Analysis and Optimization of Truetype Font Bytecode

by

Zeming Liu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Math
in
Computer Science

Waterloo, Ontario, Canada, 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

TrueType is one of the most widely used vectorized font formats. It can be optimally rendered on screens with different resolutions and different font sizes thanks to hints expressed as bytecode programs. Font engines execute the bytecode programs to adjust the outlines of the glyphs. TrueType font bytecode is a highly-dynamic stack-based bytecode language. It manipulates data with a global stack, and it uses hardware related information, such as screen resolutions and font sizes, which are unknown at compile time. Thus, it is hard to perform static analysis and optimizations on this bytecode.

Fonts are sometimes subsetted to only include the glyphs that appear in a webpage before sending to the client. Existing font manipulation techniques do not touch the bytecode, so subsetted fonts contain un-optimized bytecode programs. TrueType bytecode analysis can help reduce bandwidth demands for serving webpages.

This thesis presents improvements to COI, a tool for manipulating TrueType bytecode. New features include enhanced abstract execution as well as basic optimizations on COI, such as tree shaking, no-effect instruction removal, and dead block elimination. Finally, it completes the cycle by translating the COI back to TrueType bytecode.

We tested our tool on fonts from different font families, including Microsoft Core TrueType font Arial, and NotoSansTibetan-Bold. Our experiments show that our optimizations can reduce the size of bytecode by 0.37% to 18.82% of the test fonts in our benchmarks. On average, we can reduce the size of bytecode of our test fonts by 7.10%. Our optimized fonts yield the same bitmaps as the original font.

Acknowledgements

I would like to thank to my advisor, Professor Patrick Lam, without whom, the thesis would not have been possible. I would also like to thank to the readers of the thesis, Professor Gregor Richards and Professor Derek Rayside.

Dedication

I dedicate the thesis to my family which supports me forever, and to all my friends who helped me during my study and life at University of Waterloo.

Table of Contents

List of Tables ix

List of Figures x

1 Introduction 1

1.1 Approach . . . 2

1.2 Contributions of Thesis . . . 2

2 Related Work 4

2.1 Symbolic Execution and Static Analysis . . . 4

2.2 Symbolic Range Propagation . . . 5

2.3 Goto Elimination . . . 5

2.4 Three Address Code . . . 6

2.5 Fonttools and FreeType 2 . . . 7

2.6 COI: A First Step Towards TrueType Bytecode Analysis . . . 7

3 Background 8

3.1 TrueType Font and Font Engines . . . 8

3.2 TrueType Bytecodes . . . 8

3.3 Components of TrueType Font . . . 9

3.4 Glyph Representation in TrueType . . . 10

3.5 How Font Engines Render a Glyph . . . 10

4 Motivation **14**
4.1 TrueType Bytecode Analysis and Optimization 14
4.2 The Difficulty of TrueType Bytecode Analysis 15
4.2.1 Hardware Specification Related Instructions 15
4.2.2 Control Flow Instructions 15
4.3 Enabling Analysis of TrueType Font Bytecode 16

5 Methods **17**
5.1 Overall Structure of Project 17
5.2 Improvements 18
5.3 Symbolic Execution and COI Generation 20
5.3.1 CFG Construction 20
5.3.2 Abstract Executor 20
5.3.3 Stack Depth 21
5.3.4 Uncertain Variables 22
5.3.5 COI Translation 22
5.3.6 Detailed Modeling of Point Moving and Graphics State Instructions 24
5.3.7 Parameters and Return Values of Functions 25
5.3.8 Range Propagation and Comparison 27
5.3.9 Environment Merging 28
5.3.10 Uncertain Callee 29
5.3.11 Vest 32
5.3.12 Unstructured Jumps 33
5.4 Optimizations 43
5.4.1 Uncalled Functions 43
5.4.2 Instructions with No Effects 43
5.4.3 Dead block Elimination 44
5.5 Inverse Conversion and Validation 46

5.6 A Round Trip Example 48
5.7 Limitations 48
5.8 Bytecode Debugger 53

6 Experimental Results **54**
6.1 Uncalled Functions 56
6.2 Instructions Without Effects 57
6.3 Dead Block Elimination 61
6.3.1 Dead Blocks in Functions 61
6.3.2 Dead Blocks in Control Value Program and Glyph Programs 63
6.4 Font Subsetting 64
6.5 Overall Results 65

7 Conclusion **67**

8 Future Work **68**

References **69**

List of Tables

5.1 Resolutions and dpi of popular devices and ppem at point size 9 45

6.1 Statistical information on selected fonts 55

6.2 Uncalled Functions 56

6.3 Instructions which potentially have no effects (the Apple TrueType Font Reference Manual [6]) 58

6.5 No-effect instructions in test fonts 59

6.6 Statistical information on no-effect instructions in our benchmarks when lower bound of ppem is 15 60

6.8 Visits of if-then-else blocks in functions, where the conditions are concrete 62

6.9 Removable dead code in function programs 62

6.10 Removable dead code in control value programs and glyph programs 63

6.11 Bytecode removal with respect to font subsettings when the lower bound of ppem is 30 64

6.12 Uncalled functions with respect to the percentage of subsetting, when the lower bound of ppem is 30 65

6.13 Overall effectiveness of optimizations when lower bound of ppem is 30 . . . 66

List of Figures

3.1 The engine transforms the master outline to device dependent outline by multiplying the coordinates to a scalar. 12

3.2 The engine adjusts the scaled outline by executing the bytecode programs. 13

3.3 The engine (scan converter) turns on the pixels fall within or on the grid-fitted outlines to generate the bitmap. 13

5.1 Overall structure of the project 19

5.2 Assignment statements 23

5.3 Flow diagram of instructions with vests 33

5.4 An example of unconditional forward jump not crossing control flow structures 35

5.5 An example of unconditional backward jump crossing an if else block . . . 35

5.6 An example of unconditional forward jump over a single-branch block . . . 36

5.7 An example of unconditional forward jump over a double-branch if-then-else block 38

5.8 Jump relative on true. We adjust a JROT[] instruction to an if-then-else block where there is an unconditional jump in the then branch 39

5.9 Jump relative on false. We adjust a JROF[] instruction to an if-then-else block where there is an unconditional jump in the else branch 40

5.10 A complicated compound real life bytecode with jump instructions from Arial 41

5.11 A simple bytecode program 49

5.12 COI representation of the bytecode programs 50

5.13 Optimized COI representation 51

5.14 Optimized bytecode programs 52
5.15 A screen shot of our GUI debugger tool. 53

Chapter 1

Introduction

TrueType is a vectorized computer font format introduced in 1991 by Apple. TrueType fonts without hinting simply scale the glyphs' outlines to the actual size of the glyphs on the screen, and turn on the pixels which fall inside of the outlines. TrueType fonts with hinting can adjust the shapes of the glyphs' outlines to make them align with the rendering devices. TrueType engines can render the hinted glyphs properly on screens with different resolutions and font sizes thanks to its bytecode programs. The bytecode program is used as hints to move the glyphs' control points to their optimal positions, adjusting the outlines of the glyphs according to the pixels per EM (ppem), which is related to the screen resolution and font size. The formula for ppem is shown below. Chapter 3 introduces more details about TrueType and how font engines work.

$$\text{ppem} = \text{point size} \times \text{dpi} / 72$$

Since the bytecode programs are one of the core components in TrueType fonts, efficient manipulations of TrueType fonts require the ablility to manipulate the TrueType bytecode. Current TrueType font manipulation techniques modify the fonts without touching their bytecodes. For example, the TrueType fonts used by a web page are sent to the client along with the HTML files, and the font subsetting techniques are always applied to the fonts to make them contain only the used glyphs in the webpages, to save the network cost and the device's memory. Current font subsetting techniques [18] can not optimize the bytecodes with respect to the subsetted fonts. This makes the subset of the fonts contain un-optimized bytecode, which detracts from the benefits of fonts subsetting.

Webpages account for 17% of all Internet traffic [21], and almost 70% of websites contains custom fonts which each website serves 95 KB of fonts on average [12]. We have

observed that, for some popular fonts, bytecode accounts for almost half of their size—46% for Microsoft Core TrueType font Arial and 45% of NotoSansTibetan-Bold. So, our tool can save the bandwidth significantly when a web server transmits millions of webpages everyday.

1.1 Approach

TrueType font bytecodes are written in a highly dynamic stack based language. The program manipulates a stack to handle the values when executing bytecode instructions, and the instructions get information from graphics state as well, where the values are hardware related. The dynamic features of TrueType bytecode make it hard to perform analysis and optimizations on it. Three address codes are widely used in compiler design as an intermediate representation, and they are easy to analyze and optimize. Compilers perform optimizations on three address codes before translating them to lower level languages or assembly.

Previous work reported in Man's master's thesis [15] has proposed a three address code, COI, and implemented a prototype abstract executor which transforms the bytecode to COI. The previous project enabled the analysis and optimization of TrueType bytecode. However, it did not carry out any analysis and optimization, nor did it implement the inverse conversion from COI to TrueType bytecode.

In this thesis, we propose improvements to COI and the abstract executor, and implemented some basic COI optimizations. We also completed the cycle by translating COI back to TrueType bytecode, and tested that our optimized fonts do not change the output bitmap compared to the original fonts at certain resolutions.

1.2 Contributions of Thesis

This thesis is built on previous work which designed COI and implemented an early prototype. However, it added a number of novel research contributions in making a font analysis tool that actually works, in addition to adding engineering efforts. For example, previous work did not handle jump instructions and uncertain callees, which can be found in a many widely used fonts, such as the Microsoft Core TrueType font Arial. The research contributions of this work enabled the analysis and optimization of such fonts.

We also implemented some basic compiler optimizations on COI such as tree shaking, dead code elimination and erasing code without effects [4]. Then we transformed COI back to bytecode. The optimized TrueType fonts generate the same bitmaps as their original fonts at tested resolutions.

We also designed a TrueType bytecode debugger tool, which can monitor the state changes step by step.

We re-implemented the abstract executor, greatly increasing its efficiency. Even though we did more computations and recorded more information, our new executor

1. runs more than 100 times faster;
2. consumes less than 1/50 as much memory as the previous implementation.

Chapter 2

Related Work

The work of this thesis is based on or inspired by some related works.

2.1 Symbolic Execution and Static Analysis

Symbolic execution is a widely used static analysis technique. In symbolic execution, the variables are represented by abstract symbolic expressions. Normal execution, where all the data used are concrete, can be regarded as a special case of symbolic execution [13]. Recently, researchers have proposed concolic execution which combines symbolic execution and concrete execution to overcome limitations of symbolic execution [16]. Csallner and Tillmann determined likely program invariants by combining concrete execution with a simultaneous symbolic execution [9]. Bush and Pincus proposed to detect dynamic errors using symbolic execution by tracing the execution path [8]. Saxena and Poosankam introduced symbolic variables for loop iterations, and linked these symbolic variables to an input grammar [19].

Symbolic execution can be used to aid the analysis and optimization of programs. Demange, Jensen and Pichardie demonstrated a transformation from Java bytecode to a stackless intermediate representation and proved its semantic correctness [10]. Ma and Phang proposed shortest distance symbolic execution (SDSE) and call-chain-backward symbolic execution (CCBSE) strategies to find a realizable path to a given program point [14]. Pasareanu, Visser and Bushnell proposed a program analysis tool, Symbolic PathFinder (SPF) where the variables are represented by symbolic expressions to generate test cases and to analyze Java bytecode [5]. Shannon and Hajra proposed to symbolically execute

string class, using forward symbolic execution, which can be used to analyze SQL database queries [20].

Similarly, we use abstract symbols to represent the data, and develop abstract semantics to compute over the abstract data. Our abstract executor performs symbolic execution on TrueType bytecode programs.

