The Australian National University
2600 ACT | Canberra | Australia

School of Computing

College of Engineering and
Computer Science (CECS)

Analysis of JavaScript virtualisation based obfuscation

— 12 pt research project (S1/S2 2022)

A report submitted for the course
COMP3770, Individual Research Project

By:
Alvin J. Charles

Supervisor:
Dr. Alwen Tiu

October 2022

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the University Academic Misconduct Rules;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or Wattle site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

October, Alvin J. Charles

Abstract

Much research has been done on analysing, categorising and detecting common obfuscation techniques used to deter the analysis of JavaScript code. Little work has been done, however, on analysing virtualisation based JavaScript obfuscation, despite its known usage in averting automated bot attacks on several large-scale online shop fronts. In this paper we present a novel category of JavaScript virtualisation obfuscation techniques, context-dependent obfuscations. These obfuscations encode each virtual machine (VM) instruction ι with specific information about the state of the VM immediately before ι is executed. At runtime, each instruction is then decoded with this context information before it is executed. We implement a simple context-dependent obfuscation to demonstrate this technique. To do this, we first implement a JavaScript virtualisation obfuscation framework. Subsequently, we analyse the performance costs and security properties of this technique, and propose more advanced context-dependent encodings that use control flow as context information. We find our simple context dependent obfuscation reasonably effective in deterring static disassembly of VM instructions, with a performance degradation of only 10% to 25%.

Table of Contents

Chapter 1

Introduction

E-commerce platforms have become an increasingly popular way for customers to purchase goods and services (Mohapatra, 2013). Using the internet for commerce has many advantages including 24/7 availability, convenience and international reach. However, online shops must contend with malicious high-scale automated attacks from internet bots. It is estimated that up to 30% of internet traffic comes from malicious bots (Imperva, 2022).

Some threats from malicious bots include (and are not limited to):

- **DDoS/DoS attacks**. Websites are flooded with illegitimate requests that disrupts website service and functionality for legitimate users.
- **Credential stuffing**. Stolen credentials from data breaches are used to attempt to log in to user accounts to hijack them.
- **Scalping**. Using bots to purchase items in a way that "a normal user would be unable to undertake manually" (OWASP, 2020).

Of particular interest are 'sneaker bots'. These are bots created to scalp shoes from shoe stores such as Supreme, Shopify, Foot Locker, Nike, Adidas and others. Scalping bots like these are one of the most common, profitable and legal usages of botting (Imperva, 2019). The sneaker botting industry thrives through numerous companies (such as Ganesh bot, Cybersole and AIO bot) that sell sneaker bots to resellers, who use these bots to scalp shoes and resell them at high prices. These companies, some even with teams of developers and researchers, specialise in reverse engineering and defeating anti-bot protections used by online shoe shops to develop the bots they sell.

As JavaScript is the scripting language of the web, it is integral the functionality and security of online stores. As such, JavaScript obfuscation is a powerful tool used to hinder automated attacks through concealing the functionality of client-side code.

Obfuscation can be achieved through many means, simple techniques including minifying the source code (removing whitespace), renaming function and variable names, and encoding string and number literals in source code. Other simple techniques include injecting 'dead code' throughout the program that does nothing and disabling the use of logging functions. More complex techniques can involve parsing the program and creating a new obfuscated program with a functionally identical but more convoluted control flow. An example of this is the usage of opaque predicates, in which code is wrapped around conditional blocks that require a complex expression to be true to execute (Xu et al., 2016). This expression however, is engineered to always evaluate to true. Another control flow obfuscation is control-flow flattening, which will be discussed later in this report.

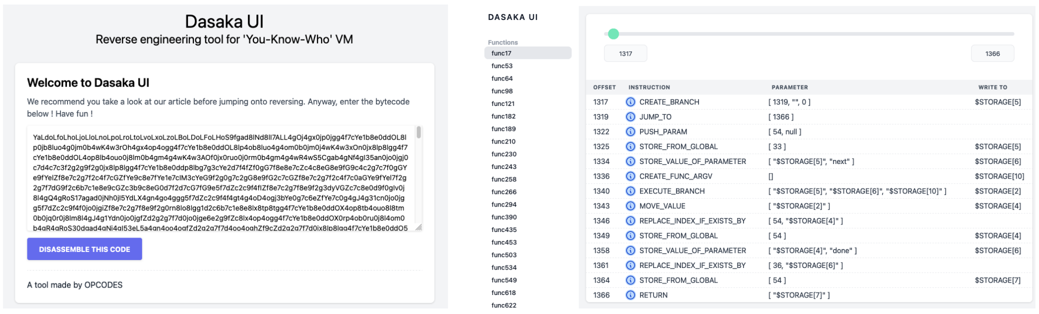
These obfuscations, while easy to apply (many tools exist to automatically apply them, see `obfuscator.io`), can be easily deobfuscated using existing tools. While they may produce code that is unintelligible to a human reviewer, programs can be written to detect and to an extent reverse the transformations these obfuscations achieve.

Virtualisation obfuscation works through compiling the source JavaScript to a set of instructions that are then executed by a JavaScript interpreter, commonly referred to as a virtual machine. Virtual machines are typically modelled after CPUs, having a stack, registers and atomic instructions. Control flow is typically implemented through jump and conditional jump instructions. This obfuscation requires a substantial amount of effort to reverse engineer as the original program is effectively 'destroyed' and is expressed only through these instructions. Due to these properties, JavaScript virtualisation obfuscation requires a substantial amount of effort to reverse engineer and as such is a preferred method of obfuscation by many large-scale online services.

Reverse engineering JavaScript virtualisation obfuscated code requires attackers to:

1. Unobfuscate the virtual machine itself
2. Understand the Instruction Set Architecture (ISA) used by the virtual machine, which may contain obfuscation-specific, non-standard instructions
3. Recover the VM payload and unobfuscate to get list of instructions
    a) Detect functions and function calls
4. Understand how the VM stores and retrieves state as the program executes
5. Build tooling to parse the VM instructions to understand what the program does. At the time of writing this paper no existing general purpose tools exist for reverse engineering JavaScript virtual machines.

Kasada is one of many security companies that offer protection against bot attacks. A part of their anti-bot protections is known to be JavaScript virtualisation obfuscation (OPCODES, 2021). Figure 1.1 shows a reverse engineering tool created by an attacker to reverse engineer a Kasada virtual machine. This tools provides attackers with a basic disassembly listing of the instructions loaded to the virtual machine. More manual work is now required to understand what the instructions do and what the program does.


Dasaka UI

Reverse engineering tool for 'You-Know-Who' VM

Welcome to Dasaka UI

We recommend you take a look at our article before jumping onto reversing. Anyway, enter the bytecode below ! Have fun !

```
YaLdoLfoLhoLjoLloLnoLpoLroLtoLvoLxoLzoLBoLDoLFoLHoS9fgad8lNd8ll7ALL4gOj4gx0jp0jgg4f7cYe1b8e0ddOL8l
p0jb8luo4g0jm0b4wK4w3rOh4gx4op4ogg4f7cYe1b8e0ddOL8lp4ob8luo4g4om0b0jm0j4wK4w3xOn0jx8lp8lgg4f7
cYe1b8e0ddOL4op8lb4ouo0j8lm0b4gm4g4wK4w3AOf0jx0ruo0j0rm0b4gm4g4wR4wS5Cgab4gNf4gl35an0jo0jgj0
c7d4c7c3f2g2g9f2g0jx8lp8lgg4f7cYe1b8e0ddp8lbg7g3cYe2d7f4fZf0gG7f8e8e7cZc4c8eG8e9fG9c4c2g7c7f0gGY
e9fYelZf8e7c2g7f2c4f7cGZfYe9c8e7fYe1e7clM3cYeG9f2g0g7c2gG8e9fG2c7cGZf8e7c2g7f2c4f7c0aGYe9fYel7f2g
2g7f7dG9f2c6b7c1e8e9cGZc3b9c8eG0d7f2d7cG7fG9e5f7dZc2c9f4flZf8e7c2g7f8e9f2g3dyVGZc7c8e0d9f0glv0j
8l4gQ4gRoS17agad0jNh0jl5YdLX4gn4go4ggg5f7dZc2c9f4f4gt4g4oD4ogj3bYe0g7c6eZfYe7c0g4gJ4g31cn0jo0jg
g5f7dZc2c9f4f0jo0jgiZf8e7c2g7f8e9f2g0rn8lo8lgg1d2c6b7c1e8e8lx8tp8tgg4f7cYe1b8e0ddOX4op8tb4ouo8l8tm
0b0jq0r0j8lm8l4gJ4g1Ydn0jo0jgfZd2g2g7f7d0jo0jge6e2g9fZc8lx4op4ogg4f7cYe1b8e0ddOX0rp4ob0ru0j8l4om0
b4gR4gRoS30dgad4gNi4gl53eL5a4gn4oo4ogfZd2g2g7f7d4oo4oghZf9cZd2g2g7f7d0jx8lp8lgg4f7cYe1b8e0ddO5
```

DISASSEMBLE THIS CODE

A tool made by OPCODES

DASAKA UI

Functions

func17
func53
func64
func98
func121
func182
func189
func210
func230
func243
func258
func266
func294
func390
func435
func453
func503
func534
func549
func618
func622

1317 1366

OFFSET	INSTRUCTION	PARAMETER	WRITE TO
1317	CREATE_BRANCH	[1319, "", 0]	$STORAGE[5]
1319	JUMP_TO	[1366]	
1322	PUSH_PARAM	[54, null]	
1325	STORE_FROM_GLOBAL	[33]	$STORAGE[5]
1334	STORE_VALUE_OF_PARAMETER	["$STORAGE[5]", "next"]	$STORAGE[6]
1336	CREATE_FUNC_ARGV	[]	$STORAGE[10]
1340	EXECUTE_BRANCH	["$STORAGE[5]", "$STORAGE[6]", "$STORAGE[10]"]	$STORAGE[2]
1343	MOVE_VALUE	["$STORAGE[2]"]	$STORAGE[4]
1346	REPLACE_INDEX_IF_EXISTS_BY	[54, "$STORAGE[4]"]	
1349	STORE_FROM_GLOBAL	[54]	$STORAGE[4]
1358	STORE_VALUE_OF_PARAMETER	["$STORAGE[4]", "done"]	$STORAGE[6]
1361	REPLACE_INDEX_IF_EXISTS_BY	[36, "$STORAGE[6]"]	
1364	STORE_FROM_GLOBAL	[54]	$STORAGE[7]
1366	RETURN	["$STORAGE[7]"]	

Figure 1.1: The user interface of *Dasaka UI*, a tool created to reverse engineer JavaScript virtual machines created by Kasada.

Our work aims to further increase the effort attackers must go through to reverse engineer JavaScript virtualisation obfuscated code through:

- **Polymorphism**. One input program compiles to a set of output payloads, such that the instructions in each payload are different but they are functionally equivalent. Web pages can then serve different payloads for different network requests, so that attackers cannot rely on a static set of instructions every time their bot interfaces with the page.
- **Context-dependent obfuscation**. The virtual machine payload is created in such a way that instructions are decoded immediately prior to their execution with some context information of the current program. This deters static disassembly of programs. Depending on what is used as context information, this technique can be used to obfuscate control flow of programs and even protect control flow integrity of running programs.

