreters, where parsing is often straightforward, Kasada's VM requires extra analysis due to its changing structure. The structure and implementation of the bytecode parsing function varies between different instances of the VM. Similarly, the values and order of the control signals in the control signal array change, affecting how different types of values are interpreted from the bytecode. Additionally, the order of opcodes in the opcode array differs between VM instances.

These dynamic elements require a flexible parsing approach that adapts to the variations in the VM's structure. Our goal is to extract a mapping between the control signals and their corresponding value types, which is

required for correctly interpreting the bytecode during the decompilation process.

3.3 Decoding the Bytecode String

The bytecode instructions are stored in an encoded string format using a custom encoding scheme. To decode these instructions, a function takes three parameters: the bytecode string, a custom alphabet string that defines the character set used for encoding, and an offset value that helps determine number boundaries. An example bytecode string starts like this:

`4dodofohojolonoporotovoxozooDoFoHoJoLoNoPS7e...`

The function processes the bytecode string character by character, using the custom alphabet to convert characters into numerical values. The offset parameter acts as a delimiter to identify where one number ends and another begins. After the initial decoding process extracts the raw integers, the function handles embedded string data found within these instructions. The final output is a sequence of 32-bit integers representing the VM's instructions, along with any decoded string constants that were embedded in the bytecode. A simplified version of this decoding function is shown below:

```
1  function decode(bytecode, alphabet, delimiter) {
2      var alphabetLength = alphabet.length;
3      var multiplierBase = alphabetLength - delimiter;
4      var result = [];
5
6      for (var i = 0; i < bytecode.length;) {
7          var value = 0;
8          var multiplier = 1;
9
10         while (true) {
11             var charIndex = alphabet.indexOf(bytecode[i++]);
12             value += multiplier * (charIndex % delimiter);
13
14             if (charIndex < delimiter) {
15                 result.push(value | 0);
16                 break;
17             }
18
19             value += delimiter * multiplier;
20             multiplier *= multiplierBase;
21         }
22     }
23
```

```
24        return result;
25    }
```

3.4 Bytecode Parsing Function and Control Signals

The bytecode parsing function is responsible for extracting various data types from the bytecode. This function works in conjunction with a control signal array to determine how to interpret different parts of the bytecode.

3.4.1 Control Signal Array

The control signal array is an array of values that act as markers in the bytecode. Each control signal corresponds to a specific data type or value interpretation method. When the bytecode parsing function encounters an integer that matches a control signal, it knows how to interpret the subsequent integers. For example, a control signal array might look like this:

```
var N = [6, 18, 14, 2, 20, 40];
```

Each number in this array corresponds to a specific type of value or interpretation method.

3.4.2 Bytecode Parsing Function

Here's an example of a bytecode parsing function:

Integers

```
var R = function (v, u, f, a) {
  var r = v[u[0]++];
  if (r & 1) {
    return r >> 1;
  }
```

Integer handling occurs in this section. First, the function reads an integer value from the bytecode array `v` using `v[u[0]++]`. If the least significant bit of this integer is 1 (checked with `r & 1`), it's interpreted as an integer. The value is obtained by right-shifting the integer by 1 (`r >> 1`). The right-shift operation (`r >> 1`) divides the odd number by 2 and discards the remainder, resulting in the final integer value.

64-bit Floating Point Values

```
if (r === f[0]) {
  var t = v[u[0]++];
  var M = v[u[0]++];
  var h = t & 2147483648 ? -1 : 1;
  var l = (t & 2146435072) >> 20;
  var x = (t & 1048575) * Math.pow(2, 32) +
      (M < 0 ?
          M + Math.pow(2, 32)
          : M
      );
  if (l === 2047) {
    if (x) {
      return NaN;
    } else {
      return h * Infinity;
    }
  } else {
    if (l === 0) {
      l++;
    } else {
      x += Math.pow(2, 52);
    }
    return h * x * Math.pow(2, l - 1075);
  }
}
```

If the integer matches the first control signal (`r === f[0]`), it's interpreted as a IEEE 754 floating-point number. The function reads two more integers and performs bitwise operations to reconstruct the number. This process involves extracting the sign, exponent, and mantissa from the integers and combining them to form the final floating-point number.

The bitwise operations used are a way to extract and combine the components of an IEEE 754 floating-point number. The sign is determined by the most significant bit of the first integer (`t & 2147483648`), the exponent is extracted from the next 11 bits (`(t & 2146435072) >> 20`), and the remaining bits form the mantissa.

Special cases like NaN and Infinity are handled explicitly:

- NaN: If the exponent (`l`) is 2047 and the mantissa (`x`) is non-zero, the function returns NaN.
- Infinity: If the exponent is 2047 and the mantissa is zero, the function returns Infinity (positive or negative, depending on the sign bit `h`).

This function reconstructs any IEEE 754 double-precision floating-point number, and special values, from its binary representation in the bytecode.

Strings

```
if (r === f[2]) {
  if (a != null && a.a) {
    return a.a(v[u[0]++], v[u[0]++]);
  }
  var O = "";
  var L = v[u[0]++];
  for (var n = 0; n < L; n++) {
    var k = v[u[0]++];
    O += String.fromCharCode(k & 4294967232 | k * 59 & 63);
  }
  return O;
}
```

String parsing occurs when the integer matches the third control signal (`r === f[2]`). The function first reads the string length, then proceeds to read that many integers from the bytecode and decode them into characters. The decoding process involves a custom formula that uses bitwise operations to transform each integer into a character code (`String.fromCharCode(k & 4294967232 | k * 59 & 63)`).

The expression `v[u[0]++]` is used throughout the function to read integers from the bytecode. It reads a 32-bit signed integer from the array `v` at the index stored in `u[0]` (the instruction pointer), then increments the instruction pointer. This mechanism allows the function to sequentially read integers from the bytecode as it processes different values.