Our ultimate goal in this work is to enable the analysis and optimization of TrueType bytecode programs, which are written in a highly dynamic stack based bytecode language. The TrueType font engine manipulates a stack to handle the data used by the program, and some values in TrueType bytecode programs are associated with hardware specifications, such as screen resolutions, which are not available at compile time.

2.2 Symbolic Range Propagation

Blume and Eigenmann proposed methods to compare abstract symbolic expressions. They have enabled arbitrary abstract expression comparison during symbolic execution using range propagation and range comparison [7]. Their work enables compilers to detect potential optimizations such as zero-trip loops and dead code. It is also used to approximate the program complexity, estimating the loop trip-count. Verbrugge, Co and Hendren presented a generalized constant propagation to statistically estimate the ranges of variables within C programs [23].

We are inspired by their work, and introduced a range propagation and comparison system in our abstract executor. TrueType bytecode programs decide to enter a branch with respect to the specifications of the rendering hardware, and these information is unknown at compile time. As we know that some branches of the bytecode programs are not executed when rendering the fonts on devices with high screen resolutions. Range propagation and comparison help us locate these dead blocks in TrueType bytecodes which are not reachable when using the fonts on modern devices. (See 5.4.2 for details).

2.3 Goto Elimination

Similar to assembly languages, TrueType bytecode programs also contain unstructured jump instructions. There are both conditional and unconditional jump instructions in the bytecode which may direct the instruction counter to any point of the program theoretically.

The jump targets in TrueType bytecode programs are unknown at compile time. Jump instructions can jump across other control-flow structures (i.e jump instructions and if-then-else blocks), forming complicated logic. So, we made efforts to eliminate the jump or goto statements in our COI. Peterson, Kasami and Tokura demonstrated an algorithm to make arbitrary flowgraph well-formed using node-splitting, and proved the equivalence [17]. Erosa and Hendren proposed an algorithm to eliminate goto statements in C programs, which directly works on high-level abstract syntax tree. Their algorithm applies a sequence of goto-movement transformations followed by a sequence of goto-elimination transformations [11].

Similarly, we moved the instruction counter following goto instructions and then modified the initial control-flow graph of the programs to eliminate jump instructions.

2.4 Three Address Code

Compilers translate the abstract syntax tree to intermediate representations by the end of the semantic analysis phase. There are many intermediate representations, including directed acyclic graphs (DAG), stack-based bytecode, and three address code.

Three address code is widely used for intermediate representations since it enables machine independent optimizations, such as dead block elimination and code motion. It is possible for compilers to generate target code without the help of intermediate representations, but the compilers will miss potential of optimizations [4].

The Soot framework converts Java bytecode to three intermediate representations, and makes it simple to manipulate Java bytecode [22]. This work was initially inspired by Soot. Analogously, we converted TrueType bytecode to COI, a stack-less three address representation, which enables basic optimizations widely used by compilers, and thus the optimization of dynamic TrueType bytecode.

Potential future optimization techniques include inverse-inlining, where code blocks with the same pattern are factored into a compiler-created function, and leaf function optimization, where non-leaf functions are translated into leaf functions. In future work, we will explore more optimization schemes which will work better on TrueType bytecode.

2.5 Fonttools and FreeType 2

Fonttools [1] is a TrueType bytecode manipulation tool. It not only performs merging and subsetting of TrueType fonts, but also converts between the TrueType font format and an XML-based format. We used Fonttools to manipulate our input and output data. We first convert .ttf files to the XML-based format, as our input data, and convert our optimized XML-based format file back to .ttf file using Fonttools.

FreeType 2 [2] is an efficient, highly customized and portable font engine. It not only supports accessing the contents of TrueType font, but also supports other popular font formats such as CCF, OpenType and so on. Freetype 2 does not manipulate text layout. Instead, it provides a low level and easy to use interface to access font files, which fulfills our needs perfectly.

FreeType 2 generates an array containing the bitmap for each glyph in a TrueType font, at specific screen resolutions. We use FreeType 2 to verify our output, by comparing the bitmaps of the original fonts to our optimized fonts with tested resolutions.

2.6 COI: A First Step Towards TrueType Bytecode Analysis

Man proposed and implemented a prototype abstract executor for TrueType bytecode [15], which converted the TrueType bytecode programs to a three address code, COI. Her work is the foundation of this thesis. It also proposed some ideas for bytecode optimizations. A lot of ideas in the previous work appear in our work.

We proposed many new ideas and introduced new features to COI in this work, to make the abstract executor stronger, as explained in chapter 5. We also performed basic optimizations on COI and closed the cycle, by implementing inverse conversion from COI to TrueType bytecode.

Chapter 3

Background

3.1 TrueType Font and Font Engines

TrueType font is a vectorized computer font format developed by Apple and Microsoft, and it has became one of the most common font format on the most popular operating systems, such as Mac OS, Ubuntu and Microsoft Windows. The glyphs in a TrueType font are digitized as a specific format to instruct a software, which is called the TrueType font engine, to draw the glyphs as needed.

When the font glyphs are requested, the font engine creates a bitmap for each glyph, and the bitmaps can be shown on the screen.

TrueType font glyphs can be displayed on screens with different resolutions precisely, because font engines create the optimal bitmaps for the glyphs whenever the font is initially imported to an application or whenever the user changes the size of the font.

3.2 TrueType Bytecodes

To create optimal bitmaps for different resolutions and font sizes, font engines execute the bytecode programs in the TrueType font files. By executing the bytecode programs, the font engine adjusts the outlines of the glyphs in the font, hence makes them in their optimal positions on the rendering screen.

TrueType font engine manipulates a stack to handle data used by the program, and the bytecode instructions also access and modify the values in graphics state, where the

values are associated with rendering devices. The dynamic features of bytecodes is the motivation of our work. It is almost impossible to perform analysis and optimizations directly on TrueType bytecode. So, we enabled analysis and optimizations of bytecode programs by converting them to a three address code, COI, and translated the optimized COI back to TrueType bytecode.

3.3 Components of TrueType Font

For the purpose of understanding our project, we introduce some important components of TrueType font. A TrueType font contains:

- a control value table (cvt),
- a font program,
- a control value program,
- a set of glyphs.

The control value table contains the values associated with font features and is useful when rendering the font on an application. The cvt has default values even when the concrete environment is not given. However, control value table is optional, and some fonts do not have cvt.

The font program and the control value program are TrueType bytecode programs. Whenever an application requests that a font be rendered, a font engine executes the font programs first. The font program defines a set of functions with integer identifiers, which can be used by other programs through CALL[] or LOOPCALL[] bytecode instructions. This is similar to defining functions in other programming languages. Font creators have presumably factored the code which can be re-used into functions.

The font engine executes the control value program whenever the font is initially imported or the user changes the font size. It resets the default values in the cvt and graphics state of the font for the specific screen and font size.

The font also contains a set of glyphs. Each glyph consists of a sequence of control points and a bytecode program, which is called a glyph program. The font engine executes the glyph program of each glyph to adjust the positions and the shapes of the outlines to make them optimal to the rendering screen.

The bytecode programs are stack-based. The engine uses a stack to manipulate the data when executing the bytecode programs, and some bytecode instructions use hardware related data which are unknown statically.

There are more components in a TrueType font, which are irrelevant for our work. For example, head table contains the information about the font such as the font version number and modification date; kern table contains the inter-character spacing for glyphs. The apple TrueType reference manual provides detailed TrueType components [6]. The knowledge of the components above is sufficient to understand our work.

3.4 Glyph Representation in TrueType

A TrueType font contains a set of glyphs. Each glyph consists of two core parts, the master outline of the glyph, and a glyph program.

The master outline of a glyph is defined by a set of contours, and it is device independent. Contours are closed curves, which are building blocks of glyph shapes. They are defined by a spline of continuous quadratic Bezier curves, whose control points are the point sequence defined in the glyph.

Theoretically, a second order Bezier curve is represented by 3 control points, one of which is off curve, and the other two are at the ends of the curve. Since the curves are 1st order continuous, one or two of the control points can be implied by their adjacent curves, hence omitted. The points are order sensitive, since they are not only used to define the track of the contour, but are also used to define the orientations.

The font engine executes each glyph program to move the control points of the outline to their optimal positions, before the font needs to be rendered (see section 3.5).

3.5 How Font Engines Render a Glyph

As we introduced in previous sections, font engines render fonts whenever applications request that fonts are displayed. A font engine renders fonts as follows.

The font engine first executes the font program. It identifies FDEF[] and ENDF[] instructions as delimiters. Every bytecode instruction between a pair of FDEF[] and ENDF[] is in the body of a function. The engine puts the set of program functions into a function table for future use. Note that the engine only executes the font program once, when the

font is initially accessed. The execution of the font program is similar to declaring and defining functions in Java.

Then the font engine executes the control value program. The control value program is executed whenever the application initially requests the font or asks for a new size to be rendered. This bytecode program resets the default values of the control value table and the graphics state for the specific devices and font sizes.

Finally, the font engine draws the bitmap for each requested glyph in the font. Because the glyph program may modify the values in the control value table and the graphics state, the font engine always resets the values in the cvt and the graphics state to their defaults, before drawing the next glyph.

The procedure of rendering a glyph is as follows. Figures 3.1, 3.2 and 3.3 show the procedure [6].

1. The engine scales the master outline of a glyph to its desired size, changing the coordinates of the points from device independent to device dependent. The font engine stretches the master outline by multiplying the coordinates by a scalar, which can be calculated with the formula below. This transformation is device dependent, as seen from the formula. The outline of the glyph is called a scaled outline (Figure 3.1) after this transformation.

$$\text{scalar} = \text{point size} \times \text{resolution} / (72 \times \text{points per inch} \times \text{units per em})$$

2. Then the engine executes the glyph program to adjust the scaled outline to its optimal position to the device and the font size. The key effect of executing the glyph function is to move the scaled outline's control points. This procedure is called grid-fitting. The purpose of grid-fitting is to eliminate the effect of chance relationships to the grid and to control key dimensions. The adjusted outline is then called the grid-fitted outline. In this case (Figure 3.2), the font engine makes both vertical lines of the Capital H wider a little bit.

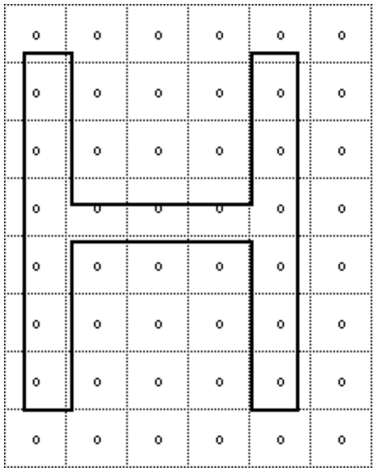

Figure 3.1: The engine transforms the master outline to device dependent outline by multiplying the coordinates to a scalar.

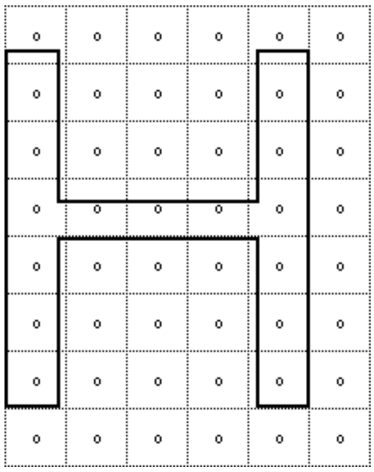

Figure 3.2: The engine adjusts the scaled outline by executing the bytecode programs.

3. A part of the font engine, which is called the scan converter, turns on the pixels which fall within or on the grid-fitted outline by applying some rules to create the bitmap (Figure 3.3) of the glyph with respect to the specific screen and font size.