In this report we will detail some background information on JavaScript obfuscation and virtualisation obfuscation. We will then discuss our initial work with Scope Virtual machine, an existing JavaScript virtualisation obfuscation framework. We will then present the Orange Juice Compiler, a JavaScript Virtualisation Obfuscation we created to research with. Then we will present the theory of context-dependent obfuscation and a proof of concept implementation of it. We then evaluate the security properties and performance cost of our implementation. After this we will discuss related works and avenues of future work.

Chapter 2

Background

Code obfuscation refers to any technique applied to a program to conceal its functionality. Obfuscation is used for a variety of purposes, including the protection of intellectual property or trade secretes, preventing the creation of unauthorised code extensions, concealing malware functionality to deter incident response or for the creation of puzzle programs to solve for recreation.

In this section we will outline of some common JavaScript obfuscation techniques before giving an overview of virtualisation obfuscation.

2.1 Common JavaScript Obfuscation Techniques

Many of the below obfuscations can be explored at https://obfuscator.io. This website is a free online tool to apply common JavaScript obfuscations.

2.1.1 Renaming identifiers

This is a simple technique that renames variable and function names in source code. Generated names can be created to be complicated and hard to distinguish at a glance such as in Listing 2.2, taking advantage of JavaScript's lax variable name rules. This obfuscation can be defeated through renaming variables and functions to more sensible names.

Listing 2.1 Original program

```
function sayHello() {
  var hello = 'hello, world!';
  console.log(hello);
}
sayHello();
```

Listing 2.2 Program after obfuscation

```
function _0x382c7e() {
  var _0x533aa3 = 'hello, world!';
  console.log(_0x533aa3);
}
_0x382c7e();
```

2.1.2 String transformations

Two main techniques exist for concealing strings in JavaScript code, using a global string array and encoding strings. These techniques can be used together. As object properties can be accessed with strings (i.e., `console.log` is equivalent to `console['log']`), string concealment can provide high value obfuscation.

Encoding strings

String literals in JavaScript can be initialised as hexadecimal strings. This can be used as a weak obfuscation to hide the value of a string at first glance.

Listing 2.3 Original program

```
console.log('hello, world')
```

Listing 2.4 Program after obfuscation

```
console["\x6C\x6F\x67"]('\x68\x65\x6C\x6C\x6F\x2C\x20\x77\x6F\x72\x6C\
    x64')
```

Global string arrays

This obfuscation gathers the strings in the program and stores them in a global string array. Any string literals in the program are replaced with calls to the global string array. This technique can be further improved through using an intermediary function to do the string lookup such as in Listing 2.6.

Listing 2.5 Hello world program with string array encoding

```
var strings = ['log', 'hello, world'];
console[strings[0]](strings[1]);
```

Listing 2.6 Hello world program with string array encoding and an intermediary function

```
var strings = ['log', 'hello, world'];
function getString(i) { return strings[(i/3)-1];}
console[getString(3)](getString(6));
```

2.1.3 Numbers to expressions

Number literals throughout the program can be turned into expressions to obscure the original number.

Listing 2.7 Original program

```
var foo=5;
```

Listing 2.8 Program after obfuscation

```
var foo=0xa8d+-0x23de+0x2f*0x8a;
```

2.1.4 Dead code injection

Dead code injection involves adding blocks of code throughout the program that do nothing except convolute it. These increase the 'signal to noise' of the program, frustrating reverse engineering attempts by leading program analysis in the wrong path.

2.1.5 Control flow obfuscations

Control flow obfuscations involve convoluting the control flow graph of the original program to deter reverse analysis.

Opaque predicates

Opaque predicates involve wrapping code in a conditional block that always executes. The test for the conditional is engineered to be a complex expression that always evaluates to true.

Listing 2.9 Hello world program with very simple opaque predicate obfuscation

```
var a = 5;
var b = 6;
if (a !== b) {
  console.log('hello, world');
}
```

Control flow flattening

Control flow flattening involves splitting the basic blocks of the original program and putting them inside an infinite loop with a switch statement that controls which block executes. This dramatically changes the control flow of the obfuscated program as compared to the original.

2.1.6 Debug protection

Debug protections are used to deter dynamic analysis of running obfuscated code. Below are some simple debug protection techniques.

Redefining `console.log`

The `console.log` function and its related functions can be redefined as shown in Listing 2.10. This prevents any logging that an attacker might attempt on the running program.

Listing 2.10 Redefining `console.log`

```
globalThis['console']['log'] = function() {};
```

Breakpoint prevention

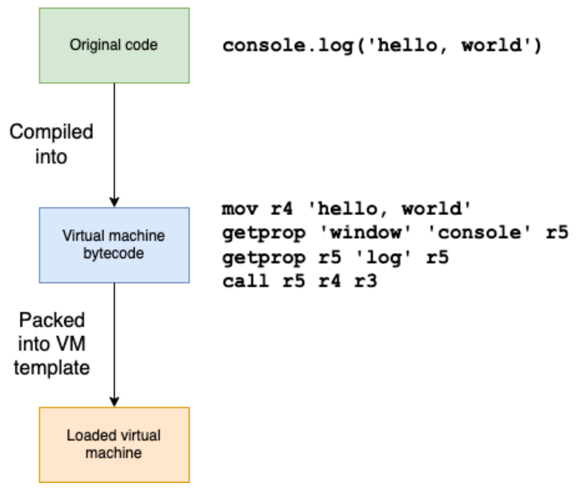
The `debugger` keyword in JavaScript can be used to stop execution and start the debugger. This can be taken advantage of by calling the debugger in an infinite loop as in Listing 2.11. The attacker will be unable to effectively breakpoint the obfuscated code to analyse it as the debugger will continually breakpoint on the infinite loop.

Listing 2.11 Breakpoint trap

```
setTimeout(function() {while (true) {debugger}})
```

2.2 Virtualisation Obfuscation

Virtualisation obfuscation works through transforming the original JavaScript to a set of instructions that are loaded into a JavaScript virtual machine, as modelled in Figure 2.1. The virtual machine is loosely modelled after a CPU and has a stack, registers and control flow through jump instructions. Unlike simpler obfuscations, virtualisation obfuscation essentially a one-way transformation, with everything except the functionality of the original code stripped away from new obfuscated program.



```
console.log('hello, world')
```

```
mov r4 'hello, world'
getprop 'window' 'console' r5
getprop r5 'log' r5
call r5 r4 r3
```

Figure 2.1: An overview of virtualisation obfuscation

2.2.1 Virtual machine organisation

It is important to understand the essentials of how programs are executed in a computer processor to understand virtualisation obfuscations, as virtual machines are modelled after CPUs. We will now provide enough description of computer organisation to contextualise virtual machine obfuscation.

Registers are a type of computer memory used to quickly retrieve and store data from. CPU's have a small, static number of registers that are used to store the data the program is currently working with. JavaScript virtual machines normally have infinite registers that can store any amount of any type of data within them. These registers are usually implemented through JavaScript arrays.

Execution proceeds as the CPU reads and executes instructions sequentially. Control flow is altered through jump and conditional jump instructions that jump execution to a defined point. All high-level control flows (conditional statements, loops, switches, etc.) are implemented through jump instructions. Control flow is also changed by function calls. Function call instructions execute a function then return code flow to the instruction after the call instruction. JavaScript virtual machines are typically similar in this regard as they also only handle control flow through only jumps and function calls.

A stack is a data type used to store a collection of elements. Items can be pushed onto and popped off the top of the stack. Stacks are typically used to store variables. Stacks are organised through 'stack frames'. A stack frame is created when a function gets called and stores the local variables created in that function, as well as the code point the function should return to after it exits. When a function exits, it tears down its stack frame. JavaScript virtual machines also have a notion of a stack and stack frames. However unlike actual programs, stack frames are usually implemented as key-value stores of variables rather than lists of bytes in memory.

Chapter 3

Initial Examination of Scope Virtual Machine

Scope virtual machine (Scope VM) is an existing, open-source JavaScript virtualisation obfuscation framework. It was last updated in 2017, but despite this was still the most complete framework of its kind that could be found at the start of this research project. The framework was in a partially broken state, unable to compile anything but simple programs.

Initial goals were to implement new obfuscations using this framework and evaluate these obfuscations using the Octane benchmarks, a suite of commonly used JavaScript benchmarks. Work commenced to update the Scope VM such that it could compile these benchmarks as in its initial state it was not able to compile any of them.

3.1 Design of Scope Virtual Machine

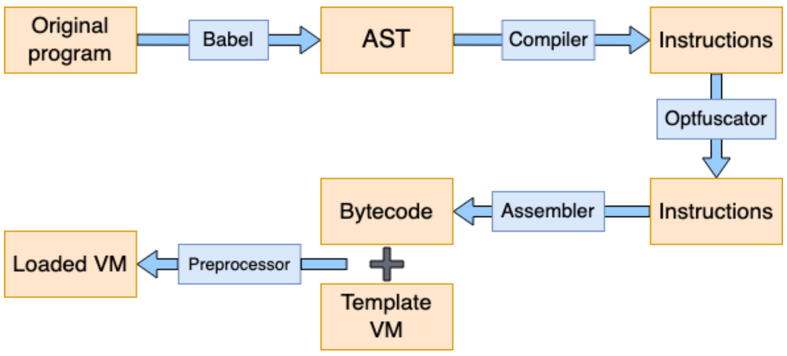


Figure 3.1: Data flow of Scope Virtual Machine

Scope VM is written entirely using JavaScript. Babel is a JavaScript library used for parsing JavaScript and is used to generate an abstract syntax tree (AST) from the input program. The compiler then uses this AST to generate VM instructions. The 'optfuscator' is used to then optimise and obfuscate these generated instructions. The assembler then assembles these instructions, producing bytecode. A template virtual machine is a virtual machine that has no instructions loaded into it. The pre-processor packs the generated bytecode into this template VM, producing a loaded virtual machine. This is the final obfuscated program. See Figure 3.1 for a diagram of this data flow.

Scope VM's compiler works through traversing the program AST, and generating instructions for every node it encounters. Global and function scopes are supported, but block scopes are not. There are two methods of storing state, a runtime variable map and registers. In the virtual machine, registers are implemented as a JavaScript array and as such each register can store anything, including functions. There is a limit of 65536 registers. The compiler gives variables an internal name, or an ID of sorts. The runtime variable map is used to map between these IDs and contents of the variable. This map is essentially used as a stack storing local variables. The compiler generates enormous amounts of dead code and in many places wraps instructions around opaque predicates.

The optfuscator performs optimisations such as pre-calculating operations between constants (e.g. `add 5 2 r3` becomes `mov 7 r3`). Register 0 holds the program counter (index of currently executing instruction) in the VM. The optfuscator conceals jumps through replacing `jump <location>` with `mov r0 <location>`. Blocks of instructions that are safe to shuffle are also randomly shuffled. As such, Scope VM generates polymorphic bytecode. Virtual machines are generated with only the instructions that are used in the program it is loaded with.

3.2 Issues with Scope Virtual Machine

We ran into several issues trying to working with Scope VM. The framework had essentially no internal documentation, very little intrinsic documentation (the source code almost seemed to be obfuscated itself) and no unit tests of any kind. The compiler generated instructions interleaved with huge amounts of dead code and opaque predicates. However there was no distinction within source code between the generation of the dead code and the legitimate code, so understanding the functionality of the compiler often required the posture of a reverse-engineer analysing obfuscated code.

The compiler output was often changed entirely by the optfuscator so understanding the functionality of compiler through compiling sample programs provided little help. The assembler was only able to take input from the optfuscator, not the compiler (assembly would fail otherwise) so removing the optfuscator from the build pipeline was not an option. The build process and the VM would often fail silently. Due to how dramatically different the program looked between compiler output, optfuscator output and assembler output, tracing errors to their possible origin within any of these processes became extremely difficult.