Other Types

```
  if (r !== f[5]) {
    if (r === f[4]) {
      return null;
    } else if (r === f[1]) {
      return true;
    } else if (r === f[3]) {
      return false;
    } else {
      return u[r >> 5];
    }
  }
};
```

3.4.3 Mapping Control Signals to Types

To effectively parse the bytecode in later phases of the decompilation, we need to extract a mapping between the control signals and their corresponding value types. This mapping might look like:

```
control_signal_mapping = {
    0: number,
    1: boolean_true,
    2: string,
    3: boolean_false,
    4: null,
    5: undefined
}
```

If none of the control signal mappings match and if it doesn't return a small integer we use the integer as index right-shifted by 5 for a register and return the value from this register (`r >> 5`). This mapping allows our decompiler to correctly interpret the bytecode based on the control signals encountered.

3.5 Parsing Approach Using Speedy Web Compiler

To parse these dynamic components, we use SWC [2024], a fast JavaScript/-TypeScript compiler written in Rust. Speedy Web Compiler (SWC) allows us to parse the obfuscated JavaScript into an Abstract Syntax Tree (AST), which we can then analyze programmatically.

Our parsing process involves using SWC to parse the obfuscated JavaScript into an AST, traversing the AST to locate the bytecode parsing function and control signal array, analyzing the structure of the bytecode parsing function to determine how different types are extracted, and creating a mapping between control signals and value types based on this analysis.

This approach allows us to account for the dynamic nature of the VM, extracting the necessary information regardless of variations in the specific implementation.

3.6 Opcode Parsing

The VM's instruction set is defined by an array of opcode functions. These functions determine the operations that the VM can perform. The order of opcodes in this array is randomized for each VM instance, what might be opcode 5 for addition in one instance could be opcode 12 in another instance.

This randomization serves as an additional protection mechanism, making analysis of the VM more difficult.

The opcode array contains functions for various operations such as arithmetic operations, comparisons, property access, and control flow. Executing an opcode involves indexing this array with the opcode number and calling the resulting function. To decompile the VM's bytecode, we must know the correct index of each operation in the opcode array, as these indices are used in the bytecode to specify which operation to execute next. Since the mapping between opcode numbers and their operations changes between VM instances, our decompiler must analyze each VM instance individually to determine the current opcode assignments.

Our parsing process for opcodes involves identifying this array in the AST and analyzing its structure to determine the available operations and their current mapping. By understanding the current opcode assignments, we can correctly interpret the bytecode's operation references. This dynamic analysis is essential as we cannot rely on fixed opcode numbers across different instances of the VM.

Chapter 4

Disassembly: From Bytecode to Basic Blocks

After extracting and parsing the interpreter components, the next step is transforming the bytecode into a human-readable IR. This process, known as disassembly, consists of converting sequences of opcodes and operands into blocks of human-readable instructions. The disassembler must correctly interpret various value types, manage register allocations, and handle complex control flow instructions such as conditional jumps and exception handling.

4.1 Intermediate Representation Design

Our IR sits between low-level bytecode and the final JavaScript output. The IR captures both data and control flow aspects of the program. We designed the IR around two primary components: values and instructions.

Values in our IR are the full range of data types that are used by the VM. Kasada's VM is register based, meaning it uses numbered registers for storing and manipulating values during execution, rather than a stack based approach. These values include primitive types such as numbers, strings, and booleans, as well as special values like `null` and `undefined`. TheIR also contains references to these registers and memory cells, and supports complex structures such as objects, arrays, and functions. Additionally, it expresses computations through various types of expressions, including member access and binary operations.

Instructions in the IR represent the various operations supported by the VM. Register assignments form the backbone of the instruction set, handling the movement of values between registers. Property assignments manage object modifications, and control flow instructions handle program flow through jumps, conditional branches, and function calls. Exception handling instructions manage the interactions between `try`, `catch`, and `finally` blocks. Memory operations provide access to the VM's memory cells.

4.2 Traversal Strategy

Our disassembler begins at the VM's entrypoint, which corresponds to the first instruction in the bytecode. From this point we start a Breadth First Search (BFS) traversal to discover and process all reachable code paths.

While a linear traversal of the bytecode would be simpler, it would struggle to correctly handle exception handlers. Exception handlers can transfer control to arbitrary locations in the code, and these handlers may themselves contain additional exception handlers. A recursive traversal naturally handles these nested structures and ensures that all possible execution paths, including those through exception handlers, are properly discovered and processed.

The traversal maintains a queue of basic blocks to process, initially containing only the entrypoint block. As the disassembler encounters control flow instructions like jumps or function definitions, it adds the instruction pointer to the queue. This ensures that we discover all code paths that could be executed. The recursive approach is important for handling nested structures like functions within functions or complex exception handling patterns, where the natural flow of code may branch into multiple paths that need to be explored independently.

4.3 Basic Block and Control Flow

A basic block is a sequence of instructions with specific structural properties. Each basic block represents a code sequence with a single entry point and one or more exit points. The most important characteristic of the basic block is that once the first instruction is executed, all subsequent instructions must be executed sequentially without any possibility of branching except at the end of the block.

The virtual machine uses a set of instructions for directing program flow: direct jumps for unconditional transfers, and `JumpIfTrue` and `JumpIfFalse` for conditional branching. When our disassembler encounters these instructions, it constructs appropriate structures in the intermediate representation that capture the program's branching behavior.

For direct jumps, we add the target instruction pointer to our traversal queue and add a `Jump` instruction to the current basic block's instructions. Conditional branching requires more complex handling. When we encounter a conditional jump instruction, our disassembler creates two new basic blocks representing the two possible execution paths. One block represents the code that executes when the condition is true, while the other contains the code for when the condition is false. Both blocks are added to our traversal queue, ensuring we explore all possible paths through the program.

Each instruction in the basic block is stored along with a unique identifier, its instruction pointer, which is the original location of the instruction in the bytecode. During traversal, we maintain a mapping between instruction pointers and their basic blocks. This mapping enables an important optimization strategy when handling code paths that converge. The bytecode often contains identical sequences of instructions that can be reached through different paths, an example of this would be a basic block that is reached by both blocks of a conditional jump.

Rather than creating duplicate blocks for these identical sequences, our disassembler recognizes when it encounters instructions it has seen before. When we discover such a duplicate sequence, we analyze the blocks involved and may split them to optimize the program structure. The shared sequence becomes its own block, and the original blocks are modified to jump to this common code. This optimization reduces the overall size of our intermediate representation and makes it easier to find higher level control-flow structures later on.