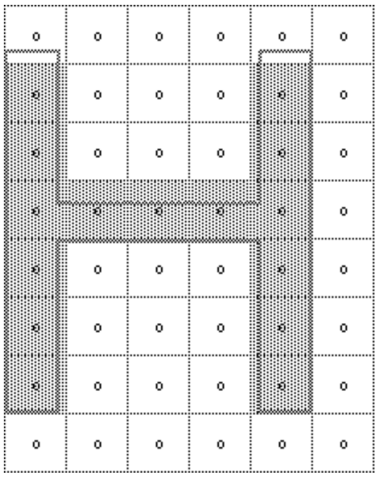

Figure 3.3: The engine (scan converter) turns on the pixels fall within or on the grid-fitted outlines to generate the bitmap.

A font engine does subtler work to render a font, but for our purpose, it is sufficient to know these steps. The Apple TrueType Reference Manual [6] provides more details.

Chapter 4

Motivation

In previous chapters, we have introduced TrueType fonts and the their bytecode programs. We also described briefly how font engines draw bitmaps for glyphs whenever an application requests that a font displayed. In this chapter, we discuss our motivation of TrueType bytecode analysis and optimization, and why we need a three address code representation to analyze and optimize the bytecodes.

4.1 TrueType Bytecode Analysis and Optimization

Many TrueType bytecodes are generated by auto hinting software [3]. Also, existing font manipulation techniques [18] modify the TrueType fonts without touching the bytecodes, which create un-optimized TrueType fonts. Bytecode programs decide to enter different branches by comparing the ppem with thresholds. Pixels per EM is related to the dpi of the rendering devices and the font sizes, and it is derived using the formula below. On some modern devices, the values of ppem are always greater than thresholds, and this causes some branches to be unreachable.

$$\text{ppem} = \text{point size} \times \text{dpi} / 72$$

So, analysis and optimization on TrueType fonts may generate more efficient TrueType bytecode for modern devices, and makes font subsetting techniques more efficient as well.

4.2 The Difficulty of TrueType Bytecode Analysis

The font engine uses a stack to manipulate the values, and it interacts with graphics state where the values are associated with rendering devices. It is almost impossible to know some values before a concrete runtime environment is given. The existence of some instructions in TrueType bytecodes makes their analysis even harder:

4.2.1 Hardware Specification Related Instructions

Some instructions push hardware or font engine related values on the stack. For example, GETINFO[] pushes the data about the version of the TrueType font engine as well as the characteristics of the current glyph on the stack; MPPEM[] measures the current ppem, which is determined by the current rendering device and the font size.

The outputs of these instructions are abstract without a concrete runtime environment, and these abstract values are propagated through the symbolic execution. The existence of these instructions impedes the static analysis of the TrueType bytecode.

4.2.2 Control Flow Instructions

Control Flow instructions in TrueType bytecode include IF[]/ELSE[]/EIF[], CALL[]/LOOPCALL[] and JMPR[]/JROT[]/JROF[] instructions. These instructions determine the next instructions to be executed.

1. IF[]/ELSE[]/EIF[] instructions form an if-then-else block. The conditions are popped off the stack, and are statically unknown.
2. CALL[]/LOOPCALL[] instructions make the program execute a pre-defined function. The target function identifiers are unknown before runtime.
3. JMPR[]/JROT[]/JROF[] instructions moves the instruction counter to a new location specified by the offset which is popped off the stack. This may change the structure of the program in an unpredictable way.

In real life, if-then-else blocks can be nested and jump instructions entangle with if-then-else blocks, forming more complicated control flow structures.

4.3 Enabling Analysis of TrueType Font Bytecode

As we discussed in the previous section, analysis directly on TrueType bytecode is hard. We need another form of TrueType font program representation, on which the optimization techniques can easily apply.

Previous work [15] designed COI, a three address code, and implemented a prototype abstract executor to convert bytecode to COI. COI enables analysis and optimization on TrueType font bytecode programs. In this work, we improved the design of COI and the abstract executor to make it more practical, and implemented basic optimizations on COI. We also round tripped it by implementing reverse conversion.

Chapter 5

Methods

Figure 5.1 illustrates the overall organization of our research tool. We have improved on previous work by adding COI optimization and round-tripping from COI back to TrueType font bytecode.

5.1 Overall Structure of Project

Our tool performs the following steps on a TrueType font.

- Extracts the bytecode of the control value program, font program and glyph programs.
- Executes the font program, building the global function table.
- Performs symbolic execution on the control value program and then on each glyph program separately.
- Generates the COI representation at the same time of symbolic execution. Our tool records the graphics state and control value table after executing the control value program and restores the environment to that after executing the control value program before executing the next glyph program.
- Optimizes COI programs.
- Compiles the optimized COI programs back to TrueType bytecodes.

- Generates an optimized font, and validates the optimized font by comparing the bitmaps of the optimized fonts to the initial fonts using external FreeType 2 library.

5.2 Improvements

Our tool builds on the previous work and we improve it in a number of ways. These improvements not only consist of research contributions, but also result in a tool, which is for the first time, usable by end users for font manipulation. Our improvements include:

- Completed the symbolic execution of instructions; modelled TrueType bytecode operations in details; enhanced our symbolic execution engine.
- Implemented variable range propagation and comparison in COI.
- Approximated uncertain call destination in TrueType bytecode.
- Handled unstructured jump instructions by modifying the bytecode program structure.
- Fixed bugs in the previous implementation, such as if-then-else block handling, and loopcall handling.
- Implemented basic compiler optimizations on COI.
- Implemented round tripping from three address code back to TrueType bytecode.
- Implemented testing of our bytecode transformations by leveraging external FreeType2 library.
- Implemented a GUI debugger to aid users understand the effects of TrueType bytecode.
- Increased the speed by over 100 times and decreased the memory usage to less than 1/50 of the earlier prototype.

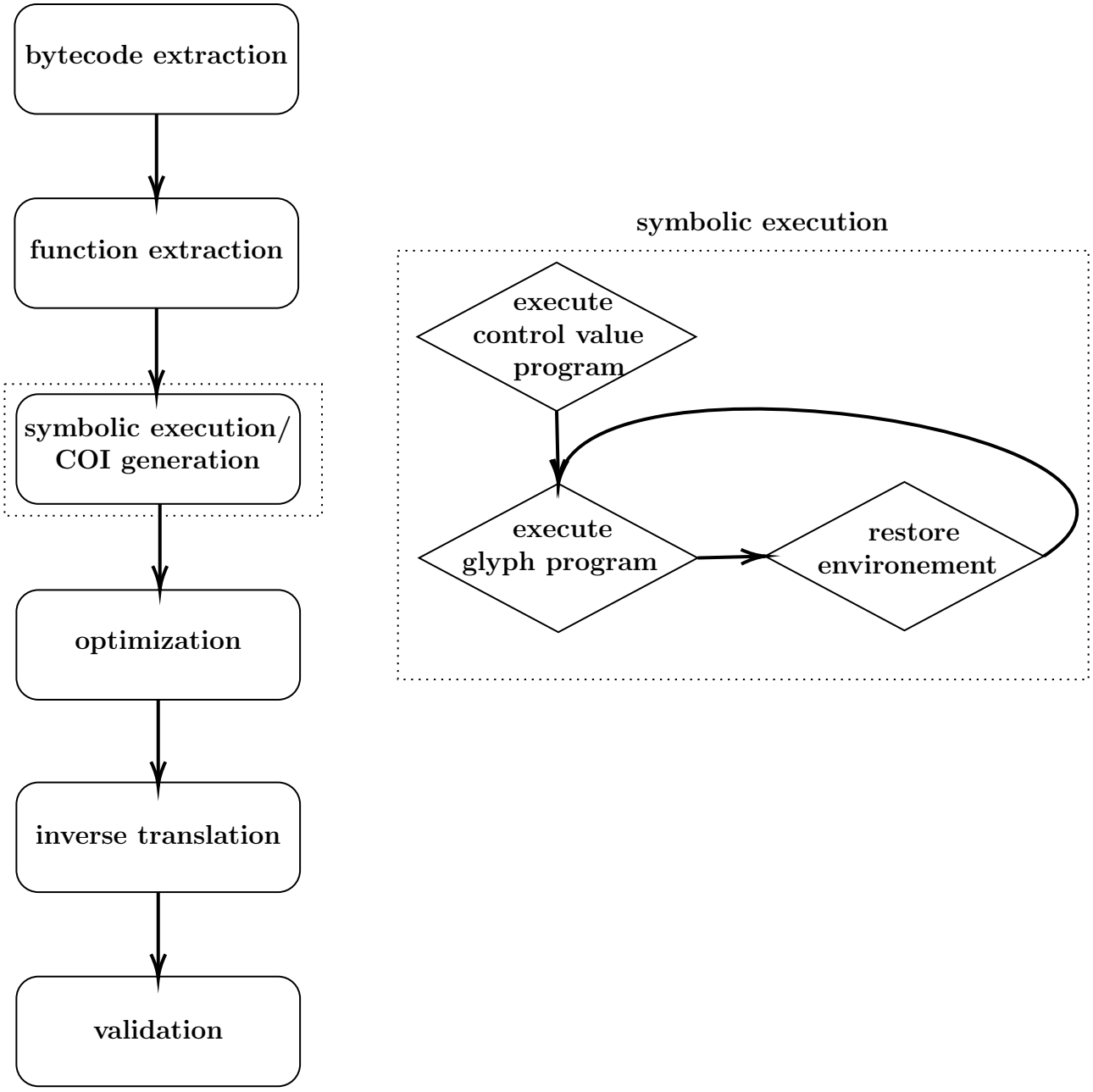

Figure 5.1: Overall structure of the project

5.3 Symbolic Execution and COI Generation

We introduce our implementations of symbolic execution and COI generation in this section. Section 5.3.1 to section 5.3.5 introduce the ideas and methods we continued to use from the previous work briefly, and we have revised the explanations and improved the implementations. Section 5.3.6 to section 5.3.12 discuss our novel inventions.

5.3.1 CFG Construction

The first stage of our tool is to read the bytecode programs from an XML based format font generated by the external FontTools [1]. We then construct the control flow graphs for the functions, the control value program and all glyph programs separately. The CFG construction for non-control-flow instructions is straight forward. We make estimation on the successors of the control-flow instructions, such as IF[]/ELSE[]/EIF[] and JROT[]/JROF[]/JMPR[].

We set the successors of an IF[] instruction to be its following instruction, the ELSE[] instruction (if applicable) and the EIF[] instruction in the same level. We assume the successor of other control-flow instructions to be their next instruction at this stage, and we will modify the CFG at symbolic execution stage.

CALL[] and LOOPCALL[] are also control flow instructions, but we treat them as non-control-flow instructions at this stage.

5.3.2 Abstract Executor

Traditionally, a TrueType font engine concretely executes the bytecode, so it has explicit inputs and outputs and produces concrete bitmaps for each glyph.

Instead of executing the TrueType bytecode with a concrete font engine, we execute the bytecode programs with our abstract executor.

The primary difference between our abstract executor and a font engine is that the values in the program stack and in the components of graphics state, such as the control value table, storage area could be uncertain. So, the abstract executor does not have a complete runtime context.

Decision making instructions, such as IF[] instruction, direct the program to different branches. When a font engine concretely executes a bytecode program, the program can

only enter one of the branches. Since we don't have a concrete runtime context and the values on the stack could be uncertain, we symbolically execute all possible branches.

Non-control-flow instructions are simply executed in sequence, but control-flow instructions require special treatment. Previous work handled control-flow instructions as follows.