Creating new obfuscations would also require an deep knowledge of the entire framework, as obfuscations could be implemented in the compiler, the optfuscator, the assembler, the virtual machine and would most likely involve a number of these parts. This posed a challenge due to the difficulty of understanding any of one these parts in Scope VM.

Due to the difficulty in trying to work with Scope VM in its broken state and to allow for the creation of complex obfuscations spanning all processes of the virtualisation obfuscation pipeline, we decided to create a new virtualisation obfuscation framework to continue research on.

Chapter 4

Orange Juice Compiler

The Orange Juice Compiler (**O**bfuscating **J**avaScript **C**ompiler) is a JavaScript virtualisation obfuscation framework created as part of this research project. It was made to allow for the research of novel virtualisation obfuscation techniques. Sample compilation outputs are exhibited in Appendix A and the instruction set reference is listed in Appendix B. Should you like to view or use the source code of this framework, please email the author at u6733719@anu.edu.au to request a copy.

4.1 Design Goals

The primary design goal of the Orange Juice Compiler (OJC) was that it was complete enough to support at least a subset of the Octane JavaScript benchmarks, as this is crucial in evaluating research performed on this framework. It was also important that the framework was easy to extend and debug. To achieve this, the framework should be designed in a modular fashion, code must be well documented and unit tests must be present to prevent regression. Due to the time constraints of this project, the performance of the obfuscated code was not a priority. Additionally, this framework makes no effort to obfuscate the generated JavaScript virtual machine itself.

4.2 Unsupported Features

OJC does not support:

- Loops that aren't for, while or do-while loops
- Labelled statements
- Function closures (anonymous functions, function callbacks, functions as arguments and as return values supported)

- ES6+ features (such as classes and arrow functions, but `this` and `new` keywords and objects supported)
- Error handling (`try`, `catch` and `throw` unsupported)
- `void` and `delete` keywords
- `||=`, `??=` and logical and assignments
- Spread syntax (`...`)
- Sequence expressions

These features were intentionally unsupported to speed development time.

4.3 Design Analysis

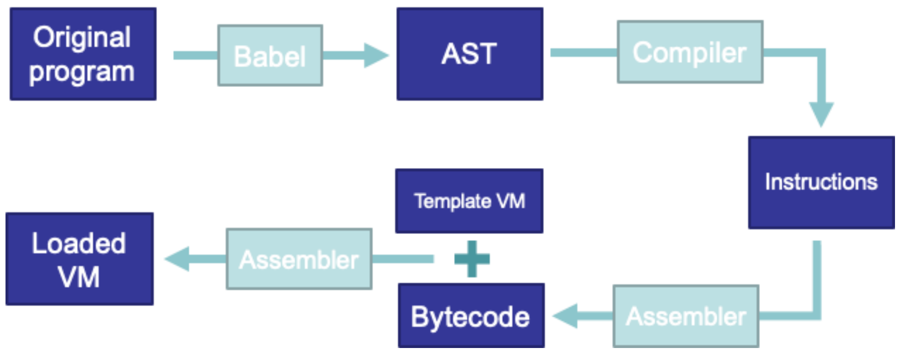


Figure 4.1: Data flow of the Orange Juice Compiler

OJC is primarily written in python. As with Scope VM, Babel is used to generate an AST. The compiler (a python script), takes this AST and output instructions in JSON format. The assembler takes these instructions, assembles them and packs them into a template VM. This loaded VM is the obfuscated program. See Figure 4.1 for a diagram of this process. We will now present an overview of each of these parts.

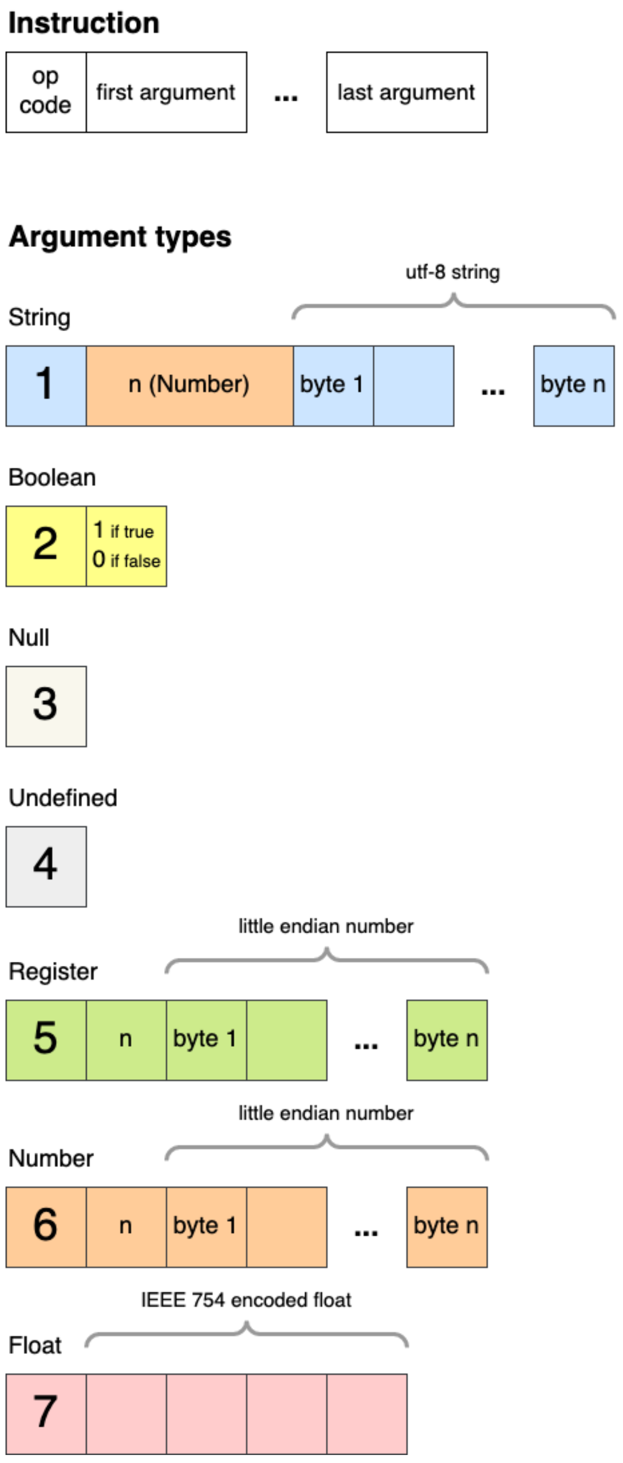


Figure 4.2: Instruction encoding of the `basic_bytecode` assembler

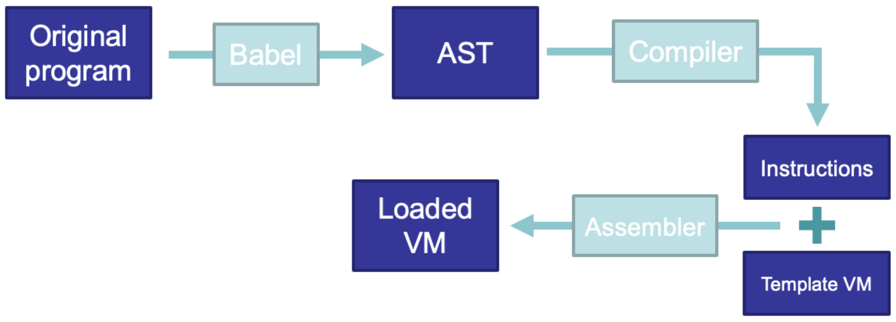


Figure 4.3: Data flow of the OJC with the `debug` assembler

4.3.1 Assembly

There are multiple assemblers in OJC and at build time one must be selected to complete the build. The purpose of the assembler is to take the instructions created from the compiler and output a loaded virtual machine. Each assembler has its own template virtual machine that it loads. As such, assemblers can load the machine however they please. This means that creating a new post-compile obfuscation just involves creating a new assembler and template VM.

These are the current assemblers in OJC and what they do:

- `debug`. The debug assembler is special as it does not assemble instructions at all. The template VM of the debug assembler is designed to be loaded with JSON instructions in the exact format the compiler outputs (see Section 4.3.3 for the specification of this format). See Figure 4.3 for how the debug assembler pipeline works. This assembler is useful for debugging and to compare against as a standard assembler.

- `basic_bytecode`. This assembler is the baseline bytecode assembly assembler. It uses a simple encoding to map instructions to bytes which can be seen in Figure 4.2. One byte is used to encode the operation, then arguments follow. Each argument begins with a single byte that expresses what type the argument is. This encoding scheme was chosen because some instructions accept arguments of multiple types. This assembler has a static mapping of operations to op codes.

- `variable_dependent`. This assembler implements randomised op codes (operation to op code mappings are randomly chosen at assembly time) and context-dependent obfuscation, which will be discussed later in this report.

4.3.2 Virtual Machine

While virtual machines vary between assemblers, they all share an architectural similarity as they must execute instructions created by the compiler.

Variable handling

OJC's virtual machine data storage is accomplished through registers and a variable map, similar to Scope VM. The variable map is modelled as a stack of key-value stores, storing mappings between variable IDs and variable contents. In the virtual machine, this is represented as an array of JavaScript objects.

Each store is initialised with one special Boolean attribute, `function_map`, and then pushed onto the stack. A variable store is initialised when:

- The program starts.
- A loop body begins
- A switch statement begins.
- A function begins. This is the only case in which `function_map` will be `true`.

What the last three statements have in common is *non-local control flow.* Non-local control flow occurs when execution abruptly jumps one or more levels out of its current executing scope. An example of this is the `return` statement, which can be called from anywhere within a function and all variables created within the whole function will be destroyed. For example, in Listing 4.1 when function `foo` returns, local variables `a` and `b` will be destroyed.

Listing 4.1 Function scope example

```
function foo() {
    let a = 5;
    if (true) {
        let b = 2;
        return;
    }
}
```

Similarly, a `break` statement can be executed anywhere within a loop or a switch, deconstructing all variables created within the loop or switch.

OJC deals with this non-local control flow through initialising a variable store every time a loop, switch or function begins. Variables will be created as needed within this store and when the loop, switch or function exits, the store is popped. Loops and switches push and pop variable stores through the `push_store` and `pop_store` instructions. How functions create and push stores will be discussed in the next section.