4.3.1 Example Intermediate Representation

Here is an example of how our IR represents a function broken down into basic blocks:

```
FUNCTION BLOCK 53:    // Function entry point
    53  SET MEMORY      v22  VALUE: arg0  // Store argument
    56  SET REGISTER    r4   VALUE: v5
    59  SET REGISTER    r7   VALUE: v22
    62  SET REGISTER    r5   VALUE: r4(r7)
    66  IF   TEST: r5   CONSEQUENT: 82    ALTERNATE: 69

BLOCK 69:   // Alternate path
    69  SET REGISTER    r7   VALUE: v4
    72  SET REGISTER    r8   VALUE: v22
    75  SET REGISTER    r6   VALUE: r7(r8)
    79  SET REGISTER    r5   VALUE: r6
    82  JUMP 82

BLOCK 82:   // Conditional branch
    82  IF   TEST: r5   CONSEQUENT: 98    ALTERNATE: 85

// Additional blocks omitted for brevity...
```

Each block starts with a header showing its type (Function or Basic block) and entry point. Instructions within blocks are labeled with their

bytecode positions. The example shows memory operations, register assignments, conditional branches, and control flow transfers. Variable names like `r4` represent registers, while `v22` represents a memory location.

4.4 Expressions

The disassembler handles various types of expressions in the bytecode, from simple arithmetic operations to complex function definitions.

Binary expressions, which include operations like addition, subtraction, and comparison, can appear in two forms in the bytecode. The first form involves immediate values, where the operands are encoded directly in the bytecode. These operations can be resolved during the disassembly process itself. The second form involves register based operations, where one or both operands come from registers. These operations must be preserved in our intermediate representation since their values will not be known until runtime.

Function construction requires especially careful handling. When the disassembler encounters the `InitFunc` opcode, it performs several important steps. First, it creates a new block specifically marked as a function entry point. This block is special because it needs to handle parameter passing, the VM allocates specific registers to store function arguments in a sequential manner, starting from register 4. If a function has two parameters, the function caller will load these arguments into register 4 and 5 respectively.

4.5 Exception Handling

Exception handling is one of the more sophisticated parts of the VM's design. Rather than using straightforward and JavaScript native try-catch-finally blocks in an opcode, Kasada's VM implements exception handling through instruction pointers. The `SetCatch` operation stores the instruction pointer where execution should continue if an exception occurs. Similarly, `SetFinally` stores the instruction pointer to a finally block that must be executed in all cases.

Our disassembler must transform these instruction pointer based exception handlers into a structured representation that captures the relationships between try, catch, and finally blocks. When we encounter a `SetCatch` instruction, we look at the instruction that comes after it to determine the complete structure of the exception handler. If the `SetCatch` is immediately followed by a `SetFinally` instruction, we know we are dealing with a full try-catch-finally structure. Otherwise, we are handling a simpler try-catch pattern.

When processing these structures, our disassembler creates blocks for each component. The try block is created at the current instruction pointer,

while catch and finally blocks are created at their respective target instruction pointers. Each of these blocks is marked with its specific type. We will use these block types in chapter 6.

Chapter 5

Constructing Control Flow Graphs

After disassembling the bytecode into basic blocks, the next step is constructing control flow graphs CFGs that represent the program's control flow. The CFG is a directed graph where nodes represent basic blocks and edges represent control flow paths between these blocks.

5.1 Nodes

To build the control flow graph for a function, we use a queue-based traversal strategy starting from the function's entry point. We maintain a queue of instruction pointers that point to basic blocks we need to process, initially containing only the entry block's instruction pointer. For each pointer in the queue, we create a new node in the graph. We must separate the disassembly and CFG construction because a basic block may be used by multiple functions.

By definition, a basic block is a sequence of instructions that can only be entered at the beginning and exited at the end, with no possibility of branching except at the end of the block. This means that only the last instruction of a basic block can affect the control flow. Therefore, as we process each block, we only look at its final instruction to determine which blocks should be processed next:

- For direct jumps, we add the target instruction pointer to our queue.
- For conditional jumps, we add both the consequent and alternate instruction pointers.
- For exception handlers, we queue the pointers to the try, catch, and finally blocks.

This queue-based approach ensures we discover all reachable blocks in the function, even when they are not arranged sequentially in the bytecode. The process continues until our queue is empty, meaning we have processed all reachable blocks in the function.

5.2 Edges

After discovering all nodes, we perform a second pass over the blocks to add the edges between them. This two-phase approach is necessary because during the first pass, when we encounter a jump instruction, its target block might not have been discovered and added to the graph yet. By first creating all nodes and then adding the edges in a second pass, we ensure all necessary nodes exist in the graph before we attempt to connect them. The edges are labeled to indicate the type of control flow they represent:

- Direct jumps create a single edge labeled *jump* to their target block.
- Conditional jumps create two edges labeled *consequent* and *alternate* for the true and false paths respectively.
- Return instructions have no outgoing edges as they terminate control flow.
- Try-catch blocks create edges from the try block to both its normal successor and the catch handler, labeled *try* and *catch* respectively.
- Try-finally blocks create edges ensuring control flows through the finally block, labeled *try* and *finally.*
- Try-catch-finally blocks create edges where both normal execution and exception handling paths flow through the finally block, using all three labels.

Each of these edge patterns helps us distinguish between different types of control flow during control flow analysis in chapter 6.

5.3 Example Control Flow Graph

To illustrate how these concepts come together, Figure 5.1 shows the control flow graph generated from the function we examined earlier. Each node represents a basic block, with edges showing possible control flow paths. The graph demonstrates conditional branching (shown by split paths) and path convergence (where multiple edges lead to the same block). The numbers in the nodes correspond to the bytecode positions where each block begins. Figure 5.2 shows a more complex function graph, involving exception handling and a loop.