- Function Call

  A function call instruction sends the flow of control to the first instruction of a predefined function and stores the instruction after the call instruction on a call stack. The instruction counter resumes executing the caller after it finishes executing the callee function.

  Since function calls can be nested and recursive, the abstract executor uses a stack to store call records. When a function is called, the abstract executor pushes a new instance of call record on the stack, and it pops off the top call record at return. This behaves like function call management at runtime in other programming languages.

  The LOOPCALL[] instruction is specific to TrueType fonts. It calls a function multiple times consecutively. The font engine reads the iteration count from the graphics state at runtime.

  Because the stack elements could be uncertain in symbolic execution, the call targets are also uncertain sometimes. This is similar to dynamic dispatch in Object Oriented languages or function pointers in C. Section 5.3.10 explains how our abstract executor supports uncertain callees in our new implementation.

- If-then-else Block

  We assume that the program can enter either branch at runtime. So, we execute the then-branch first and restore the environments to the status before entering else-branch. Then we execute the instructions in the else-branch. We finally merge the environments of the two branches at the end (see section 5.3.9 for details), which is a standard abstract execution technique.

5.3.3 Stack Depth

As implemented in previous work, we track the stack depth when performing symbolic execution. Since we generate three address code at the same time as symbolic execution, we assign identifiers associated with the stack depth to variables.

5.3.4 Uncertain Variables

The value of a COI variable could be either concrete or uncertain. An Uncertain variable is introduced in two possible ways.

1. **By executing an instruction which pushes an uncertain hardware related value to the stack.** For example, MPPEM[] pushes the ppem to the program stack; MPS[] instruction pushes the point size on the stack.
2. **By environment merging.** If the values of the same variable in different branches are not the same or the value of the variable is uncertain in either branch, the variable in the merged environment becomes uncertain.

We previously asserted that the stack depths are the same at the end of the two branches of an if-then-else block. We generalized this assertion in this work to that the stack effects are the same in arbitrary number of branches of the program. This could happen when a callee is uncertain. We handle uncertain callees by executing all possible functions, where the number of possible functions can be arbitrary.

5.3.5 COI Translation

In this section, we briefly introduce how the previous implementation converted bytecodes to COI. We still use these ideas in our work to represent abstract data types, to convert arithmetic instructions, to express the parameters of instructions, to translate CALL[]/LOOPCALL[] instructions and so on.

- Abstract Data Types

  An instruction may push an abstract graphics state value on the stack. We defined some abstract data types for these instructions. For example, MPPEM pushes a PPEM_X or a PPEM_Y to the stack, where PPEM_X refers to the ppem value with respect to horizontal coordinate, and PPEM_Y is respect to vertical coordinate.

- Assignment statement

  Figure 5.2 shows some examples of assignment statements. Instructions which push values on the stack generate assignment statements, such as PUSH[], DUP[], SWAP[],

1.	**PUSHB[9]**	**$var_1 := 9**
2.	**SRP0[ ]**	**$graphics_state[rp0] := $var_1**
3.	**MPPEM[ ]**	**$var_1 := PPEM_X**

Figure 5.2: Assignment statements

and MPPEM[] instructions. The instructions which modify values in the indexed storage area, such as WS[] (i.e write storage) instruction, also generate assignment statements.

The left hand side of an assignment statement is either a variable name which is determined by the stack depth, or a graphics state entry. The right hand side can be one of a constant, uncertain data type, variable, graphics state entry or binary/unary operation depending on which instruction generates the assignment statement.

- Action statement

  Instructions which perform TrueType specific actions generate action statements. For example, point moving instructions, such as MDRP[], generate action statements.

- Binary and Unary Operations

  We treated arithmetic and logic operations as binary and unary operations. Binary operations include ADD[], SUB[], MUL[], DIV[], GTEQ[], LT[] and so on. Unary operations are NOT[] and NEG[]. We translated these instructions in a straight forward way, combined with an assignment statement.

```
1          1. ADD[ ]          $var_1 := $var_2 + $var_1
2          2. NEG[ ]          $var_1 := - $var_1
```

Listing 5.1: Binary and Unary Statements

Listing 5.1 shows examples of statements that represent binary operations and unary operations.

- Arguments

  The values consumed by an instruction go to the argument list of the corresponding COI statements.

  For example, SHZ[] shifts the zone using reference point, which is is popped off the stack. We translated SHZ[] as following. $var_1 is the variable of the top element on the stack, and it is in the argument list of the shz action statement.

```
1           shz($var_1)
```

- CALL[] and LOOPCALL[]

  A CALL[] instruction is translated to a special assignment statement. We put return values on the left hand side of the assignment statement. A function that returns void does not have the left hand side component. The right hand side of the assignment statement is a call expression followed by its call destination and the argument list. Section 5.3.7 explains how our new implementation finds the parameters and return values of functions.

  LOOPCALL[] is similar, but it also indicates the iteration count on the right hand side.

  Listing 5.2 shows examples.

```
1           1. $var_1,$var_2 := call 0 ($var_1)
2           2. loopcall 0 5 ()
```

Listing 5.2: CALL and LOOPCALL

5.3.6 Detailed Modeling of Point Moving and Graphics State Instructions

From this section, we emphasize our improvements and innovations.

One of our improvements is that we modelled point moving instructions (such as MDAP[], MDRP[]) and graphics state instructions (such as SRP0[], SDS[]) in more details. The previous implementation only took care of the stack effects of these instructions, which we found to be insufficient.

Some point moving instructions have side effects, such as setting zone pointers and setting reference points. A better modelling of these instructions helps us to locate instructions with no effects (see section 5.4.2). For example, if an instruction has a side effect of changing the reference point 0 to p, we don't need to execute a set reference point 0 (SRP0[]) instruction right after it which sets the reference point to p again.

Take the MDRP[] instruction as an example. The previous implementation modelled it simply by popping a point number p off the stack. In fact, MDRP[] moves point p so that the distance from its new position to the current position of reference point 0 is the same as the distance between the two points in the original uninstructed outline, and then adjusts it to be consistent with the Boolean settings [6]. So, it uses some graphics states, such as zp0 (zone pointer), zp1, rp0 (reference point), freedom vector, etc. We deep copy these states when abstract executing these instructions and store it in the data structure of the COI, and we execute their side effects as well in this work.

5.3.7 Parameters and Return Values of Functions

A TrueType CALL[] instruction generates a call statement in COI, where the left hand side indicates the return values and the variables in the parentheses on the right are the arguments consumed by the function.

```
1
2 PUSHB[0]          $var_3 := 0
3 CALL[ ]           $var_1, $var_2 := call $var_3 ( $var_2, $var_1 )
```

Listing 5.3: Function call with parameters and return values

In Listing 5.3, the CALL[] instruction calls function 0. Assume that we have 3 elements on the program stack before executing CALL[]. The top element is the call destination (i.e 0). The instructions in function 0 modify 2 values on the stack.

In the previous implementation, the parameters and return values of a function were determined by a pre-computed stack effect for the function. If the called function reduced the depth of the stack, the call statement considered the reduced variables as parameters and no values were returned; if the called function increased the depth of the stack, the increased variables were considered as return values of the function and the function did not take arguments.

However, we found that the previous approximation was incomplete. The arguments of a function should be all the values used in the function body, while the return values of a function should be the values changed by the function, which could be either new values or modified values. A function can have both parameters and return values. But in our previous implementation, a function can not have both parameters and return values. Thus we replaced it with a new function parameter and return value finding mechanism.

In our new implementation, the number of parameters is the depth to which the called function instructions penetrate the stack, and the number of return values is the difference between the number of parameters and the stack effect of the function. In this way, every value used by the instructions of the function is an argument, and every value modified or written by the function is a return value. Thus, our new implementation can handle functions with arbitrary return values and parameters.

We also need to assign variable names to arguments and return values. Let the number of parameters be p, and number of return values be r. The arguments of the function call are simply the top p elements of the stack before executing the CALL[] instruction, and the return values are the top r variables of the stack after return from the called function.

In Listing 5.4, assume that we have 3 values on the program stack before executing the PUSHB[0] instruction. Listing 5.5 shows the instructions of function 0 and the program stack before calling function 0 ($\$var\_3$ is the top element). The values in the stack are represented by COI variables.

```
1                PUSHB[0]
2                CALL[ ]
```

Listing 5.4: A function call example

```
1   function 0:          program stack upon
2                        entry to function 0:
3
4   ADD[ ]               $var_1
5   ADD[ ]               $var_2
6   ENDF[ ]              $var_3
```

Listing 5.5: Function 0 instructions and stack

Function 0 simply performs two additions. The first ADD[] pops two values off the stack, calculating their sum, and pushes the result back. The second ADD[] pops the result of the last addition operation and another value off the stack, and pushes the sum back to the stack again.

It is obvious that function 0 reads three values on the stack and pushes the sum of them to the stack.

$\$var\_1$ will be the only value on the stack after function 0 returns. The value of $\$var\_1$ is changed to be the sum of the top three values by the function. So, we represent the CALL[] instruction with the following statement.

```
1   $var_1 := call $var_4 ( $var_3, $var_2, $var_1 )
```

Notice that we interpreted the call destination in the COI statement as a variable. This was a concrete function id in the previous implementation. Our new implementation representation is more accurate since the callees are sometimes uncertain (see section 5.3.10 for details).

5.3.8 Range Propagation and Comparison

Inspired by Blume and Eigenmann [7], we designed a range system to track ranges for COI variables. Since the type of a concrete value in bytecode is one of integer, fixed point number, or a boolean value, the range of a variable can be represented with extended integer (i.e $(-\infty, +\infty)$).

Variable value ranges are helpful for solving the problem of uncertain callees (section 5.3.10), narrowing the range of ppem (section 5.4.3), and locating unreachable bytecode.

The ranges are propagated through arithmetic and logic operations. Instructions writing to indexed storage (such as WS[] and WCVTP[]) also store ranges of written values. These ranges can be extracted by indexed storage reading instructions (such as RS[] and RCVT[]).

The range of a variable is also merged when merging environment of branches, and the merged range is the smallest connected set which contains both ranges as shown in the formula below.

$$[a, b] \cup [c, d] = [\min(a, c), \max(b, d)]$$

For example, if a variable's range is $[-2, 2]$ after exiting a then branch, while it is $[-1, \infty)$ at the end of the else branch, then the merged range of the variable after this if-then-else block is $[-2, \infty)$.

The priority of our range system is soundness. We then try our best to narrow the range. So, it is our goal that the actual range of variables are subsets of ranges we get (i.e $R' \subseteq R$, where R is the actual range and R' is the range we compute).

The lower bound and upper bound are the same for concrete values. They are initially introduced by push instructions. For instance, PUSHB[1] generates a variable with range $[1, 1]$.

5.3.9 Environment Merging

Values that depend on program input are only known at runtime. We don't know the the values when performing symbolic execution. TrueType bytecode can propagate uncertain values in two ways.

The first way, which can initially introduce uncertain values, is through instructions which push hardware related values. For instance, the MPPEM[] instruction pushes the ppem of the current rendering context; and MPS[] measures the current point size and pushes it on the stack. It is in principle impossible to know the values that will be pushed by these instructions before executing the code concretely, although we can estimate them for certain target display classes.

The second way is by environment merging. For example, in the COI, there may be a variable $\$var$ defined in both branches of an if-then-else block, and $\$var$ still exists after the program leaves the block. We merge the values of $\$var$ in the two branches right after leaving the if-then-else block. If $\$var$ has different values in different branches or it is uncertain in either of the branches, then $\$var$ gets a merged range as described in Section 5.3.8.