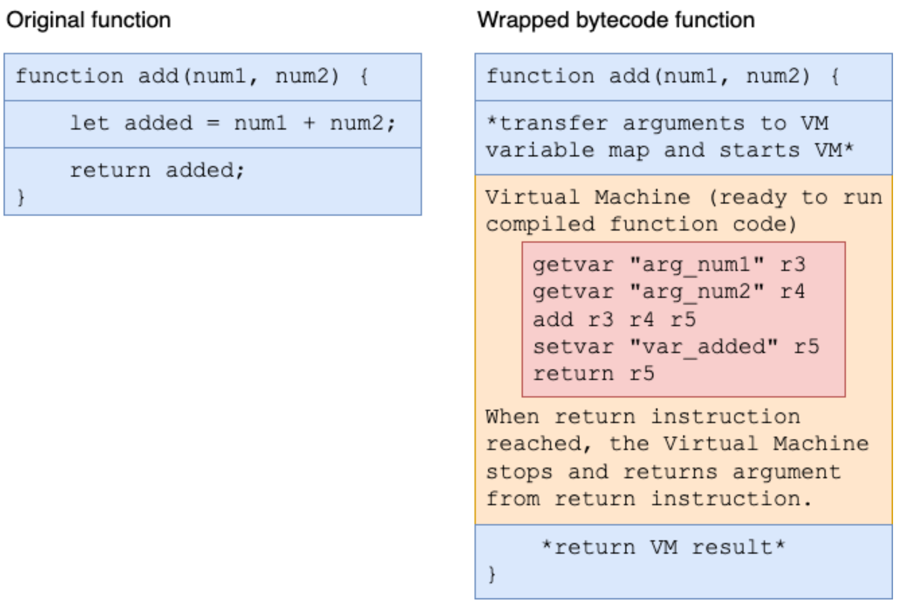
The instruction `setvar` is used to create or update a variable in the variable map. `setvar` has two arguments, the string ID of a variable and the value it is setting the variable to. `setvar` searches the variable store stack from the top to the bottom, checking if the ID it needs to update is in any map. If the ID does not exist, `setvar` creates the variable in the topmost store and sets its value. If the ID does exist, setvar updates the variable's value in the map it found it in if that map is not a function map. If the ID was found in a function map, `setvar` only updates it if it is the topmost function map, as this means that the variable is a local variable of the currently executing function. Otherwise, the variable it would be updating would be the local variable of a function earlier in the stack trace, which would not happen in normal execution.

The instruction `getvar` is used to retrieve the value of a variable from the variable map. It operates exactly like `setvar` and returns the value of the variable if it is in the map, ignoring maps of functions earlier in the stack trace. Otherwise, it returns `undefined`.

Functions

OJC supports first-class functions, meaning that functions can be passed as arguments, can be returned from functions and can be assigned as a value to a variable. To achieve this functionality, compiled functions must interface in the same way as normal JavaScript functions.

Take, for example the function `add` that takes arguments `num1`, `num2` and returns `num1 + num2`. This function must compile to a JavaScript function that also takes arguments `num1`, `num2` and returns `num1 + num2`.



```
function add(num1, num2) {
    let added = num1 + num2;
    return added;
}
```

```
function add(num1, num2) {
*transfer arguments to VM
variable map and starts VM*
Virtual Machine (ready to run
compiled function code)
    getvar "arg_num1" r3
    getvar "arg_num2" r4
    add r3 r4 r5
    setvar "var_added" r5
    return r5
When return instruction
reached, the Virtual Machine
stops and returns argument
from return instruction.
    *return VM result*
}
```

Figure 4.4: Diagram of function wrapping

This can be achieved through function wrapping as outlined in Figure 4.4. This technique is implemented through the `create_func` instruction. This instruction's arguments include the starting address of bytecode of the function and a list of argument ids.

The virtual machine's main fetch and execute instruction loop is implemented inside a function. This function can be called with a bytecode address at which it will start execution. This function exits when the final instruction is reached, or the `return` instruction is called. When a `return` instruction is called, the function will return the argument of this instruction as its own return value before it exits.

`create_func` then uses a JavaScript closure to create a function that when called:

1. Push the current value of the `this` keyword to a global stack of `this` values. When bytecode needs the value of `this`, the `whatsthis` instruction will be called that returns the value at the top of the `this` stack.
2. Backs up the current program counter and registers.
3. Creates a new variable store with the `function_map` attribute set to `true`.
4. Populates the newly created variable store with the arguments passed to the function mapped under the given list of argument ids.
5. Call the virtual machine's execution function at the given starting address of the function. Wait for the function to finish execution and store its return value.
6. Pop variable stores up until and including the first variable store where `function_map` is set to `true`. This will be the variable store created in step 3. This process effectively deconstructs the function's stack frame.
7. Restore the backed up program counter and registers.
8. Pop the `this` stack.
9. Return the stored return value.

4.3.3 Compilation

The compiler traverses the AST of the program, visiting each node. Nodes are visited sequentially and depth first and instructions are generated expressing each node as it is visited. All AST nodes can be categorised as either **statements** or **expressions**. Expressions are defined as any 'unit of code that resolves to a value', whereas statements do not return values and typically perform actions. An `if` statement would be an statement, whereas the code `5+2` is an example of an expression.

If the compiler visits a statement, it generates the instructions required to run that statement. If the compiler visits an expression, it generates the instructions required to load that expression to a register, and saves that register as an attribute to the AST node of the expression. Any node that contained the expression can then use it through the register it is loaded in.

Instruction output format

The compiler outputs instructions in JSON format. Specifically, every instruction is a JSON object inside a list. Instructions can either be **labels** or **operations**. Labels mark points in the program that can later be jumped to by jump instructions or function calls. Operations are the actual instructions of the program. To denote what type an instruction is, all instructions have a `type` attribute which will always be set to either `"label"` or `"operation"`. All instructions also have the `string_repr` attribute, which contains a string representing the whole instruction.

Label specific attribute:

- **id**. This attribute stores an ID string corresponding to the label. Jump instructions will denote this ID as their jump location.

Operation specific attributes:

- **op**. String denoting what this instruction's operation is.
- **args**. Arguments for the instruction, formatted as a list containing JSON objects with two attributes:
  - **type**. String denoting type of argument (can be either "string", "boolean", "label", "register", "number", "null" or "undefined")
  - **value**. Value of argument (not used for null or undefined type arguments).
- **approx_loc**. Line number of source code this instruction was generated from. Useful for debugging.
- **top_scope_declared**. This attribute contains the number of variables that will be present in the top-most variable map when this instruction is executed. This is used for context-dependent obfuscation which will be discussed later in this report.

Registers

In the virtual machine, there is no limit to the amount of registers and no restriction on what can be stored in them. As such, the compiler uses registers liberally to move data around.

At compile time, registers allocations are kept track of through an array of Booleans, each representing whether or not the register of its index is claimed or not. If the register number in question is larger than the length of the list, it is unclaimed.

At any point in compilation, a register that is claimed is currently in use and holding data that will be used later in the program. When using registers, the compiler will pick the first unclaimed register. If all existing registers are claimed, a new register will be created and that will be picked. When the compiler has finished using a particular register, it will unclaim it so that it can be used later.

Variables and scope

The compiler gives variables unique IDs to prevent collisions between variables that have the same name but are declared in different scopes. At runtime, they are only referenced by these ID strings. These IDs also act as an obfuscation as they are randomly generated strings.

The compiler maintains a database of scopes as it compiles the program. As the compiler enters a new scope, it creates a new entry for it in the scope database, linked to the parent scope the new scope resides in. Details of variables created in this scope will be stored in this entry. This entry also stores the type of the AST node that initiated the creation of this scope. For example, if a variable is declared in the top scope of a `while` loop, the base node of the scope database entry it will be detailed in will be `WhileStatement`.

The previous section on Variable handling in the virtual machine details how the compiler generates instructions that handle variables. In summary, the `setvar` and `getvar` instructions are used to create, update and retrieve variables and the `push_store` and `pop_store` instructions are used to manage scopes in non-local control flows.

Chapter 5

Developed Obfuscations

Two obfuscations were implemented with OJC (beyond just compilation). A simple polymorphic bytecode obfuscation ensures that one input program can compile to many different obfuscated programs. A proof of concept context-dependent obfuscation was also implemented that works to deter static disassembly of bytecode. We will now go over these obfuscations in more detail. Both obfuscations were implemented with the `variable_dependent` assembler.

5.1 Polymorphic Bytecode

Polymorphic bytecode is achieved through randomising operation op-codes at assembly time. As discussed in Section 4.3.3, the compiler instruction output format identifies operations through string names. As such, the assembler has complete control over what each operation's op code is. The virtual machine stores all operations in an array, with the index of an operation being its op code. At assembly time the assembler first randomises the operation to op code mappings and assembles instructions with this random mapping. The assembler then modifies the order of operations in the operation list in the loaded virtual machine to match the randomly selected mapping.

5.2 Context-Dependent Obfuscation

In this section we will define context-dependent obfuscation and detail our proof of concept implementation of this obfuscation. We will then propose possible implementations of advanced context-dependent obfuscations that use control-flow as context. Context-dependent obfuscation refers to making the interpretation of the bytecode dependent on some 'context' information from the virtual machine.

The assembler implements an encode function f that takes a program instruction i and some context c and produces the obfuscated instruction $f(i, c)$. The virtual machine then contains a decode function g that takes an obfuscated instruction y and context c to retrieve the original instruction $g(y, c)$. These functions must be constructed such that $g(f(i, c), c) = i$.

The obfuscation strength of this technique relies on what is being used as context information. Ideal context is some aspect of the running virtual machine state that can be known at compile time, but is very difficult to find through static analysis of virtual machine bytecode after compilation.

The g function must have a low performance cost to avoid degrading the performance of the virtual machine. The performance of the f function will influence how long it takes for the assembler to obfuscate the program.

5.2.1 Implementation

In our implementation we used the number of items in the topmost variable store as context information. Variable stores are detailed in Section 4.3.2. Using the compiler scope database (discussed in Section 4.3.3), this context information is known at compile time. Variable stores are created and pushed onto the stack when a switch, loop body or function begins execution. The compile time scope database stores the base node type of every created scope, the variables created in each scope and the parent-child relationships between all scopes. The compiler also keeps track of the current scope throughout compilation.

To calculate the number of variables in the topmost variable store, the compiler sums the variables in the current scope and its parent scopes until it reaches either the topmost scope, or a parent scope who's base node is either a switch, loop or function. This summation is incremented by one to account for the `function_map` attribute all variable stores have. This algorithm is presented in python pseudocode in Listing 5.1.

Listing 5.1 Calculating number of variables in top store at compile time

```python
def calculate_variables_in_top_store(current_scope):
    num_top_store = 0
    while True:
        num_top_store += variables_in_scope(current_scope)
        current_scope = parent_of_scope(current_scope)
        if base_node_type(current_scope) in ['switch', 'loop', '
            function'] or is_top_scope(current_scope):
            break
    return num_top_store + 1
```

This calculation is performed every time an instruction is generated and stored in the `top_scope_declared` attribute of the JSON instruction output (detailed in Section 4.3.3).

After the assembler generates the bytes for an instruction, it uses the `top_scope_declared` attribute to encode it. Listing 5.2 shows an excerpt of the `variable_dependent` assembler responsible for encoding instructions.

Listing 5.2 Encoding instructions at assembly time

```
class Instruction:
    def encode(self, instr):
        key = self.info['top_scope_declared']
        return [byte ^ key for byte in instr]
```

The loaded virtual machine stores the bytecode as a list of individual bytes. The function `b_next` shown in Listing 5.3 is used in the virtual machine to retrieve these bytes to load instructions. As it loads bytes, it XORs them with a key that is set before every instruction is fetched, as shown in the VM excerpt in Listing 5.4. This key is set to number of items in the topmost variable store, the context information. Our implementation assumes that the size of the key will at most be 255. The modulus of the key with 255 could be used to encode and decode instructions for larger programs that cross this key size limit.

Listing 5.3 Virtual machine fetching and decoding bytecode

```
function b_next() {
    return bytecode[instruction _index++] ^ key;
}
```

Listing 5.4 Decode key being set before instructions are fetched

```
while (instruction_index < bytecode.length) {
    key = Object.keys(variables[variables.length - 1]).length;
    let op = b_next();
...
```

5.2.2 Proposed control flow obfuscations

In this section we propose two possible context-dependent obfuscations that use control flow as context information. We did not implement these obfuscations due to time constraints. These obfuscations incidentally protect program control flow integrity, as only valid control flow is used to decode instructions.