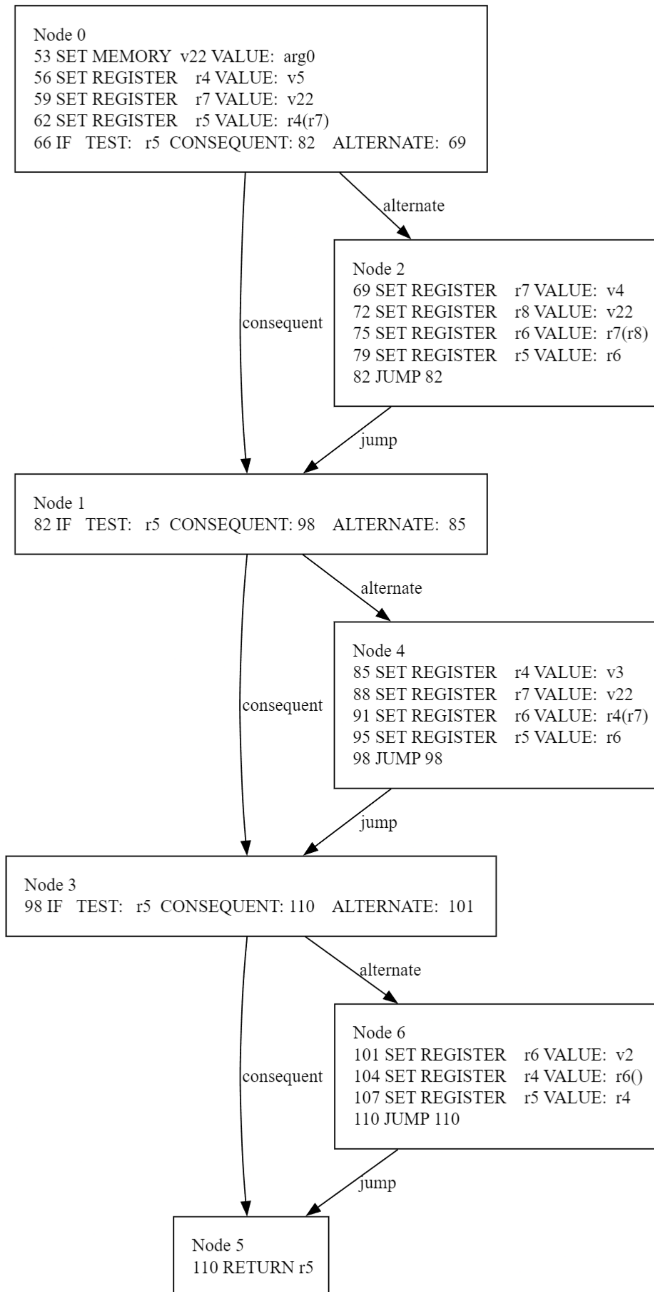


Figure 5.1: Control flow graph of a function

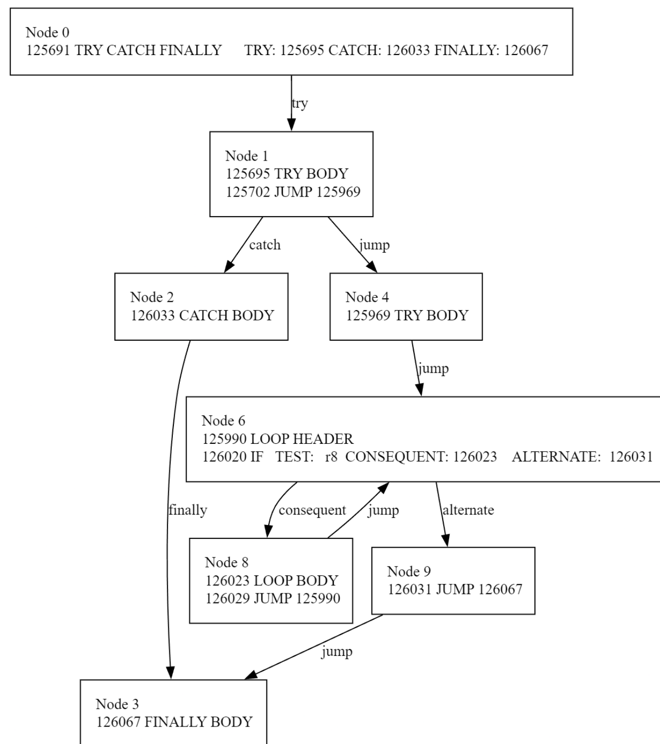


Figure 5.2: Control flow graph of a try-catch-finally and loop in try body

Chapter 6

Control Flow Analysis

Control flow analysis is an important phase in the decompilation process that identifies high-level language constructs from low-level control flow graphs. In particular, control flow analysis is required to identify loops, conditional statements, and try-catch blocks within the program's control flow graph. This analysis provides the necessary information for the code generation phase covered in chapter 7 to produce structured, readable JavaScript code that reflects the original program's control flow.

6.1 Structuring Loops

The first phase of control flow analysis involves identifying and structuring loops. To find loops in the control flow graph, we first need to identify all cycles. For this, we use Johnson's algorithm, which efficiently finds all elementary circuits in a directed graph [Johnson, 1975].

6.1.1 Loop Structures and Detection

We distinguish between three types of loop structures, each with distinct control flow patterns that we can identify in the control flow graph. Using Johnson's algorithm, we first find all cycles in the graph, then analyze each cycle's structure to classify it:

- **While Loop**: A structure containing a conditional test at its header node, followed by a loop body that ends with a latching node. The header node's conditional test determines whether to enter the loop body or exit the loop. We identify this pattern when the header node contains a conditional jump and the latching node contains an unconditional jump back to the header.
- **Do-While Loop**: Similar to a while loop, but with the test condition at the end rather than the beginning. This ensures the loop body

executes at least once before the test. We identify this pattern when we find a latching node containing a conditional jump back to the header node.

- **Infinite Loop**: The simplest loop structure, containing just a header and an unconditional jump back to it. Since these loops lack a built-in exit condition, we must identify break statements by finding nodes within the loop that have edges leading to nodes outside the loop structure. We only search for break statements in infinite loops, as while and do-while loops in the VM never contain break statements, they always use their conditional test for loop termination. We classify a loop as infinite when its latching node contains an unconditional jump and there are no conditional exits at the header or latching nodes.