The previous implementation only merged the stack (represented as COI variables). However, merging values of graphics state and indexed storage areas (i.e., control value table and storage area) is also important, because two branches of a program may have different effects on graphics state and indexed storage. These values may be retrieved for further computation. Failure to merge these values leads to inaccurate range propagation and COI generation.

Consider the following COI in Listing 5.6. The then-branch sets storage area 1 to 5, but the else-branch sets storage area 1 to 10. The program reads the value of storage area 1 after the if-then-else block.

```
1     if ($var_1) {
2         $var_1 := 1
3         $var_2 := 5
4         storage_area[$var_1] = $var_2
5     }else{
6         $var_1 := 1
7         $var_2 := 10
8         storage_area[$var_1] = $var_2
9     }
10
11    $var_1 := 1
12    $var_1 := storage_area[$var_1]  // $var_1 is uncertain,
13                                    //  and it has range of [5, 10]
```

Listing 5.6: Storage area merging example

In our new implementation, we not only merge the values in the graphics state and indexed storage, but also the ranges of variables, as stated in section 5.3.8.

We generalized environment merging to support merging arbitrary number of branches. A CALL[] instruction may have an uncertain destination, and all of these candidate functions are considered as branches. Arbitrary branch merging enables us to execute every possible function at a CALL[] instruction. This is one of the techniques we used to solve the uncertain callee problem (section 5.3.10).

We asserted that the stack effect of different branches are the same at environment merging. We found that there exist bytecode programs in some fonts violate this assertion. If it happens in the control value program, our tool exits immediately and the tool does not work on this font. If it happens in a glyph program, we stop executing the current glyph program, and try the next glyph, which means our tool does not work on this glyph. In this case, we can not remove the uncalled functions, since our symbolic execution is incomplete, and we do not know whether the skipped glyph programs call the unused functions we found.

5.3.10 Uncertain Callee

The CALL[] instruction is one of the control-flow instructions which influences the next instructions to be executed. The environment can be different when returning from different functions. The destination of a call instruction depends on the stack and thus can potentially be uncertain, which impedes the symbolic execution.

Uncertain callees sometimes occur in real life TrueType bytecode. Consider the code in Listing 5.7, which can be found in Microsoft Core TrueType font Arial. Assume that only one variable is on the stack before executing the first instruction. Assume $\$var\_1$ is uncertain and has range $(-\infty, +\infty)$.

```
1
2       // stack height = 1 ; $var_1 has range (−∞,+∞)
3
4       DUP[ ]              $var_2 := $var_1
5       ROUND[01]           $var_2 := ROUND[01]($var_2)
6       PUSHB[64]           $var_3 := 64
7       SUB[ ]              $var_2 := $var_3 - $var_2
8       PUSHB[0]            $var_3 := 0
9       MAX[ ]              $var_2 := max($var_3,$var_2)
10      DUP[ ]              $var_3 := $var_2
11      PUSHB[44, 192]      $var_4 := 44
12                          $var_5 := 192
13      ROLL[ ]             $var_6 := $var_5
14                          $var_5 := $var_3
15                          $var_3 := $var_4
16                          $var_4 := $var_6
17      MIN[ ]              $var_4 := min($var_5,$var_4)
18      PUSHW[4096]         $var_5 := 4096
19      DIV[ ]              $var_4 := $var_5 / $var_4
20      ADD[ ]              $var_3 := $var_4 + $var_3
21      CALL[ ]             call $var_3 ( )
```

Listing 5.7: TrueType bytecode from Arial with uncertain callee

The abstract executor pops the call destination off the stack which is unknown at static. The uncertainty of the program in Listing 5.7 was introduced initially by copying $\$var\_1$, which is the input of this function. Note this is much easier to see in COI on the right than bytecode: The call target is indicated as $\$var\_3$, and we can follow its value explicitly. In this program, $\$var\_3$ at the bottom is uncertain.

Next, we re-examine the above code from Listing 5.7, but now include the ranges for the COI variables. Listing 5.8 shows the ranges of the variables on the left hand side of the COI statements.

```
1
2       $var_2 := $var_1                                (−∞,+∞)
```

```
3        $var_2 := ROUND[01]($var_2)          (−∞, +∞)
4        $var_3 := 64                         [64, 64]
5        $var_2 := $var_3 - $var_2            (−∞, +∞)
6        $var_3 := 0                          [0, 0]
7        $var_2 := max($var_3,$var_2)         [0, +∞)
8        $var_3 := $var_2                     [0, +∞)
9        $var_4 := 44                         [44, 44]
10       $var_5 := 192                        [192, 192]
11       $var_6 := $var_5                     [192, 192]
12       $var_5 := $var_3                     [0, +∞)
13       $var_3 := $var_4                     [44, 44]
14       $var_4 := $var_6                     [192, 192]
15       $var_4 := min($var_5,$var_4)         [0, 192]
16       $var_5 := 4096                       [4096, 4096]
17       $var_4 := $var_5 / $var_4            [0, 3]
18       $var_3 := $var_4 + $var_3            [44, 47]
19       call $var_3 ( )
```

Listing 5.8: TrueType code with uncertain callee, now with the ranges of the assigned variables on the left

We know that the candidate call destination ranges from 44 to 47 by following the propagation of variable ranges. We deal with uncertain callees by symbolically executing every candidate function, which is analogous to standard treatments of dynamic dispatch.

We complete executing the whole bytecode program when we try a candidate function. If the symbolic execution fails at some point, we recover the program state back to that before the CALL[], and try the next candidate function. If the execution completes successfully, the candidate function is marked as called, and then we restore the environment and go to the next candidate.

All working candidate functions may be called under different concrete run time environments, and the called functions will be kept within the font program in the optimization stage. If two candidate functions have different stack effects, but both enable the rest of the program to execute, then we mark both of them as called functions. According to our experiments, the stack effects of all valid candidate functions are the same.

5.3.11 Vest

This thesis introduces the novel concept of a *vest*. A vest temporarily modifies the control-flow structure of a CFG for the duration of one compilation pass. We originally anticipated that control-flow targets would change more dynamically than what we found in practice. Although vests are novel, they are less useful than we had originally expected, and we do not use the full generality of vests in our transformations. We have implemented vests in our abstract executor and use this mechanism to handle some cases of jump instructions.

Every instruction in a COI CFG can possess a vest. A vest is a pointer to another instruction. Vests can be recursive (i.e. a vest can also have a vest). When our abstract executor executes an instruction with a vest, it moves the execution to the vest instruction, and takes off the vest if the vest instruction does not have another vest.

Consider the following example.

```
1          PUSHB[0] (line 3)
2          PUSHB[1]
3          PUSHB[2] (line 2)
```

In the code above, the 3 instructions are in program order (i.e. the 1st line's successor is the 2nd line, and the 2nd line's successor is the 3rd line). The instruction in the parentheses after each instruction is its vest. According to rules we defined, the executor executes line 2 and line 3 in order, but line 1 will not be executed since the execution is not directed back to line 1 in this program.

The abstract executor works as follows. The execution starts from line 1. The abstract executor's instruction counter moves to line 3 directly without executing line 1 since line 1 has a vest pointing to line 3. Then the execution directly moves to line 2, since line 3 has a vest pointing to line 2. Line 2 does not have a vest and it is executed. In the next step, the instruction counter moves to line 3 by following the natural control flow. Line 3 has no vest anymore (the vest is taken off at the first visit) and it is executed. The instruction counter finally moves forwards to the successor of line 3. Hence, line 1 has not been executed.

Figure 5.3 illustrates the situation. The diagram on the left hand side is the control flow of the original instructions. Right hand side is the flow diagram after we assign vests to the instructions and apply our rules. The nested square represents the vest of the instruction; the lines with arrows indicate the control flow; the numbers next to the lines indicate the order; and the descriptions tell what happens at each step.

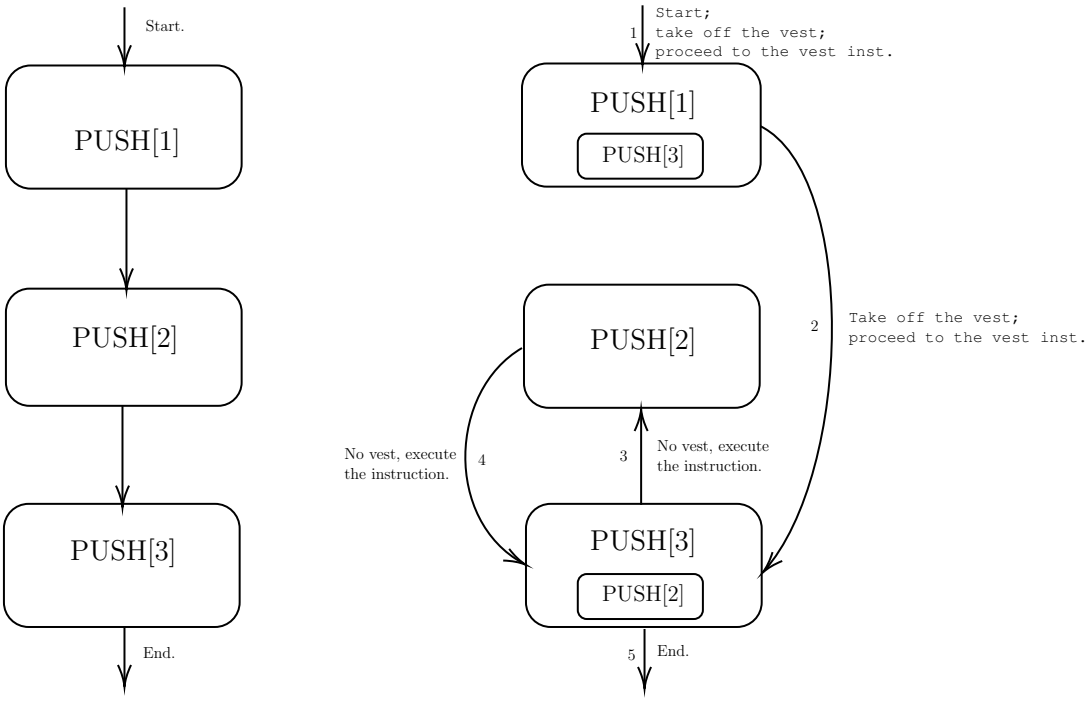

Figure 5.3: Flow diagram of instructions with vests

As we described in section 5.3.1, we estimated the CFG for control-flow instructions at CFG construction stage. We use vests to modify the control flow of the bytecode programs temporarily, enabling our abstract executor to handle the jump instructions (section 5.3.12 describes how we use vests). Our framework returns the CFG to its initial state after the pass finishes.

5.3.12 Unstructured Jumps

To our surprise, we found complicated unstructured jump instructions in real life fonts. Some jump instructions were entangled with other jump instructions as well as if-then-else blocks, forming more complicated logic.

In this work, we propose a novel method for processing jump instructions. We deal with jump instructions by converting them to loops, and by modifying the initial control flow. We ensure that there is no jump or goto statements in the COI.

In our benchmark suite, there exist implicit jump offsets (i.e. hard to know statically). We need symbolic execution to determine the offsets of jump instructions. We only deal with jump instructions with concrete offsets, and flag an error if a jump instruction has an uncertain offset. Fortunately, every offset we have seen so far is a concrete integer.

We also aim to statically determine the jump conditions for conditional jump instructions (i.e. JROT[], JROF[]) by popping a boolean variable off the program stack.

Finally, we examine whether a jump instruction jumps across if-then-else blocks. We have different methods for handling jump instructions with respect to three specifications;

1. **conditional or unconditional**;
2. **jump forwards or backwards**;
3. **jump across control flow structure or not**.

We demonstrate some cases with respect to the specifications above.