Previous execution locations as context

This obfuscation would encode each instruction i with the locations of all possible instructions that could have executed immediately prior to i.

This obfuscation could only be applied to instructions within a function, with the first instruction of a function being unobfuscated. This is because it is impossible to know at compile time from where a function could be called. Functions could also be called by native JavaScript functions, such as event handlers, which are not obfuscated. The proposed implementation of this obfuscation assumes the compiler does not generate computed jump instructions. OJC currently never generates computed jumps.

A control flow graph could be constructed from the original instructions. If an instruction is not the first instruction within a basic block, then its previous instruction will always be the instruction prior to it in the block. Otherwise, each basic block in the graph will have a list of possible blocks that could have executed before it. The possible prior instructions of the first instruction of each block will always be the last instruction of any of the possible previous blocks. As such, the location of possible previously executed instructions to any instruction i is known at compile time.

This would involve more complex f and g functions as the context information is now a list of possible contexts. One solution to this problem is that multiple instructions could be generated for each piece of context and grouped together, as shown in the equations below. The decode function would then somehow take this group of instructions and the context and unobfuscate the instruction.

$$f(i, \{c_1, c_2, \ldots, c_n\}) \rightarrow \{y_1, y_2, \ldots, y_n\}$$
$$g(\{y_1, y_2, \ldots, y_n\}, c \in \{c_1, c_2, \ldots, c_n\}) \rightarrow i$$

Basic block merging

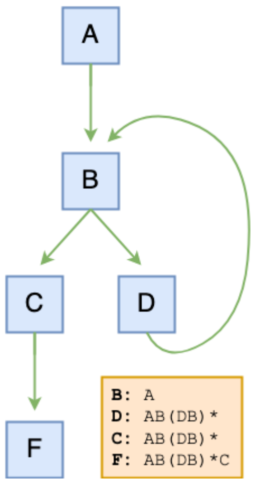


Figure 5.1: Example of basic blocks within a function. Regular expressions are used to encode valid sequences of previously executed blocks.

Some pairs of basic blocks can be uniquely identified by the sequence of jumps taken to get to them. To find blocks that are uniquely identified this way, consider the sequence of jumps encoded as a regular expression. An example of this is shown in Figure 5.1. We will call these expressions path regular expressions, as they encode the path required to get to a block.

Two blocks can be uniquely identified between the paths taken to get to them if the intersection of the languages of their path regular expressions are empty. The language of a regular expression is the complete list of strings the regular expression accepts. For example, blocks D and C in Figure 5.1 could not be differentiated by the control flow taken to get to them, but blocks D and F could.

Two blocks that satisfy this unique control flow property could be 'merged' at assembly time. Let us pick two mergeable blocks a and b, with block b having more instructions than block a. Merging these blocks would involve the following steps. These steps assume the virtual machine keeps track of control flow within the currently executing function.

1. Merging blocks in program instructions:
    a) Path regular expressions for a and b are calculated. We will denote these a_{path} and b_{path} respectively.
    b) Copies of the instructions of blocks a and b are backed up as we will delete and overwrite these blocks later.
    c) All instructions that jump to a are rewritten to jump to b instead. b was chosen as it is the larger block.
    d) Block a is deleted from the program.
    e) All instructions in block b are overwritten with a block of special one byte `merge` instructions. This is now a merged block and we will denote it ab.
2. Handling merged block in virtual machine:
    a) When the virtual machine reaches a `merge` instruction, code is inserted that checks the recorded control flow against the a_{path} and b_{path} regular expressions.
        i. If previous control flow matches b_{path}, the `merge` instructions in block ab will be overwritten in the running VM with the original instructions of block b that were saved in step 1b.
        ii. If previous control flow matches a_{path}, the `merge` instructions in block ab will be overwritten in the running VM with the original instructions of block a that were saved in step 1b. `nop` instructions will be used to overwrite the remaining `merge` instructions.
    b) When the virtual machine reads the final byte of block ab, code is inserted that overwrites the the instructions of this block with `merge` instructions again.

Figure 5.2 shows an example of program control flow after applying basic block merging.

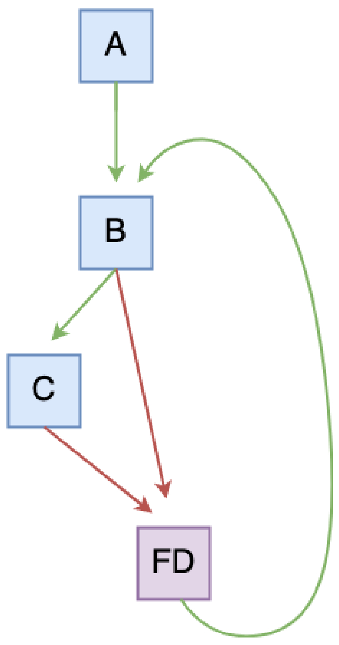


Figure 5.2: Example of basic block merging. Blocks F and D in Figure 5.1 have been merged. Red paths are the jumps that have been rewritten to the new merged block.

Chapter 6

Evaluation

In this section we will discuss the security properties and performance impacts of context-dependent instructions.

6.1 Analysis of Security Properties

Collberg et al. in their paper *A Taxonomy of Obfuscating Transformations* present the following 3 key metrics to use when analysing the strength of an obfuscation technique:

- **Potency** measures how difficult it is for a human reviewer to understand the code generated by the obfuscation.
- **Resilience** measures how effective the obfuscation is against automatic deobfuscation techniques (existing or new).
- **Cost** measures how much performance overhead the obfuscation introduces.

Virtualisation obfuscation highly potent, resilient and costly. It is potent as a human reviewer must understand the virtual machine architecture and instruction set to understand the instructions that make up the program, after unobfuscating the VM itself. It is resilient as automatic attacks may be able to recover VM bytecode and disassemble it (as the Kasada VM reverse engineering tool did as shown in Figure 1.1), but a human reviewer will still need to go through and understand the disassembled instructions to make sense of the program. Doing this again requires knowledge of the VM architecture and instruction set. Resilience is further bolstered with addition of polymorphic bytecode. The price of this is the cost metric, with virtualisation obfuscation severely degrading the performance of the original program.

Our implementation of context-dependent instructions further improves the resilience metric through deterring static analysis of VM bytecode. Bytecode assembly itself does little to deter attackers from statically disassembling VM payloads. Attackers simply need to trace VM execution to understand how different instructions are packaged, then implement this functionality themselves to create a static disassembler. However, an attacker attempting to do this with our implementation of context-dependent instructions will observe that the virtual machine uses its constantly changing runtime state (context) to decode instructions. This prevents attackers from ripping and using instruction parsing code straight from the VM to build a static disassembler.

The attacker has two options to defeat this protection:

- **Brute force attack.** They key space of this obfuscation is the one byte, and so is easy to brute force. However, there is no way to tell from pure static analysis that an instruction has decoded correctly. As such, the attacker must either guess that they have decoded an instruction correctly and move on, or confirm their guess through tracing the running virtual machine. As instructions are variable lengths, if the attacker decoded an instruction incorrectly and attempts to continue disassembly, they will most likely decode the following instructions incorrectly as well.

- **Custom disassembler.** The attacker could implement a smart disassembler that keeps track of when variables and stores are created and deleted. It is possible to write a disassembler that entirely defeats this obfuscation with this technique, but doing so would required the attacker to completely understand the virtual machine variable store system and instruction set. This could take a significant reverse engineering effort, especially if the virtual machine is itself obfuscated well.

Some attempts to deter brute force attacks could include spreading op codes across two or more bytes and randomly creating copies of existing operations to artificially increase the number of op codes. When the disassembler wants to encode an operation, it could randomly pick one of the many op codes that corresponds to it. The VM would need to be appropriately modified as well to include these copies in its operation list. This would increase the key space of valid op codes that the attacker would need to brute force and also increase the randomness of created bytecode.

This obfuscation however, can be completely defeated through building a custom disassembler that keeps track of variable movements. We still consider this to be a resilient obfuscation due to the effort required to build such a tool.

Our implementation is also created to be a simple working proof of concept of context-dependent obfuscations. We hypothesise that using this technique on more complex pieces of context such as control flow (as discussed in Section 5.2.2) paves the way for more resilient obfuscations. We also hypothesise that using control flow as context provides increased code protection, as control flow is difficult to spoof.

6.2 Performance Impact

It is important to note that the purpose of this study is not a performance analysis of virtualisation obfuscation as a whole, neither is it a benchmark study of context-dependent obfuscation specifically. Due to time constraints, no effort has been made to optimise our implementation of context-dependent instructions or the Orange Juice Compiler as a whole.

The decode step of context-dependent obfuscation is the function $g(y, c)$, with g being the decode function, y being the obfuscated instruction and c being the context the instruction is encoded with. The performance cost of this comes from retrieving the context c and performing the function g. The cost of retrieving the obfuscated instruction y should only be considered if the process of doing so is different to a normal bytecode virtual machine. Our implementation, apart from performing an XOR operation on retrieved bytes (this being the g function), did not store or retrieve the instructions differently from a standard bytecode VM. Therefore, the cost of retrieving y would not be part of the performance overhead of the obfuscation.

6.2.1 Benchmark selection

Benchmarks were undertaken comparing the performance of virtual machines created by the `basic_bytecode` and `variable_dependent` assemblers.

The `basic_bytecode` assembler is the baseline bytecode assembler of OJC. It directly assembles the instructions created by the compiler into the format shown in Figure 4.2 and does not apply any further obfuscation. The `variable_dependent` assembler was created by extending the `basic_bytecode` assembler. It adds context-dependent obfuscation as discussed earlier and randomised op codes.

The benchmarks selected were the Richards, Deltablue and Crypto benchmarks from the Octane benchmark suite. These benchmarks were selected primarily as OJC was able to compile them. Here are some details regarding these benchmarks:

- **Richards.** OS kernel simulation benchmark, main focus is testing function calls and property storing/loading. 539 lines.
- **Deltablue.** One way constraint solver, main focus is testing polymorphism. 880 lines.
- **Crypto.** RSA encryption and decryption, main focus is testing bit operations. 1689 lines.

6.2.2 Operating system, benchmark and hardware setup

Benchmarks were set up with benchmark.js, a benchmarking library that supports high-resolution timers and automatically re-runs tests until it has enough data to output accurate results. Benchmarks were run on NodeJS which uses the V8 JavaScript engine.

Benchmarks were run on a computer running macOS Big Sur, with a 2.3 Ghz 8-Core Intel Core i9 processor and 16 GB of 2667 Mhz DDR4 memory.

6.2.3 Results

Table 6.1: This table lists benchmark results.

	Samples counted	`basic_bytecode` Execution time (s)	`variable_dependent` Execution time (s)	Performance loss
Richards	14	0.24 (±3.69%)	0.30 (±3.50%)	20.0%
Deltablue	10	0.42 (±2.35%)	0.47 (±2.44%)	10.6%
Crypto	5	17.21 (±2.49%)	22.38 (±4.09%)	23.1%

Performance loss varied significantly between these benchmarks. Further analysis is needed to understand significant difference between the performance loss of the Deltablue benchmark compared to the Richards and Crypto benchmark. In future, we aim to update OJC to support more benchmarks as more data is needed to make further statements on the performance impacts of our implementation of context-dependent obfuscations.

Performance loss could also be reduced through not recalculating the decode key at the start of every instruction, as this is unneeded. Knowing that only certain instructions change the number of variables in the topmost store (`setvar`, `getvar`, `delvar`, `push_store`, `pop_store`, bytecode function calls and `return`), the key should only be recalculated after one of these instruction execute.