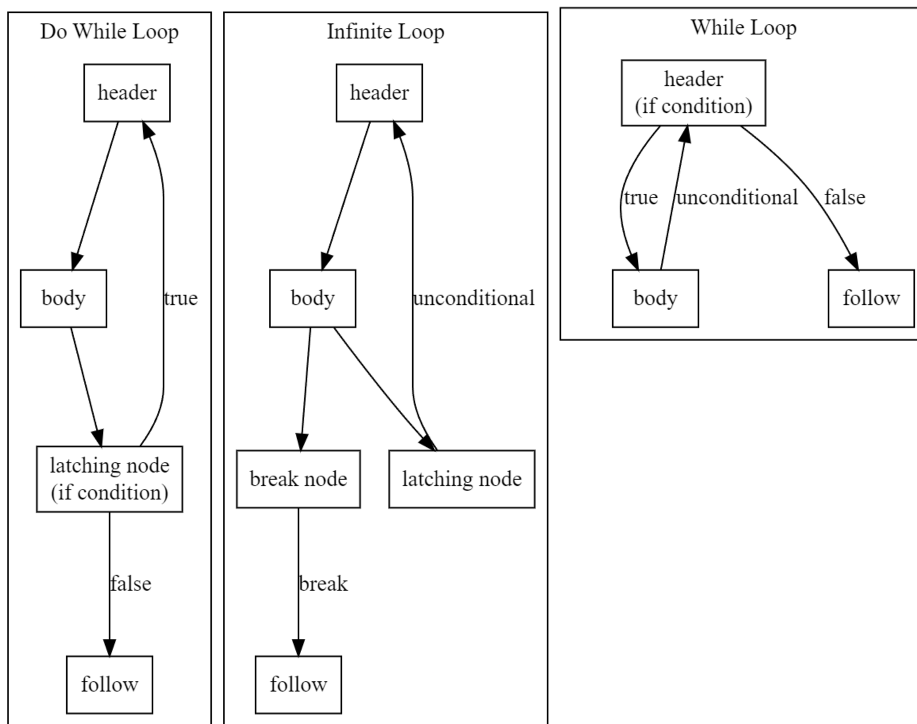


Figure 6.1: Control flow graph patterns for loops.

6.2 Structuring Exception Handlers

After identifying loops, we structure exception handlers in the control flow graph. Exception handlers create unique control flow patterns where execu-

tion can transfer to error handling code from multiple points. In JavaScript, these are known as try-catch-finally blocks.

6.2.1 Types of Exception Handlers

We identify three types of exception handling structures:

- **Try-Catch**: The simplest form, containing a `try` block and a `catch` block. If an exception occurs in the `try` block, control transfers to the `catch` block.
- **Try-Finally**: Contains a `try` block and a `finally` block that executes regardless of whether an exception occurred. The `finally` block executes both after normal completion and after exception handling.
- **Try-Catch-Finally**: Combines both patterns, with a protected block, an error handler, and a cleanup block that executes in all cases.

6.2.2 Control Flow Patterns

Exception handlers create distinct control flow patterns that we can identify in the graph:

- The try block is connected to its catch handler through special exception edges, indicating where control transfers on error.
- Finally blocks have incoming edges from both the normal exit of the try block and the exit of the catch block (if present).
- The exit of the finally block represents where control flow continues after the exception handler.

6.2.3 Structuring Algorithm

The algorithm for identifying exception handlers works as follows:

1. Identify try blocks by looking for nodes with outgoing exception edges
2. For each try block:
    - If it has an edge to a catch block, mark that block as its exception handler
    - If it has an edge to a finally block, or if its catch block has an edge to a finally block, mark that as its finally handler
3. Find the follow node where control flow continues after the exception handler by:

- For try-catch: finding the common target of the try and catch blocks' normal exits
- For try-finally and try-catch-finally: using the normal exit of the finally block

The identification and structuring of exception handlers before conditionals is important because exception handling constructs often contain control flow edges that could be mistakenly identified as conditional jumps.

6.3 Structuring Conditionals

The last step in our control flow analysis involves identifying and structuring conditional statements. In JavaScript, these are known as if/else constructs, representing points in the program where execution can follow different paths based on a condition. Our approach to structuring conditionals is based on Cifuentes' algorithm for structuring [Cifuentes, 1993], which provides a method for identifying conditional structures. We first compute dominance information using the Simple Fast Dominance Algorithm [Cooper et al., 2001], which provides a method to determine the dominance relationships between nodes in the control flow graph. This dominance information is required for identifying where control flow paths converge and thus where conditional structures end.

6.3.1 Types of Conditionals

We distinguish between two types of conditional structures:

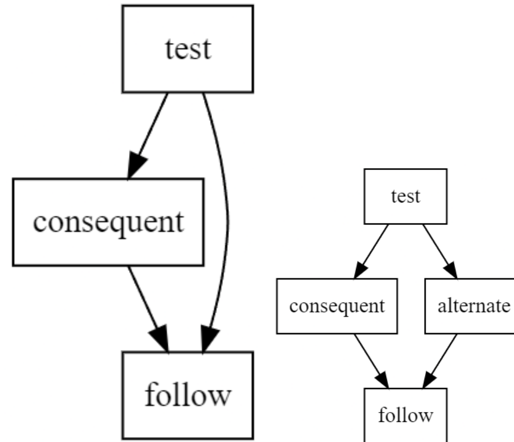
- **One-way conditional (if)**: A structure containing a test condition and a single conditional branch. When the condition is true, execution follows the conditional branch (the `then` clause); when false, execution continues to the follow node.
- **Two-way conditional (if-else)**: A structure containing a test condition and two branches, the consequent branch taken when the condition is true, and the alternate branch taken when the condition is false.

6.3.2 Components of a Conditional Structure

A conditional structure consists of several key components:

- **Test Node**: The basic block containing the conditional jump instruction that evaluates the test condition.
- **Consequent Node**: The target block executed when the condition is true.