Unconditional / Forward / Not across

A forward unconditional jump instruction which does not cross any layer of if-then-else blocks is simple. We move the instruction counter to the target simply. Every instruction that the instruction counter skipped will not be executed and translated to COI. As a result, the optimized bytecode will not contain these skipped instructions. Figure 5.4 shows an example of this kind of jump instruction.

Unconditional / Backward / Across

A jump instruction with negative offset has the similar role of a while loop, so we interpret a backward unconditional jump as a while loop. A backward unconditional jump instruction must cross an IF[] to potentially terminate. We flag an error if we execute a backward unconditional jump instruction which does not cross control-flow structures.

The bytecode in Figure 5.5 calculates the sum of integers from 1 to 10 and writes the final result to a location in the storage area. We interpret it as a while loop. Note that the jump instructions are not translated explicitly, and we ensure there are no jump statements in COI.

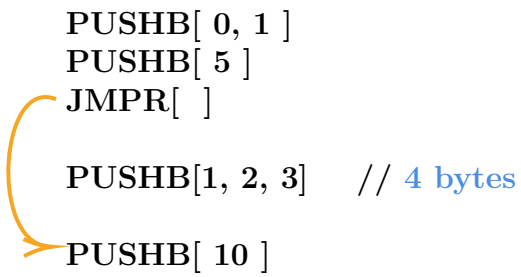

Figure 5.4: An example of unconditional forward jump not crossing control flow structures

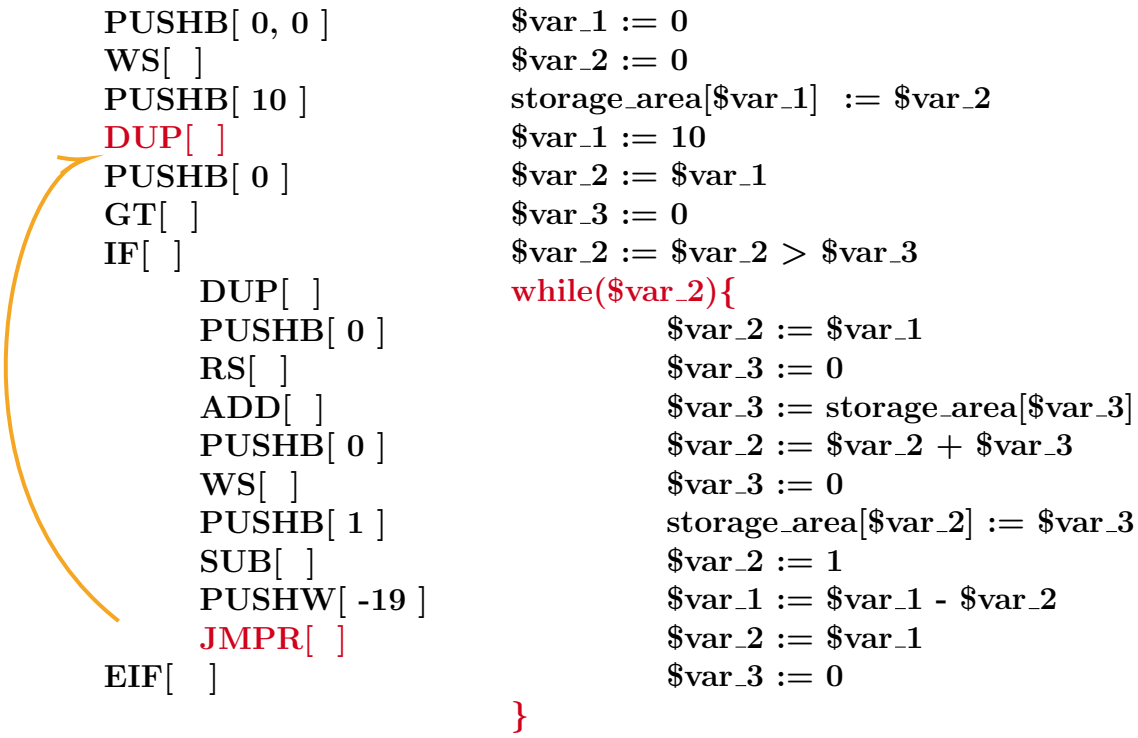

Figure 5.5: An example of unconditional backward jump crossing an if else block

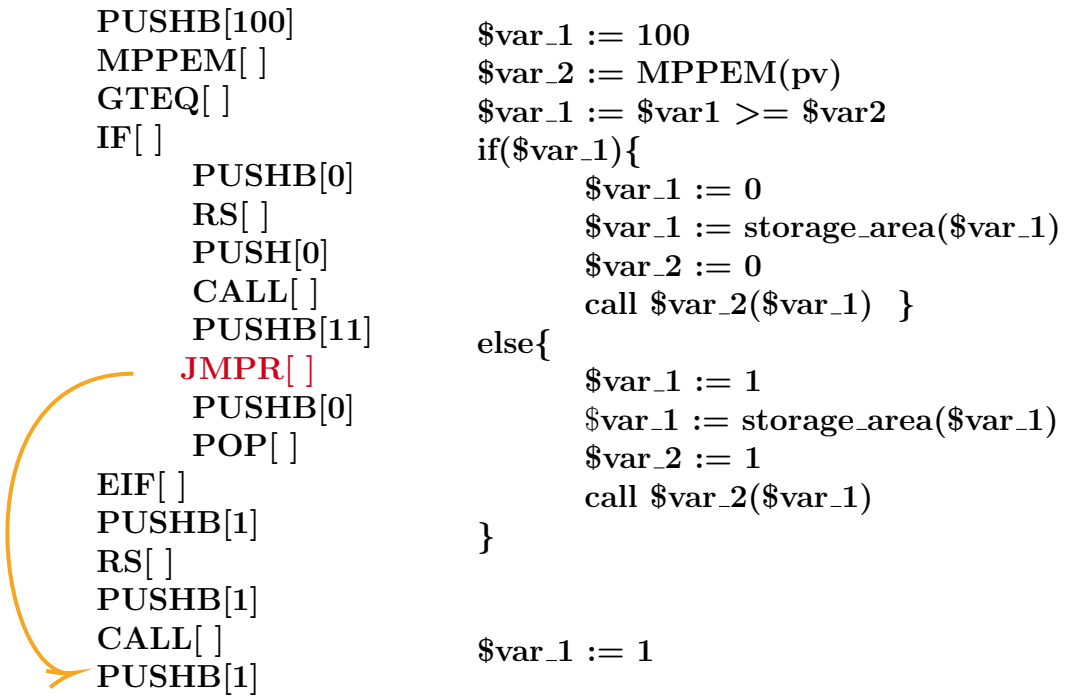

Figure 5.6: An example of unconditional forward jump over a single-branch block

Unconditional / Forward / Across

There are some jump structures in our benchmarks where the offsets are positive and they jump across if-then-else blocks. We will change the structure of initial if-then-else blocks accordingly. We have introduced our novel vests, in section 5.3.11. Now, we use vests to process these kinds of jump instructions.

Figures 5.6 and 5.7 show two examples where a forward unconditional jump instruction changes the structure of the initial if-then-else block.

In Figure 5.6, if the condition is True, the bytecode reads storage area 0, and calls function 0 with the value of storage area 0 as the parameter; otherwise, it reads storage area 1 and calls function 1 with value of storage area 1.

In Figure 5.6, the if-then-else block contains only a then-branch, and there is an unconditional jump instruction in the then-branch which jumps forwards across the if-then-else block. The instructions between the EIF[] and the target instruction will not be executed if the program enters the then-branch; they will be executed only if the condition fails. We interpret this as an if-then-else block with both then-branch and else-branch, where the then-branch consists of instructions between IF[] and JMPR[], and the else-branch consists of instructions between EIF[] and the target instruction.

We handle this case with the following algorithm:

1. create an ELSE[] instruction, and set its successor to the successor of the EIF[];
2. set the vest of target to the ELSE[], and set the vest of the ELSE[] to the EIF[];
3. change the successor of EIF[] to the target instruction.
4. adjust the second and the third successors of IF[] to the ELSE[] and EIF[];
5. continue executing the program at the target instruction.

In Figure 5.7, if the ppem of the current device is greater than or equal to 100, the program calls function 0 with value of storage area 0 and sets cvt[1] to 100; if the ppem of the current device is smaller than 100, the program calls function 1 with the value of storage area 1.

Since the if-then-else block contains both branches, we handle this case slightly differently:

1. set the vest of the target instructon to the ELSE[];
2. adjust the successor of the last instruction before the ELSE[] to point to the first instruction after the EIF[];
3. adjust the successor of EIF[] to the target instruction;
4. adjust the successor of the last instruction before the JMPR[] to the EIF[];
5. restore the state to that from before executing the IF[] instruction, and resume the execution at IF[].

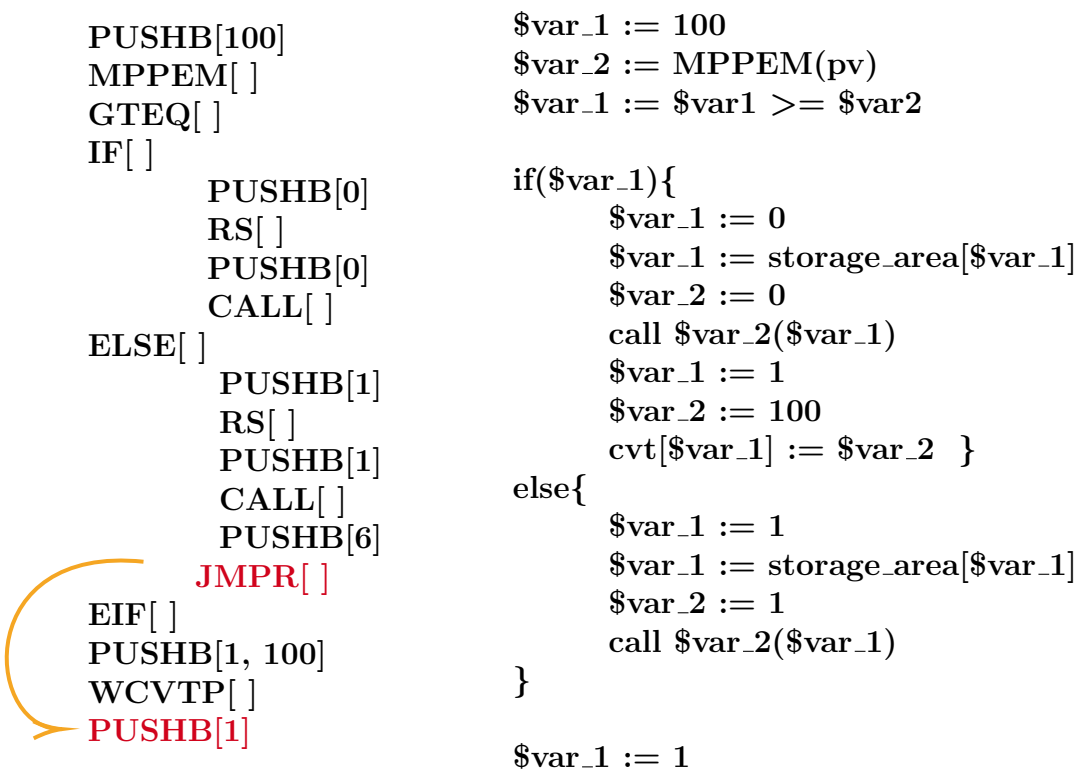

Figure 5.7: An example of unconditional forward jump over a double-branch if-then-else block

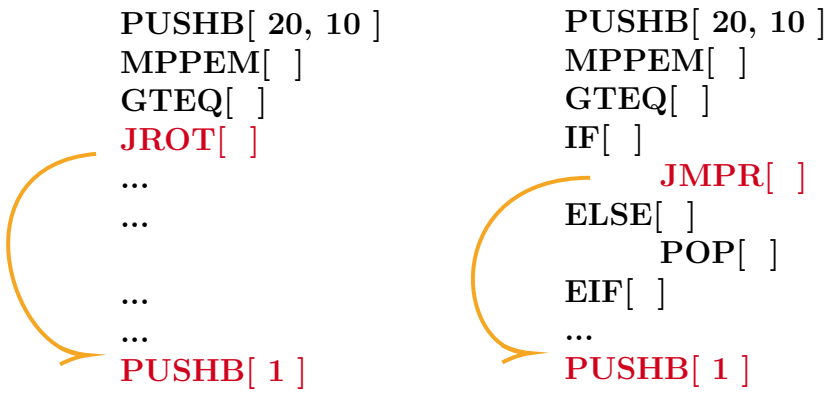

Figure 5.8: Jump relative on true. We adjust a JROT[] instruction to an if-then-else block where there is an unconditional jump in the then branch

Conditional Jump Instructions

To handle a conditional jump instruction (JROT[], JROF[]), we first transform it into an unconditional jump instruction wrapped within an if-then-else block, as shown in figures 5.8 and 5.9.