Chapter 7

Related Work

In their 2019 paper *Leveraging WebAssembly for Numerical JavaScript Code Virtualization* Wang et al. present JSPro, a virtualisation system for JavaScript. This work translates JavaScript code to WebAssembly, then obfuscates the compiled WebAssembly binary. This works obfuscates purely numerical, computation based JavaScript code as this is the only JavaScript syntax JSPro appears to support. As such, this technique cannot be used to obfuscate code that is used to interact with the webpage or called by event handlers.

Obfuscated WebAssembly can also be generated using Tigress. Tigress is a well known source-to-source obfuscator and virtualiser C programs. To produce WebAssembly code, C programs can be obfuscated through Tigress then compiled to WebAssembly using emscripten, a LLVM to WebAssembly compiler. One of the available transformations Tigress provides is virtualisation obfuscation. The Tigress virtualiser supports an incredible range of obfuscation mechanisms that result in a highly diverse range of output virtual machines. Some of these mechanisms include standard obfuscations such as randomised op codes and duplicate instructions. Other mechanisms are more complex, including randomly generating the instruction set architecture of the virtual machine. As virtualisation obfuscation is already a very costly JavaScript obfuscation, care must be taken in developing and applying obfuscations that further introduce heavy performance penalties.

Kuang et al. present DSVMP, a machine code virtualisation obfuscator designed to randomly choose execution paths at program runtime. This effectively deters dynamic analysis of obfuscated code, as control flow changes unpredictably during re-runs of the same program. This obfuscation is similar in nature to obfuscations that produce bytecode polymorphism, the goal of which is to generate many different but functionally equivalent variants of the VM to increase attacker workload. Polymorphism is more relevant in JavaScript context as JavaScript code is repeatedly transferred to clients, so different VM variants can be sent on different requests. The DSVMP approach effectively simulates this for traditional install-once software.

Kochberger et al. present an in-depth investigation into deobfuscation tools for machine code virtualisation obfuscators. While many of the specific methodologies these tools use do not apply to JavaScript virtualisation, many of the conceptual frameworks they use do. Most tools attempt to detect the virtual machine and understand its instruction set, then create a mapping from virtual machine instructions to a suitable intermediate representation (IR). This IR can then be simplified then converted to regular machine code, which can be analysed by a plethora of tools. Much work has been done to develop algorithms that can automate various parts of this process, such VM identification and instruction extraction.

JavaScript virtualisation obfuscation provides an interesting contrast to standard machine code virtualisation obfuscation. The dynamic, expressive syntax and behaviour of JavaScript allows extensive obfuscations to be developed on JavaScript virtual machines, such as the context dependent obfuscations discussed in this report. In contrast to low-level virtual machines however, JavaScript virtual machines must produce code that interfaces with native JavaScript, to allow support for event handlers and JavaScript based website interactivity. It is also far easier for attackers to retrieve JavaScript virtual machines. This is as JavaScript code is executed by a client that an attacker has full control over, unlike a CPU. Attackers could even build custom browsers to assist in defeating obfuscations.

To increase the obfuscation potential of JavaScript virtual machines and mitigate against automatic virtual machine deobfuscation algorithms, effort must be made to create techniques that deter static analysis in novel ways using the expressive syntax of JavaScript.

Chapter 8

Concluding Remarks

8.1 Future Work

There are many avenues of future work on this topic.

- As discussed in Section 5.2.2, context-dependent instructions that use previous execution locations as context and basic block merging could be implemented and analysed. Using multiple layers of context-dependent obfuscations could also be implemented and analysed.
- A benchmark study of virtualisation obfuscation could be conducted to find what most influences performance costs of code obfuscated this way. This would involve adding support for more benchmarks to the Orange Juice Compiler and possibly instrumenting a JavaScript runtime to trace performance issues.
- Real-world JavaScript virtualisation obfuscated code could be retrieved from websites that use obfuscation. The virtual machines and instruction sets could be analysed for any interesting obfuscation techniques that they may use.
- Research could be conducted on attempting to automatically decompile JavaScript virtual machine instructions. We hypothesise that many JavaScript VMs might have similar instruction sets. If this is true, building a general purpose JavaScript virtual machine decompiler could be worthwhile effort.

8.2 Conclusion

We present three contributions to the obfuscation community:

1. We present the Orange Juice Compiler, a JavaScript virtualisation obfuscation framework developed to do obfuscation research with.

2. We define context-dependent obfuscation as an obfuscation technique that encodes every virtual machine instruction with information about the state of the running program immediately prior to its execution. We propose advanced context-dependent obfuscation techniques that obfuscate program control flow and 'merge' basic blocks of compiled program together.

3. We present a proof of concept context-dependent obfuscation implemented within the Orange Juice Compiler. We analyse the security properties of this implementation and find it reasonably effective in deterring static disassembly. We benchmark this implementation with a subset of the Octane benchmarks and find imposes a performance overhead of 10% to 20%.

Appendix A

Appendix: Sample Orange Juice Compiler Output

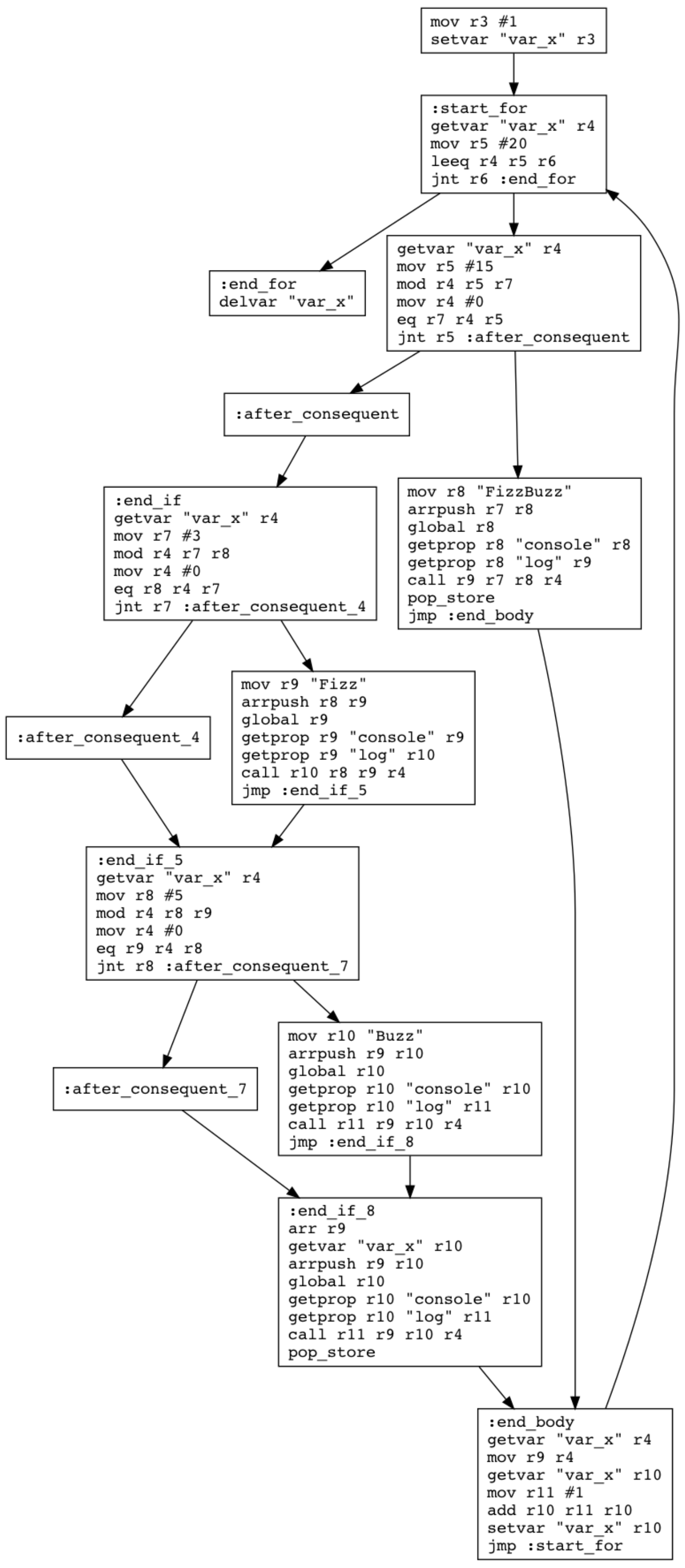
In this appendix we will present some sample output from the Orange Juice Compiler. All programs were compiled with debug mode to generate descriptive label and variable names. Disassembly graphs were generated automatically.

A.1 Fizz Buzz

Listing A.1 Fizz Buzz program

```
for (let x=1; x <= 20; x++) {
  if (x % 15 == 0) {
    console.log('FizzBuzz');
    continue;
  }
  if (x % 3 == 0) {
    console.log('Fizz');
  }
  if (x % 5 == 0) {
    console.log('Buzz');
  }
  console.log(x);
}
```

Listing A.1 compiles to the disassembly shown in Figure A.1.



```
mov r3 #1
setvar "var_x" r3
```

```
:start_for
getvar "var_x" r4
mov r5 #20
leeq r4 r5 r6
jnt r6 :end_for
```

```
:end_for
delvar "var_x"
```

```
getvar "var_x" r4
mov r5 #15
mod r4 r5 r7
mov r4 #0
eq r7 r4 r5
jnt r5 :after_consequent
```

```
:after_consequent
```

```
mov r8 "FizzBuzz"
arrpush r7 r8
global r8
getprop r8 "console" r8
getprop r8 "log" r9
call r9 r7 r8 r4
pop_store
jmp :end_body
```

```
:end_if
getvar "var_x" r4
mov r7 #3
mod r4 r7 r8
mov r4 #0
eq r8 r4 r7
jnt r7 :after_consequent_4
```

```
:after_consequent_4
```

```
mov r9 "Fizz"
arrpush r8 r9
global r9
getprop r9 "console" r9
getprop r9 "log" r10
call r10 r8 r9 r4
jmp :end_if_5
```

```
:end_if_5
getvar "var_x" r4
mov r8 #5
mod r4 r8 r9
mov r4 #0
eq r9 r4 r8
jnt r8 :after_consequent_7
```

```
:after_consequent_7
```

```
mov r10 "Buzz"
arrpush r9 r10
global r10
getprop r10 "console" r10
getprop r10 "log" r11
call r11 r9 r10 r4
jmp :end_if_8
```

```
:end_if_8
arr r9
getvar "var_x" r10
arrpush r9 r10
global r10
getprop r10 "console" r10
getprop r10 "log" r11
call r11 r9 r10 r4
pop_store
```

```
:end_body
getvar "var_x" r4
mov r9 r4
getvar "var_x" r10
mov r11 #1
add r10 r11 r10
setvar "var_x" r10
jmp :start_for
```