- **Alternate Node**: (In two-way conditionals) The target block executed when the condition is false.
- **Follow Node**: The block where control flow converges after the conditional structure. The follow node has the important property of being immediately dominated by the test node, meaning all paths to the follow node must pass through the test node.



(a) One-way conditional (b) Two-way conditional

Figure 6.2: Control flow graph patterns for conditionals.

6.3.3 Structuring Algorithm

Our algorithm for identifying conditional structures follows the approach described by Cifuentes [1993], traversing nodes in reverse post-order to handle nested conditionals effectively. The key insight is that the follow node of a conditional must be immediately dominated by the test node and must be a junction point for both execution paths starting from the test node.

The algorithm processes nodes as follows:

1. Traverse nodes in reverse post-order (to handle nested conditionals from innermost to outermost).
2. For each conditional node, identify potential follow nodes by finding nodes that:
   - Are immediately dominated by the conditional node.
   - Are reachable by at least two paths from the conditional node.
3. Select the closest such node (by reverse post-order numbering) as the follow node.

Chapter 7

Code Generation

The final phase of our decompilation process involves transforming the analyzed and structured control flow graphs into readable JavaScript code. Using the structural information obtained during control flow analysis, the code generator produces high-level JavaScript. This chapter presents the algorithms and techniques used to generate human-readable code from our intermediate representation.

7.1 Graph Traversal Strategy

The core of our code generation algorithm is a modified Depth First Search (DFS) traversal of the control flow graph. Unlike traditional DFS, our implementation must handle several special cases where the standard traversal pattern would not suffice. Structured nodes, such as loops, conditionals, and exception handlers, require special processing. Multiple paths may converge at a single node, requiring careful handling to avoid code duplication.

Our traversal algorithm maintains a visited set to track processed nodes and follows a structure-aware approach to processing each node. Starting at the function's entry node, the algorithm first checks if the current node has an associated structure (loop, conditional, or exception handler). Based on this information, it either processes the node as a structured control flow element or as a basic block. The algorithm then recursively processes child nodes based on the structure's semantics, ensuring proper handling of convergent paths through the visited set.

Figure 7.1 shows an example of our graph traversal, generating the statements for the `try` body of a try-catch-finally structure. We first mark Node 2 and Node 3 as visited, so that the traversal cannot exit the `try` body. Then DFS is started at Node 1, visiting Node 4 and Node 6 in reverse post order traversal.

7.2 Structure-Based Code Generation

7.2.1 Loop Processing

Using the loop structures identified during control flow analysis, the code generator produces the appropriate JavaScript loop constructs. For each loop structure, the algorithm generates the necessary condition expressions and body statements based on the structured information. The generator handles three types of loops: while loops, do-while loops, and infinite loops with break statements. The condition expressions are generated from the header basic block, while the body is generated through recursive traversal of the loop's body nodes.

7.2.2 Conditional Processing

Building on the conditional structures identified in Control Flow Analysis, the code generator transforms these structures into JavaScript if-else statements. The algorithm handles both one-way and two-way conditionals, generating the appropriate test expressions and branch bodies.

7.2.3 Exception Handler Processing

The code generator processes the exception handling structures identified during control flow analysis to produce JavaScript try-catch-finally blocks. The algorithm generates the appropriate exception handling constructs based on the structure type: try-catch, try-finally, or try-catch-finally. Each component of the exception handler is generated through recursive traversal of its corresponding nodes in the control flow graph.

7.3 Basic Block Translation

Basic blocks are translated sequentially by processing each instruction in the block. The instruction types that require translation include register assignments, property assignments, memory assignments, return statements, and throw statements. The translation is straightforward since most VM instructions map directly to equivalent JavaScript operations. For example, a register assignment in the VM becomes a simple variable assignment in JavaScript, and a property assignment in the VM translates to an equivalent property access operation in JavaScript.

7.4 Scope Management

The code generator implements two levels of scope for variables. Memory variables are declared in the global scope since they must be accessible by

all functions in the program. Register variables are declared at the start of each function body.

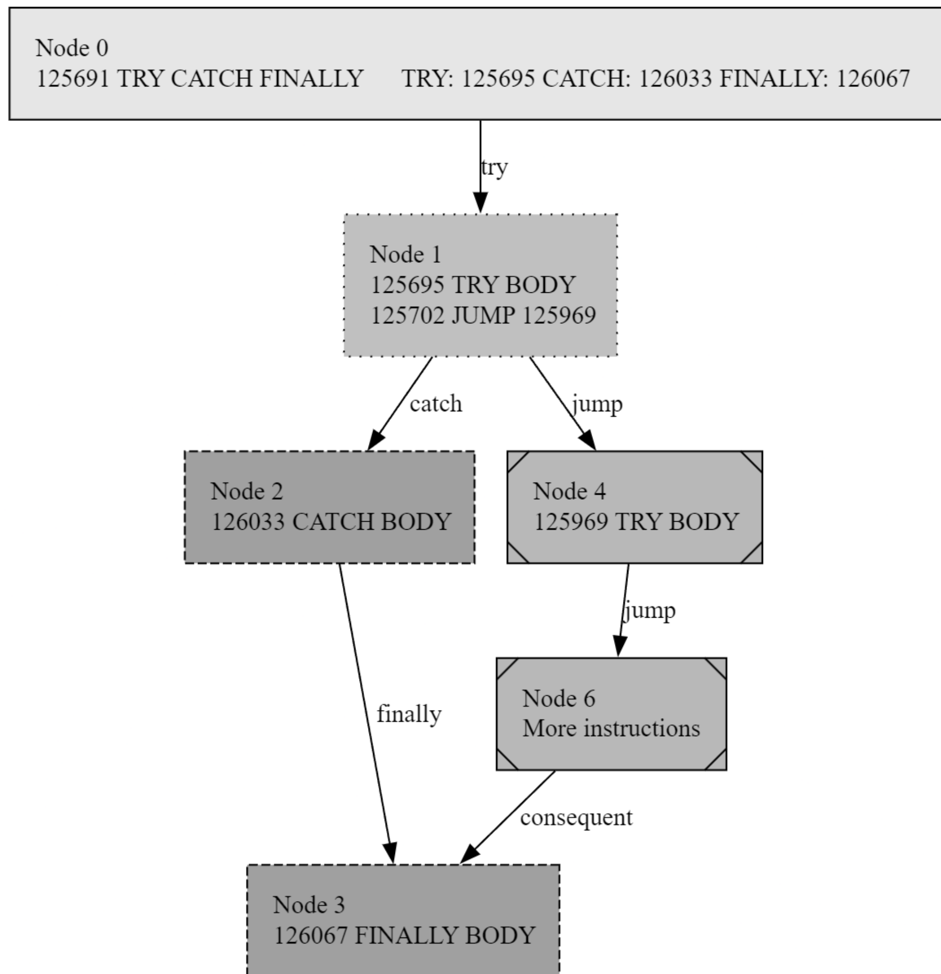


Figure 7.1: Graph traversal for generating the try body.