We then adjust the target offset properly to keep the target unchanged, and we pop the offset value in the branch where the jump will not take place. Then we resume the execution from the IF[] instruction and handle it as an normal unconditional jump instruction.

A backward conditional jump instruction not crossing any if-else block acts as a do-while loop. This is exactly the same as how we handle an unconditional jump instruction.

An example of compound control flow structure

Figure 5.10 demonstrates complicated real life TrueType bytecode with compound jumping logic from Microsoft Core TrueType font Arial. The COI in Listing 5.9 is the output of our abstract executor.

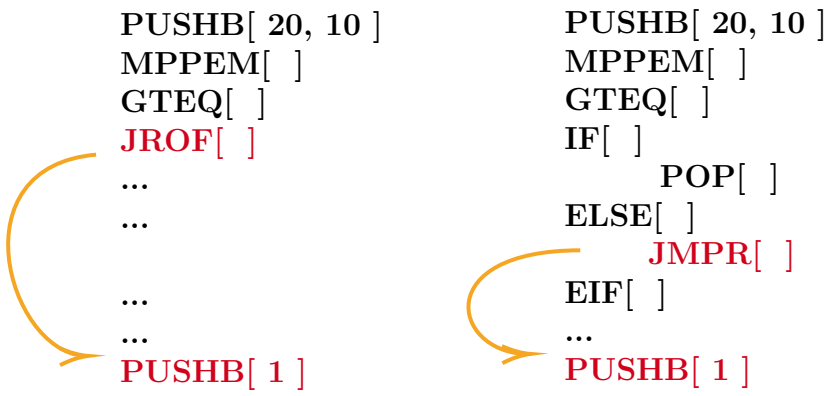

Figure 5.9: Jump relative on false. We adjust a JROF[] instruction to an if-then-else block where there is an unconditional jump in the else branch

Figure 5.10 contains both conditional and unconditional jump instructions, and all of of the jump instructions jump over control-flow structures. Some of the jump instructions even jump directly to the end of the function. Our abstract executor can handle all the compound control-flow structures like this in the tested fonts by modifying initial control flow. Our COI representation does not contain jump or goto statements.

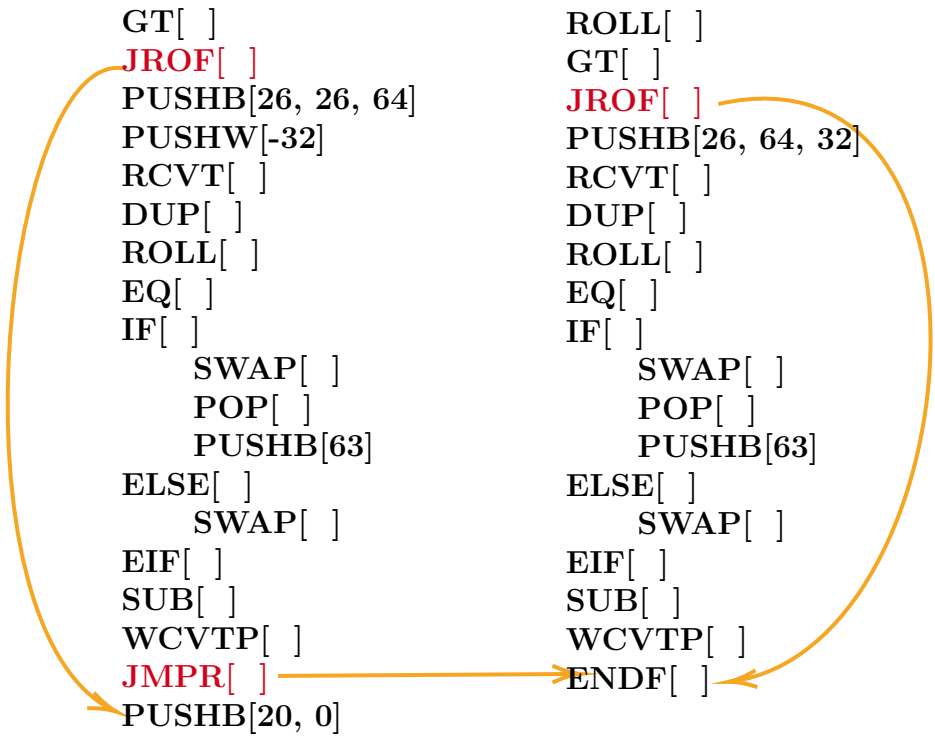

Figure 5.10: A complicated compound real life bytecode with jump instructions from Arial

```
1
2  $fpgm_24_30  :=  $fpgm_24_30  >  $fpgm_24_31
3  if( $fpgm_24_30 ){
4      $fpgm_24_28  :=  26
5      $fpgm_24_29  :=  26
6      $fpgm_24_30  :=  64
7      $fpgm_24_31  :=  -32
8      $fpgm_24_32  :=  26
9      $fpgm_24_32  :=  cvt[$fpgm_24_32]
10     $fpgm_24_33  :=  $fpgm_24_32
11     $fpgm_24_34  :=  $fpgm_24_33
12     $fpgm_24_33  :=  $fpgm_24_31
```

```
13      $fpgm_24_31  :=  $fpgm_24_32
14      $fpgm_24_32  :=  $fpgm_24_34
15      $fpgm_24_32  :=  $fpgm_24_32  =  $fpgm_24_33
16      if( $fpgm_24_32 ){
17          $fpgm_24_32  :=  $fpgm_24_31
18          $fpgm_24_31  :=  $fpgm_24_30
19          $fpgm_24_30  :=  $fpgm_24_32
20          $fpgm_24_31  :=  63
21      }else{
22          $fpgm_24_32  :=  $fpgm_24_31
23          $fpgm_24_31  :=  $fpgm_24_30
24          $fpgm_24_30  :=  $fpgm_24_32  }
25      $fpgm_24_30  :=  $fpgm_24_31  -  $fpgm_24_30
26      cvt[$fpgm_24_29] := $fpgm_24_30
27  }else{
28      $fpgm_24_29  :=  20
29      $fpgm_24_30  :=  0
30      $fpgm_24_31  :=  $fpgm_24_30
31      $fpgm_24_30  :=  $fpgm_24_28
32      $fpgm_24_28  :=  $fpgm_24_29
33      $fpgm_24_29  :=  $fpgm_24_31
34      $fpgm_24_29  :=  $fpgm_24_29  >  $fpgm_24_30
35      if( $fpgm_24_29 ){
36          $fpgm_24_28  :=  25
37          $fpgm_24_29  :=  64
38          $fpgm_24_30  :=  32
39          $fpgm_24_31  :=  25
40          $fpgm_24_31  :=  cvt[$fpgm_24_31]
41          $fpgm_24_32  :=  $fpgm_24_31
42          $fpgm_24_33  :=  $fpgm_24_32
43          $fpgm_24_32  :=  $fpgm_24_30
44          $fpgm_24_30  :=  $fpgm_24_31
45          $fpgm_24_31  :=  $fpgm_24_33
46          $fpgm_24_31  :=  $fpgm_24_31  =  $fpgm_24_32
47          if( $fpgm_24_31 ){
48              $fpgm_24_31  :=  $fpgm_24_30
49              $fpgm_24_30  :=  $fpgm_24_29
50              $fpgm_24_29  :=  $fpgm_24_31
51              $fpgm_24_30  :=  63
52          }else{
```

```
53              $fpgm_24_31  :=  $fpgm_24_30
54              $fpgm_24_30  :=  $fpgm_24_29
55              $fpgm_24_29  :=  $fpgm_24_31  }
56          $fpgm_24_29  :=  $fpgm_24_30  -  $fpgm_24_29
57          cvt[$fpgm_24_28] := $fpgm_24_29
58      }
59  }
```

Listing 5.9: A compilcated compound real life bytecode with jump instructions from Arial

5.4 Optimizations

One of our goals in developing a framework for TrueType bytecode analysis was to enable bytecode optimization. We implemented some basic optimizations widely used in compilers to optimize our COI, including removing uncalled functions (tree shaking), erasing code with no effects and dead block elimination.

5.4.1 Uncalled Functions

We continue to remove uncalled functions from the global function table as the previous implementation did. The previous abstract executor did not handle call instructions with uncertain callees and neither did it work on jump instructions. Thus, it skipped the glyphs containing these structures. Hence, it was not safe to remove the uncalled functions identified by the previous tool, since the symbolic execution was incomplete. We have improved the accuracy of uncalled function removal by handling the jump instructions and uncertain callee correctly, and it is safe to remove uncalled functions now. When generating bytecodes for function program, we only generate bytecodes for functions which are called or potentially called.

5.4.2 Instructions with No Effects

There exist some bytecodes which neither have any effect on program state nor move any points. Using mechanisms we introduced in the previous sections, we can easily locate these instructions and remove them from the bytecode program.

As discussed in section 5.3.6, we modelled the execution of instructions in more detail. If we know that an instruction changes the state, but the state has already been that the instruction will change it to, then the instruction has no effect.

As with any program optimization, this transformation must preserve program behavior. It then tries to identify as many no-effect instructions as possible. Of course, our tool finds a subset of the complete set of no-effect instructions.

```
1          ...                                    ...
2          PUSHB[ ]                               PUSHB[ ]
3          9                                      9
4          MDRP[10110]                            MDRP[10110]
5          ...                                    ...
6          PUSHB[ ]
7          9
8          SRP2[ ]
9          ...
```

Listing 5.10: An example of locating no-effect instructions

In Listing 5.10 above, SPR2[] is an instruction without effect; the reasons are as follows: MDRP[] sets reference point 2 to 9 as a side effect. Then the following SRP2[] instruction has no other effect besides popping an element off the program stack, since SRP2[] sets the reference point 2 to the top element of the stack, and the value of reference point 2 has been set to 9 by MDRP[] before executing SRP2[]. Thus, we can remove SRP2[] and the second PUSHB[], and the optimized code will be that shown on the right.

5.4.3 Dead block Elimination

Dead blocks are instructions which are not possible to be reached. This happens mostly at if-then-else blocks and at conditional jump structures.

In section 5.3.8, we introduced abstract range propagation. Variable range propagation enables us to restrict the ranges of unknown values.