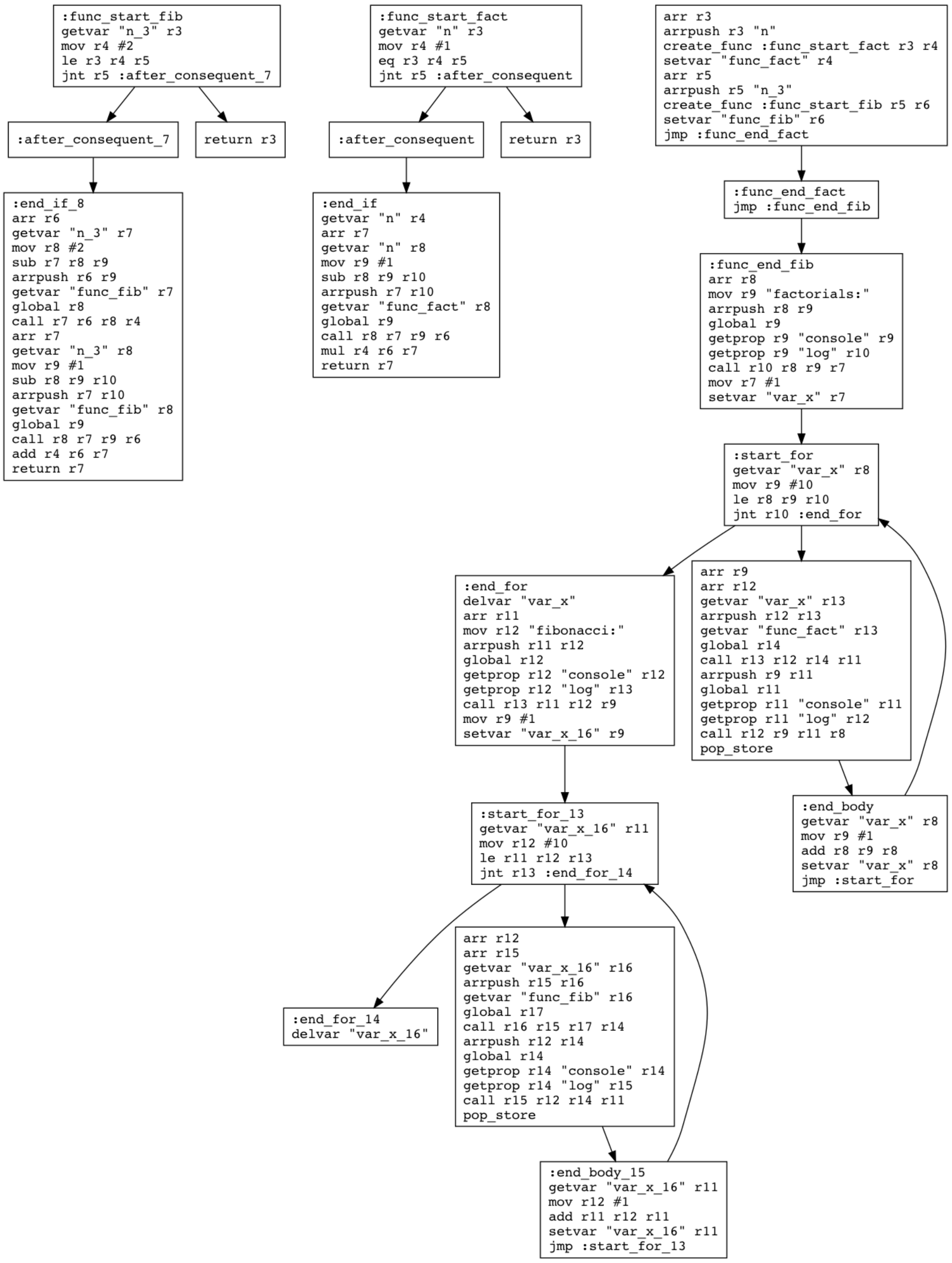
Figure A.1: Generated disassembly of Fizz Buzz program in Listing A.1

A.2 Fibonacci and Factorials

Listing A.2 Program displaying first ten fibonacci numbers and factorials

```
function fact(n) {
    if (n==1) {
        return 1;
    }
    return n*fact(n-1);
}

function fib(n) {
    if (n < 2) {
        return 1;
    }
    return fib(n-2) + fib(n-1);
}

console.log('factorials:')
for (let x = 1; x<10; x+=1) {
    console.log(fact(x));
}

console.log('fibonacci:')
for (let x = 1; x<10; x+=1) {
  console.log(fib(x));
}
```

Listing A.2 compiles to the disassembly shown in Figure A.2.



```
:func_start_fib
getvar "n_3" r3
mov r4 #2
le r3 r4 r5
jnt r5 :after_consequent_7
```

```
:after_consequent_7
```

```
return r3
```

```
:end_if_8
arr r6
getvar "n_3" r7
mov r8 #2
sub r7 r8 r9
arrpush r6 r9
getvar "func_fib" r7
global r8
call r7 r6 r8 r4
arr r7
getvar "n_3" r8
mov r9 #1
sub r8 r9 r10
arrpush r7 r10
getvar "func_fib" r8
global r9
call r8 r7 r9 r6
add r4 r6 r7
return r7
```

```
:func_start_fact
getvar "n" r3
mov r4 #1
eq r3 r4 r5
jnt r5 :after_consequent
```

```
:after_consequent
```

```
return r3
```

```
:end_if
getvar "n" r4
arr r7
getvar "n" r8
mov r9 #1
sub r8 r9 r10
arrpush r7 r10
getvar "func_fact" r8
global r9
call r8 r7 r9 r6
mul r4 r6 r7
return r7
```

```
arr r3
arrpush r3 "n"
create_func :func_start_fact r3 r4
setvar "func_fact" r4
arr r5
arrpush r5 "n_3"
create_func :func_start_fib r5 r6
setvar "func_fib" r6
jmp :func_end_fact
```

```
:func_end_fact
jmp :func_end_fib
```

```
:func_end_fib
arr r8
mov r9 "factorials:"
arrpush r8 r9
global r9
getprop r9 "console" r9
getprop r9 "log" r10
call r10 r8 r9 r7
mov r7 #1
setvar "var_x" r7
```

```
:start_for
getvar "var_x" r8
mov r9 #10
le r8 r9 r10
jnt r10 :end_for
```

```
:end_for
delvar "var_x"
arr r11
mov r12 "fibonacci:"
arrpush r11 r12
global r12
getprop r12 "console" r12
getprop r12 "log" r13
call r13 r11 r12 r9
mov r9 #1
setvar "var_x_16" r9
```

```
arr r9
arr r12
getvar "var_x" r13
arrpush r12 r13
getvar "func_fact" r13
global r14
call r13 r12 r14 r11
arrpush r9 r11
global r11
getprop r11 "console" r11
getprop r11 "log" r12
call r12 r9 r11 r8
pop_store
```

```
:end_body
getvar "var_x" r8
mov r9 #1
add r8 r9 r8
setvar "var_x" r8
jmp :start_for
```

```
:start_for_13
getvar "var_x_16" r11
mov r12 #10
le r11 r12 r13
jnt r13 :end_for_14
```

```
:end_for_14
delvar "var_x_16"
```

```
arr r12
arr r15
getvar "var_x_16" r16
arrpush r15 r16
getvar "func_fib" r16
global r17
call r16 r15 r17 r14
arrpush r12 r14
global r14
getprop r14 "console" r14
getprop r14 "log" r15
call r15 r12 r14 r11
pop_store
```

```
:end_body_15
getvar "var_x_16" r11
mov r12 #1
add r11 r12 r11
setvar "var_x_16" r11
jmp :start_for_13
```

Figure A.2: Generated disassembly of the Fibonacci and factorials program in Listing A.2

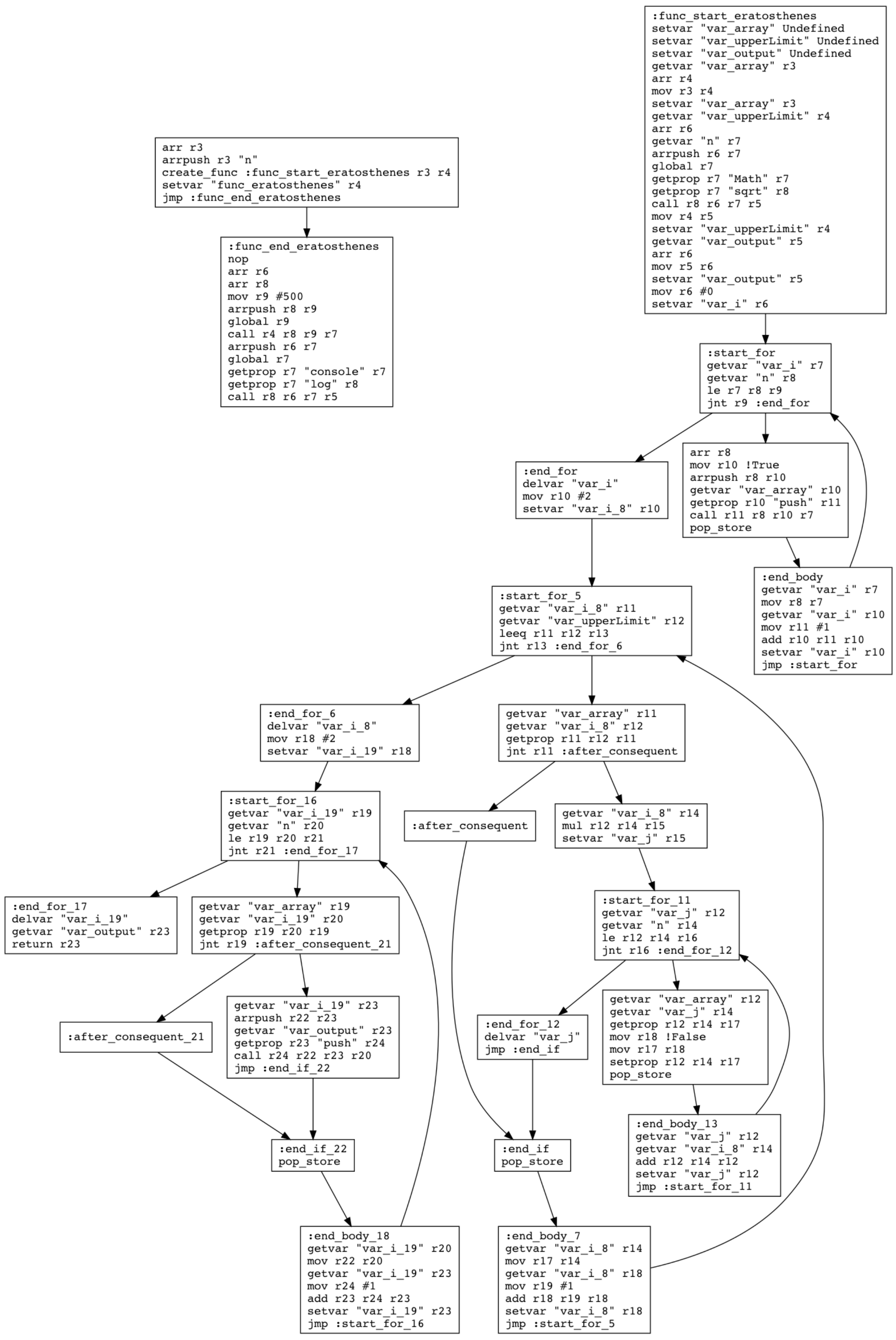
A.3 Sieve of Eratosthenes

Listing A.3 Sieve of Eratosthenes program

```
function eratosthenes(n) {
    // Eratosthenes algorithm to find all primes under n
    var array = [], upperLimit = Math.sqrt(n), output = [];

    // Make an array from 2 to (n - 1)
    for (var i = 0; i < n; i++) {
        array.push(true);
    }

    // Remove multiples of primes starting from 2, 3, 5,...
    for (var i = 2; i <= upperLimit; i++) {
        if (array[i]) {
            for (var j = i * i; j < n; j += i) {
                array[j] = false;
            }
        }
    }

    // All array[i] set to true are primes
    for (var i = 2; i < n; i++) {
        if(array[i]) {
            output.push(i);
        }
    }

    return output;
};

console.log(eratosthedknes(500));
```

The program above was copied from this Stack Overflow answer: https://stackoverflow.com/a/15471749.