7.5 Code Generation Example

To demonstrate our code generation pipeline, let us examine a simple function from Kasada's VM. Figure 6.1 shows the control flow graph of this function, and Listing 1 shows the generated JavaScript code:

This example demonstrates our code generator handling register variable declarations, conditional branching, and function calls. The control flow graph was successfully transformed into structured JavaScript code.

```javascript
function func_53(arg0) {
    let r4, r5, r6, r7, r8;
    v22 = arg0;
    r4 = v5;
    r7 = v22;
    r5 = r4(r7);
    if (!r5) {
        r7 = v4;
        r8 = v22;
        r6 = r7(r8);
        r5 = r6;
    }
    if (!r5) {
        r4 = v3;
        r7 = v22;
        r6 = r4(r7);
        r5 = r6;
    }
    if (!r5) {
        r6 = v2;
        r4 = r6();
        r5 = r4;
    }
    return r5;
}
```

Listing 1: Decompiled JavaScript code example

Chapter 8

Related Work

This chapter examines existing research and techniques relevant to JavaScript VM decompilation. We begin by analyzing traditional JavaScript obfuscation and deobfuscation approaches, then explore the emergence of VM-based protection systems. We review foundational decompilation research and control flow analysis techniques, concluding with an analysis of current research gaps that our work addresses.

8.1 JavaScript Deobfuscation Tools

Traditional JavaScript obfuscation can be categorized into three main types according to Collberg's taxonomy: control obfuscation, data obfuscation, and layout obfuscation. Rauti and Leppänen [2018] conducted a comprehensive study of these techniques, revealing significant limitations in their effectiveness against automated analysis.

Their study of nine different JavaScript obfuscators demonstrated that despite claims of making code practically impossible to reverse engineer, most obfuscated code can be effortlessly reversed using automated deobfuscation tools like JSBeautifier, JSNice, and PoisonJS. Only two obfuscators (obfuscator.io and the Esoteric JavaScript Obfuscator) showed meaningful resistance against automated deobfuscation.

These limitations of traditional approaches have led to the development of more sophisticated protection mechanisms, particularly VM-based obfuscation, which fundamentally changes the protection model by introducing a custom execution environment rather than relying on code transformations.

8.2 VM-Based Obfuscation in JavaScript

VM-based obfuscation has emerged as an advanced approach to JavaScript protection, addressing the limitations of traditional obfuscation techniques.

Various commercial systems have adopted this approach, including bot protection providers such as Cloudflare and commercial JavaScript protection services like JScrambler [2024]. These systems convert JavaScript code into custom bytecode that executes within a specialized virtual machine.

This approach creates several fundamental challenges for deobfuscation tools:

- The need to understand custom bytecode formats.
- Complexity in analyzing the VM's execution model.
- Handling of dynamic instruction sets and interpreters.
- Analysis of the relationship between VM and bytecode.

While the specific implementation details of commercial VM-based systems are not publicly documented, their growing adoption indicates the effectiveness of this protection approach.

8.3 Existing Decompiler Research

The foundations of modern decompilation techniques largely stem from Cifuentes' seminal work. Their PhD thesis established fundamental principles for transforming low-level code back into high-level representations, introducing techniques for control flow analysis and structure recovery that remain relevant today. While Cifuentes' work focused primarily on compiled languages, it provides a theoretical framework that can be adapted for JavaScript VM decompilation.

Despite substantial research on decompiling traditional compiled programs, there exists a notable gap in academic research specifically addressing JavaScript virtual machine decompilation. This gap is particularly significant given the increasing adoption of VM-based obfuscation in web applications. Current knowledge about JavaScript VM decompilation exists primarily in technical blog posts and security research writeups, such as Willbold [2019]'s analysis of virtualization-based obfuscation techniques and the examination of Shape Security's JavaScript virtual machine [Dominguez, 2024].

8.4 Control Flow Analysis Techniques

Control flow analysis forms the cornerstone of modern decompilation techniques, with several fundamental algorithms underpinning current approaches. Cooper et al. [2001] introduced an efficient algorithm for computing dominance relationships in control flow graphs that has become standard in

program analysis tools. The algorithm's combination of efficiency and simplicity makes it particularly suitable for decompiler implementations.

For loop detection, Johnson's algorithm provides a robust method for finding elementary circuits in directed graphs. While developed for general graph theory applications, it has proven invaluable in decompilers for identifying and analyzing loop structures [Johnson, 1975].

The field has further evolved through techniques such as interval analysis [Allen, 1970] and structural analysis [Cifuentes, 1993]. These approaches, while primarily developed for compiled languages, provide valuable insights that can be adapted for JavaScript VM analysis. However, their application to virtualized JavaScript code presents unique challenges due to the dynamic nature of VM-based execution.

8.5 Research Gaps and Novel Contributions

Our analysis of existing research reveals several significant gaps in the field of JavaScript VM decompilation:

1. **Dynamic Analysis Challenges**: Traditional decompilation techniques assume static relationships between code and control flow. JavaScript VMs frequently employ dynamic instruction sets and control flow mechanisms that change between executions, requiring new approaches to analysis.

2. **Limited Academic Research**: While JavaScript obfuscation and deobfuscation have received significant attention [Rauti and Leppänen, 2018], the specific challenges of VM-based obfuscation in JavaScript remain understudied in academic literature.

3. **Adaptation of Classical Techniques**: Existing program analysis techniques require significant adaptation to handle the unique characteristics of JavaScript VMs, including their dynamic nature and browser-based execution environment.

Our work addresses these gaps through three main contributions:

- A methodology for analyzing dynamic VM components that adapts to variations between VM instances, handling the challenge of changing instruction sets and control flow mechanisms.

- Extensions to traditional control flow analysis techniques that specifically account for the dynamic nature of JavaScript VM bytecode, building upon established algorithms while adapting them to this new context.

- A systematic approach to reconstructing high-level JavaScript code from VM bytecode, while handling the unique challenges of browser-based execution environments.

The novelty of our approach lies not just in addressing these individual challenges, but in providing a comprehensive framework that bridges the gap between classical decompilation theory and modern JavaScript protection mechanisms. By adapting and extending traditional techniques while accounting for the specific challenges of JavaScript VMs, we provide a foundation for analyzing increasingly sophisticated browser-based protection systems. This systematic approach to JavaScript VM decompilation represents a significant step forward in understanding and analyzing modern web application protection mechanisms, while also contributing to the broader field of program analysis and decompilation.

Chapter 9

Conclusions

This thesis presented a decompiler for Kasada's JavaScript virtual machine, focusing on transforming VM bytecode into readable JavaScript code. We demonstrated that established decompilation techniques and algorithms from traditional compiled languages can be successfully adapted for JavaScript VM-based obfuscation. Through systematic application of graph theory and proven program analysis methods, we showed that such VMs can be effectively analyzed and decompiled despite their dynamic nature.

9.1 Summary of Research Contributions

Our research successfully delivered a working decompiler that addresses the challenges posed by Kasada's VM-based obfuscation. The decompiler effectively analyzes and adapts to the VM's dynamic components, including its varying instruction sets and bytecode parsing mechanisms. By applying well-established algorithms like Johnson's cycle detection and Cooper's fast dominance algorithm, we were able to accurately identify and reconstruct control flow structures. The successful application of these traditional program analysis techniques to a JavaScript VM demonstrates that core decompilation principles remain effective even when applied to modern obfuscation methods.

9.2 Evaluation of Research Objectives

The decompiler effectively handles the VM's dynamic components by building upon fundamental program analysis techniques. Following established decompilation principles, we constructed control flow graphs to represent the program's structure, used dominator analysis to identify control dependencies, and applied cycle detection to reconstruct loop structures. Through this systematic application of program analysis fundamentals, we successfully reconstruct complex control flow patterns including nested loops and

exception handlers, generating readable JavaScript code. Our work demonstrates that while JavaScript VMs introduce new challenges, particularly in their dynamic nature, the core principles of program analysis remain essential and effective when properly adapted to this domain.

9.3 Limitations and Future Work

The main limitation of this work is the absence of DFA. While our decompiler successfully reconstructs control flow structures and generates syntactically correct code through graph-based analysis, implementing DFA would provide valuable additional benefits. It would enable more sophisticated analysis of data relationships and dependencies, leading to better variable naming, more accurate identification of compound expressions, and improved optimization of the generated code.

9.4 Future Research Directions

Looking forward, this research opens several promising avenues for future work. Beyond implementing Data Flow Analysis, there are opportunities to explore how other classical program analysis techniques could be adapted for JavaScript VM decompilation. Our successful application of traditional algorithms suggests that other established methods might also be effectively adapted to this domain. There is also potential for developing optimization techniques specific to decompiled JavaScript code and extending the decompiler to handle other JavaScript VM-based obfuscation systems.

9.5 Concluding Remarks

This research demonstrates that classical decompilation techniques and algorithms, when properly adapted, remain powerful tools even for modern JavaScript VM-based obfuscation. The successful application of graph theory and established program analysis methods to Kasada's VM shows that foundational computer science principles continue to be relevant in addressing contemporary challenges in code analysis and deobfuscation. While our implementation would benefit from Data Flow Analysis, the current results validate our approach of building upon proven program analysis techniques. This work not only provides practical tools for analyzing VM-based JavaScript obfuscation but also demonstrates the value of classical program analysis methods in modern contexts.

Bibliography

Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, SIGPLAN '70, pages 1–19, New York, NY, USA, 1970. Association for Computing Machinery. doi: 10.1145/800028.808479.

Shrenik Bhansali, Ahmet Aris, Abbas Acar, Harun Oz, and A Selcuk Uluagac. A first look at code obfuscation for webassembly. In *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 140–145. ACM, 2022. doi: 10.1145/3507657.3528560.

Cristina Cifuentes. A structuring algorithm for decompilation. In *XIX Conferencia Latinoamericana de Informatica*, 1993.

Cristina Cifuentes. *Reverse compilation techniques.* PhD thesis, Queensland University of Technology, Brisbane, Australia, 1994.

Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4(1-10):1–8, 2001.

Erik Dominguez. Shape security's JavaScript VM - part 1, 2024. URL `https://www.botting.rocks/shapesecuritys-javascript-vm-part-1/`.

Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

JScrambler. Vm-based obfuscation, 2024. URL `https://docs.jscrambler.com/code-integrity/documentation/transformations/vm-based-obfuscation`.

petgraph. petgraph: Graph data structure library, 2024. URL `https://github.com/petgraph/petgraph`. Accessed: 2024-09-26.

Sampsa Rauti and Ville Leppänen. A comparison of online javascript obfuscators. In *2018 International Conference on Software Security and Assurance (ICSSA)*, pages 7–12. IEEE, 2018.

Yunhe Shi, David Gregg, Andrew Beatty, and M Anton Ertl. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(4):1–36, 2008.

SWC. Speedy web compiler (swc), 2024. URL `https://swc.rs/`. Accessed: 2024-09-26.

Jonas Willbold. The secret guide to virtualization obfuscation in JavaScript, 2019. URL `https://jwillbold.com/posts/obfuscation/2019-06-16-the-secret-guide-to-virtualization-obfuscation-in-javascript/`.