Listing A.3 compiles to the disassembly shown in Figure A.3.



```
arr r3
arrpush r3 "n"
create_func :func_start_eratosthenes r3 r4
setvar "func_eratosthenes" r4
jmp :func_end_eratosthenes
```

```
:func_end_eratosthenes
nop
arr r6
arr r8
mov r9 #500
arrpush r8 r9
global r9
call r4 r8 r9 r7
arrpush r6 r7
global r7
getprop r7 "console" r7
getprop r7 "log" r8
call r8 r6 r7 r5
```

```
:func_start_eratosthenes
setvar "var_array" Undefined
setvar "var_upperLimit" Undefined
setvar "var_output" Undefined
getvar "var_array" r3
arr r4
mov r3 r4
setvar "var_array" r3
getvar "var_upperLimit" r4
arr r6
getvar "n" r7
arrpush r6 r7
global r7
getprop r7 "Math" r7
getprop r7 "sqrt" r8
call r8 r6 r7 r5
mov r4 r5
setvar "var_upperLimit" r4
getvar "var_output" r5
arr r6
mov r5 r6
setvar "var_output" r5
mov r6 #0
setvar "var_i" r6
```

```
:start_for
getvar "var_i" r7
getvar "n" r8
le r7 r8 r9
jnt r9 :end_for
```

```
:end_for
delvar "var_i"
mov r10 #2
setvar "var_i_8" r10
```

```
arr r8
mov r10 !True
arrpush r8 r10
getvar "var_array" r10
getprop r10 "push" r11
call r11 r8 r10 r7
pop_store
```

```
:end_body
getvar "var_i" r7
mov r8 r7
getvar "var_i" r10
mov r11 #1
add r10 r11 r10
setvar "var_i" r10
jmp :start_for
```

```
:start_for_5
getvar "var_i_8" r11
getvar "var_upperLimit" r12
leeq r11 r12 r13
jnt r13 :end_for_6
```

```
:end_for_6
delvar "var_i_8"
mov r18 #2
setvar "var_i_19" r18
```

```
getvar "var_array" r11
getvar "var_i_8" r12
getprop r11 r12 r11
jnt r11 :after_consequent
```

```
:start_for_16
getvar "var_i_19" r19
getvar "n" r20
le r19 r20 r21
jnt r21 :end_for_17
```

```
:after_consequent
```

```
getvar "var_i_8" r14
mul r12 r14 r15
setvar "var_j" r15
```

```
:end_for_17
delvar "var_i_19"
getvar "var_output" r23
return r23
```

```
getvar "var_array" r19
getvar "var_i_19" r20
getprop r19 r20 r19
jnt r19 :after_consequent_21
```

```
:start_for_11
getvar "var_j" r12
getvar "n" r14
le r12 r14 r16
jnt r16 :end_for_12
```

```
getvar "var_i_19" r23
arrpush r22 r23
getvar "var_output" r23
getprop r23 "push" r24
call r24 r22 r23 r20
jmp :end_if_22
```

```
:after_consequent_21
```

```
:end_for_12
delvar "var_j"
jmp :end_if
```

```
getvar "var_array" r12
getvar "var_j" r14
getprop r12 r14 r17
mov r18 !False
mov r17 r18
setprop r12 r14 r17
pop_store
```

```
:end_if_22
pop_store
```

```
:end_if
pop_store
```

```
:end_body_13
getvar "var_j" r12
getvar "var_i_8" r14
add r12 r14 r12
setvar "var_j" r12
jmp :start_for_11
```

```
:end_body_18
getvar "var_i_19" r20
mov r22 r20
getvar "var_i_19" r23
mov r24 #1
add r23 r24 r23
setvar "var_i_19" r23
jmp :start_for_16
```

```
:end_body_7
getvar "var_i_8" r14
mov r17 r14
getvar "var_i_8" r18
mov r19 #1
add r18 r19 r18
setvar "var_i_8" r18
jmp :start_for_5
```

Figure A.3: Generated disassembly of the sieve of Eratosthenes program in Listing A.3

Appendix B

Appendix: Orange Juice Compiler Instruction Reference

A literal can be either a string, number, float, register, Boolean, null or undefined. See Figure 4.2 to see how these arguments are encoded into bytecode.

The OJC virtual machine implements the following 48 operations.

global `<dst>`	Move JavaScript global object into register `dst`
mov `<dst> <src>`	Move `src` to register `dst` `src` can be either a register or literal
add `<a> <b> <dst>`	Calculate `a+b` and store in register `dst` `a` and `b` can be either registers or number literals
mul `<a> <b> <dst>`	Calculate `a*b` and store in register `dst` `a` and `b` can be either registers or number literals
div `<a> <b> <dst>`	Calculate `a/b` and store in register `dst` `a` and `b` can be either registers or number literals
jmp `<dst>`	Jump execution to address `dst`
jt `<cond> <dst>`	If value of register `cond` is truthy, jump to address `dst`
jnt `<cond> <dst>`	If value of register `cond` is falsy, jump to address `dst`
setvar `<var> <value>`	Set variable `var` to `value` in variable mappings `var` must be a string `value` can be either a register or literal

getvar `<var> <dst>`	Retrieve variable `var` to register `dst` `var` must be a string
delvar `<var>`	Delete variable `var` `var` must be a string
delvar `<var>`	Delete variable `var` `var` must be a string
arr `<dst>`	Load empty array into register `var`
arrpush `<arr> <item>`	Push `item` into array stored in register `arr` `item` can be either a register or literal
obj `<dst>`	Load empty object into register `var`
setprop `<obj> <prop> <value>`	Set property `prop` on object stored in register `obj` to `value` `prop` and `value` can both be either registers or literals Instruction detailed in Section 4.3.2
getprop `<obj> <prop> <dst>`	Retrieve property `prop` on object stored in register `obj` to register `dst` `prop` can be either a register or literal Instruction detailed in Section 4.3.2
eq `<r1> <r2> <ans>`	Check if `r1 == r2`, store result in register `ans` `r1` and `r2` must both be registers
neq `<r1> <r2> <ans>`	Check if `r1 != r2`, store result in register `ans` `r1` and `r2` must both be registers
eqt `<r1> <r2> <ans>`	Check if `r1 === r2`, store result in register `ans` `r1` and `r2` must both be registers
neqt `<r1> <r2> <ans>`	Check if `r1 !=== r2`, store result in register `ans` `r1` and `r2` must both be registers
ge `<r1> <r2> <ans>`	Calculate `r1 > r2`, store result in register `ans` `r1` and `r2` must both be registers
geeq `<r1> <r2> <ans>`	Calculate `r1 >= r2`, store result in register `ans` `r1` and `r2` must both be registers
le `<r1> <r2> <ans>`	Calculate `r1 < r2`, store result in register `ans` `r1` and `r2` must both be registers

`leeq <r1> <r2> <ans>`	Calculate `r1 <= r2`, store result in register `ans` `r1` and `r2` must both be registers
`mod <r1> <r2> <ans>`	Calculate `r1` % `r2`, store result in register `ans` `r1` and `r2` must both be registers
`shl <r1> <r2> <ans>`	Calculate `r1 << r2`, store result in register `ans` `r1` and `r2` must both be registers
`shr <r1> <r2> <ans>`	Calculate `r1 >> r2`, store result in register `ans` `r1` and `r2` must both be registers
`ushr <r1> <r2> <ans>`	Calculate `r1 >>> r2`, store result in register `ans` `r1` and `r2` must both be registers
`bit_and <r1> <r2> <ans>`	Calculate `r1` & `r2`, store result in register `ans` `r1` and `r2` must both be registers
`bit_or <r1> <r2> <ans>`	Calculate `r1 | r2`, store result in register `ans` `r1` and `r2` must both be registers
`xor <r1> <r2> <ans>`	Calculate `r1`^`r2`, store result in register `ans` `r1` and `r2` must both be registers
`pow <r1> <r2> <ans>`	Calculate `r1 ** r2`, store result in register `ans` `r1` and `r2` must both be registers
`inside <r1> <r2> <ans>`	Check if `r1 in r2`, store result in register `ans` `r1` and `r2` must both be registers
`check_instance <r1> <r2> <ans>`	Check if `r1 instanceof r2`, store result in register `ans` `r1` and `r2` must both be registers
`unary_plus <r1> <ans>`	Calculate `+r1`, store result in register `ans` `r1` must be a register
`unary_neg <r1> <ans>`	Calculate `-r1`, store result in register `ans` `r1` must be a register
`unary_not <r1> <ans>`	Calculate `!r1`, store result in register `ans` `r1` must be a register
`unary_bit_not <r1> <ans>`	Calculate `~r1`, store result in register `ans` `r1` must be a register
`unary_typeof <r1> <ans>`	Calculate `typeof r1`, store result in register `ans` `r1` must be a register

`nop`	No operation - do nothing
`whatsthis` `<dst>`	Load current value of `this` into register `dst`
`create_func` `<f_start> <args> <r>`	Create wrapper function and store in register `r` `f_start` must be the address of bytecode function `args` must be a register containing an array of function parameters Instruction detailed in Section 4.3.2
`return` `<val>`	Return `val` from function `val` can be either register or literal
`push_store`	Push new variable store Instruction detailed in Section 4.3.2
`pop_store`	Pop topmost variable store Instruction detailed in Section 4.3.2
`regex` `<pattern> <flags> <reg>`	Create regex object and store in register `reg` `pattern` and `flags` can be either registers or literals

Bibliography

Collberg, C.; Thomborson, C.; and Low, D., 1997. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand. [Cited on page 31.]

Imperva, 2019. What is a sneaker bot. https://www.imperva.com/learn/application-security/sneaker-bot/. [Cited on page 1.]

Imperva, 2022. 2022 Imperva Bad Bot Report. Technical report. [Cited on page 1.]

Kochberger, P.; Schrittwieser, S.; Schweighofer, S.; Kieseberg, P.; and Weippl, E., 2021. Sok: Automatic deobfuscation of virtualization-protected applications. In *The 16th International Conference on Availability, Reliability and Security*, 1–15. [Cited on page 36.]

Kuang, K.; Tang, Z.; Gong, X.; Fang, D.; Chen, X.; and Wang, Z., 2018. Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling. *Computers & Security*, 74 (2018), 202–220. [Cited on page 35.]

Mohapatra, S., 2013. E-commerce strategy. In *E-Commerce Strategy*, 155–171. Springer. [Cited on page 1.]

OPCODES, 2021. Reverse engineering kasada javascript vm obfuscation. https://opcodes.fr/publications/2021-08/kasada-javascript-vm-obfuscation-reverse-part1. [Cited on page 2.]

OWASP, 2020. Oat-005 scalping. https://owasp.org/www-project-automated-threats-to-web-applications/assets/oats/EN/OAT-005_Scalping.html. [Cited on page 1.]

Wang, S.; Ye, G.; Li, M.; Yuan, L.; Tang, Z.; Wang, H.; Wang, W.; Wang, F.; Ren, J.; Fang, D.; et al., 2019. Leveraging webassembly for numerical javascript code virtualization. *IEEE Access*, 7 (2019), 182711–182724. [Cited on page 35.]

Xu, D.; Ming, J.; and Wu, D., 2016. Generalized dynamic opaque predicates: A new control flow obfuscation method. In *International Conference on Information Security*, 323–342. Springer. [Cited on page 2.]