The MPPEM[] instruction pushes the current value of ppem on the stack. The formula for calculating ppem is given again below. Ppem is associated with the point size and the screen resolution, and it is unknown to our abstract executor. Modern devices such as smart phones and laptops have high screen resolutions, and we can assume the value of ppem to be bigger than a device dependent threshold. Table 5.1 shows the screen

Device	Resolution	DPI	ppem
Apple iPhoneX	1125 × 2436	463	57.875
Google Pixel	1920 × 1080	441	55.125
Microsoft Surface	1366 × 768	148	18.500
Nokia Lumia 520	480 × 800	233	29.125
Samsung Galaxy S8	2960 × 1440	568	71.000
Amazon Fire	1024 × 600	170	21.250
Lenovo ideapad 1366	768 × 2436	118	14.750
Lenovo Yoga 2 Pro	3200 × 1800	276	34.500

Table 5.1: Resolutions and dpi of popular devices and ppem at point size 9

resolutions of selected popular modern devices as well as the values of ppem when point size is 9. If we assume the resolution of rendering devices are not smaller than 80 dots per inch (dpi), and the font sizes in webpages are not smaller than 9, we can restrict the lower bound of ppem to 10 according to the formula below.

$$\text{ppem} = \text{point size} \times \text{dpi} / 72$$

Many if-then-else blocks' conditions are derived, directly or indirectly, by comparing the ppem to some thresholds. According to our experiments, some of the threshold are smaller than the ppem values shown in table 5.1. Thus, one of the if-then-else branches will not be reached when the fonts are rendered on modern devices. It is therefore possible to specialize fonts by removing bytecode for rendering on devices that are no longer common today. We set the lower bound of ppem values to a concrete value to find and eliminate the unreachable blocks under the assumption.

```
1  PUSHB[ ]              $var_1 := 10
2  10
3  MPPEM[ ]              $var_2 := PPEM(pv)
4  GTEQ[ ]               $var_1 := $var_1 >= $var_2
5  IF[ ]                 if( $var_1 ){
6  ...                      ...
7  ELSE[ ]               }else{
8  ...                      ...
9  EIF[ ]                }
10 ...                  ...
```

Listing 5.11: A dead block example

We demonstrate an example in Listing 5.11. The bytecode compares ppem with 10. It enters the then-branch if the ppem is smaller than or equal to 10; it enters the else-branch otherwise. If we assume that the ppem on rendering devices are not smaller than 10, we can erase the instructions for the then-branch and also the IF[]/ELSE[]/EIF[] instructions.

5.5 Inverse Conversion and Validation

After we finish the optimizations on COI, we then convert the COI back to bytecode and replace the original TrueType bytecodes with the optimized bytecodes. Since COI is a three address code intermediate representation, we can perform the inverse conversion simply by pattern matching, which is similar to the code generation phase of a compiler.

We manage a stack of COI variables to mimic the program stack when generating the bytecode. When a new COI variable is assigned, we push an element on the stack; we pop elements when a statement consumes variables as parameters. Note that if a statement both assigns variables and consumes variables, we pop parameter variables first, and then push the new assigned variables. Finally we append the translated bytecodes for the corresponding COI statement to the current bytecode block.

Most non-control-flow bytecode instructions have a one-to-one mapping to COI statements. DUP[], MINDEX[], CINDEX[], SWAP[], ROLL[] are special, which can be indicated by surrounding statements as well as the top few elements of the variable stack.

```
1        $var_1 := 96                          PUSHB[96]
2        $var_2 := PPEM(pv)                    MPPEM[ ]
3        $var_1 := $var_1 >= $var_2            GTEQ[ ]
```

Listing 5.12: One-to-one mapping statements

In Listing 5.12, it is easy to see that line 1 corresponds to a PUSHB[] instruction, which pushes 96 on the stack. The second line corresponds to an MPPEM[] instruction, and the third line can be interpreted as a GTEQ[] instruction. Both statements in line 1 and line 2 push a variable to the COI variable stack, and the statement in line 3 consumes 2 variables on the stack and pushes a variable to the variable stack. When we translate line 1 and line 2, we push $\$var\_1$ and $\$var\_2$ on the stack; we then pop 2 variables consumed by the third statement and push the new $\$var\_1$ variable on the stack.

```
1  $fpgm_24_32 := $fpgm_24_31        $fpgm_24_32 := $fpgm_24_31
2  $fpgm_24_31 := $fpgm_24_30        $fpgm_24_33 := 10
3  $fpgm_24_30 := $fpgm_24_32
4  $fpgm_24_32 := 10
```

Listing 5.13: Translation of COI copy and assignment statements

When we see an assignment statement which copies a variable from another variable, we translate it to one of a DUP[], MINDEX[], CINDEX[], SWAP[], or ROLL[] instruction. We look ahead to the following one or more instructions, and check the top elements in the variable stack to choose a proper bytecode instruction.

Consider the statements on the left hand side of Listing 5.13. If the top 2 variables' identifiers are $\$fpgm\_24\_31$ and $\$fpgm\_24\_30$ on the variable stack, and the following 3 statements swap the 2 variables. If $\$fpgm\_24\_32$ is not used by the next fourth statement, we can translate the first 3 lines of the statements combined to a SWAP[] instruction.

Consider the code on the right hand side of Listing 5.13. The first assignment statement assigns a new variable with the top variable on the variable stack, and the following instruction assignments a constant to a new variable. We translate the first statement to a DUP[] instruction.

Since we ensured that no jump statements exist in our COI, the only control-flow statements we need to handle are if-then-else blocks and while blocks. We manipulate a block stack to track which block we are currently in. We push a new block to the block stack if we see either an if-then-else block or a while block. We pop the top block, and extend the instructions of this block to the next top if we have finished translating the statements in a block. We translate and append the instructions to the current block which is indicated by the top block instance on the stack. If the stack is empty, the current block is the main block.

We translate COI statements back to TrueType bytecode for global functions, the control value program and glyph programs separately, and replace the bytecode programs of the original TrueType font file with the optimized ones.

Finally, we implemented a program to render both the original font and the optimized font with different font sizes and resolutions using the external FreeType 2 [2] library, and check whether the bitmaps match each other. If the bitmaps of the optimized font match the original font, we can draw a conclusion that the optimized font is equivalent to its origin under the test conditions.

5.6 A Round Trip Example

We show a simple round trip example in this section. We made up this simple bytecode program, and it clearly demonstrates the translations and optimization stages of our tool. Assume that Figure 5.11 is a bytecode program of a font, and the font contains two simple global functions.

Figure 5.12 shows our COI representation of the bytecode programs according to the rules defined in section 5.3. Note that we have a declaration for each function, which is similar to high level programming languages, such as Java.

We assume that the lower bound of ppem is 10. Thus, the condition of the first if-then-else block is always true under the assumption, and this makes the storage area 0 concrete. Function 0 is called in the then branch who has a single instruction MDRP[ ]. As we mentioned in section 5.4.2, MDRP[ ] instruction has a side effect of resetting the reference point 2, so the statement in line 22 has no actual effects. Finally, the program extracts the value in storage area 0, which is 8 in this case, and compares it to 0. Thus the condition of the second if-then-else branch is always false under the assumption, and it causes the then-branch of the second if-then-else block unreachable.

Figure 5.13 shows the optimized COI representation. Since function 1 is uncalled under the assumption, we can remove function 1 from the global function table. We finally perform inverse translation, and get the optimized bytecode programs in Figure 5.14.

5.7 Limitations

Though our tool generates COI for most TrueType bytecode programs successfully, it does not work in some cases in which the stack depth assertion is violated. We can make some simple cases work by using some tricks. However, we have to skip all complicated cases. We can not remove the uncalled functions of the fonts whose bytecode programs have this problem, because the symbolic execution of the bytecode programs is not complete.

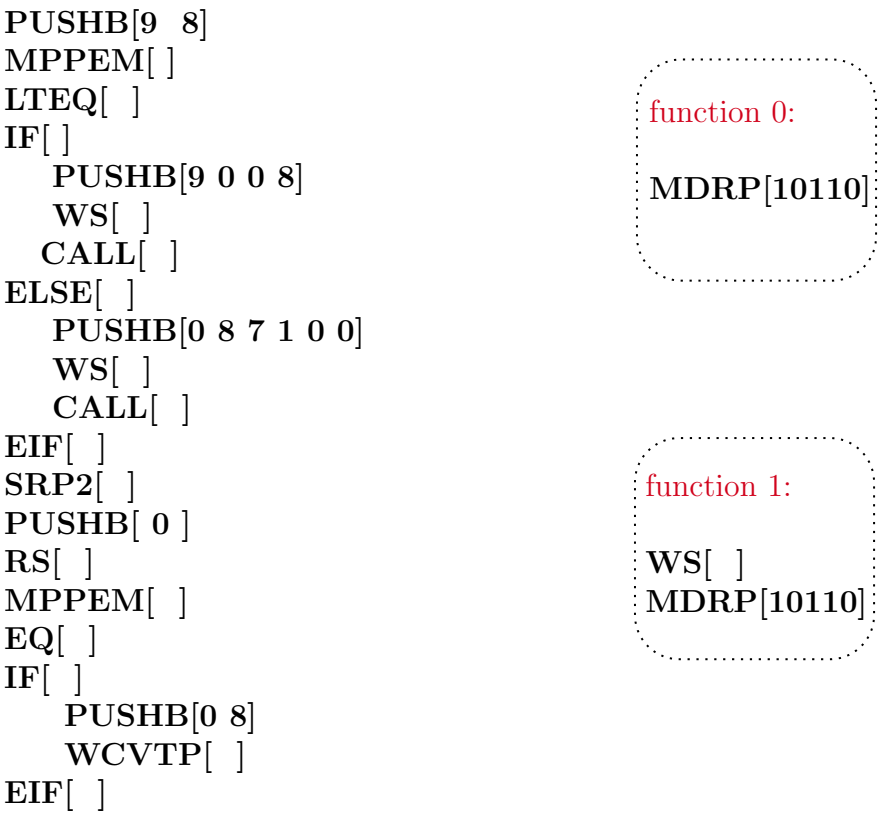

Figure 5.11: A simple bytecode program

```
1.  $var_1 := 9
2.  $var_2 := 8
3.  $var_3 := ppem(pv)
4.  $var_2 := $var_2 <= $var_3
5.  if ($var_2) {
6.  $var_2 := 9
7.  $var_3 := 0
8.  $var_4 := 0
9.  $var_5 := 8
10. storage_area[$var_4] := $var_5
11. call $var_3 ($var_2)
12. }else{
13.     $var_2 := 0
14.     $var_3 := 8
15.     $var_4 := 7
16.     $var_5 := 1
17.     $var_6 := 0
18.     $var_7 := 0
19.     storage_area[$var_6] := $var_7
20.     call $var_5 ($var_4, $var_3, $var_2)
21. }
22. graphics_state[rp2] := $var_1
23. $var_1 := 0
24. $var_1 := storage_area[$var_1]
25. $var_2 := ppem(pv)
26. $var_1 := $var_1 == $var_2
27. if ($var_1)
28. {
29.     $var_1 := 0
30.     $var_2 := 8
31.     cvt[$var_1] := $var_2
32. }
```

```
function 0 ($arg_1)
{
    MDRP ($arg_1)
}
```

```
function 1 ($arg_1, $arg_2, $arg_3)
{
    storage_area[$arg_2] := $arg_3
    MDRP ($arg_1)
}
```

Figure 5.12: COI representation of the bytecode programs

```
1.  $var_2 := 8
2.  $var_3 := ppem(pv)
3.  $var_2 := $var_2 <= $var_3
4.  $var_2 := 9
5.  $var_3 := 0
6.  $var_4 := 0
7.  $var_5 := 8
8.  storage_area[$var_4] := $var_5
9.  call $var_3 ($var_2)
10. $var_1 := 0
11. $var_1 := storage_area[$var_1]
12. $var_2 := ppem(pv)
13. $var_1 := $var_1 == $var_2
```

Figure 5.13: Optimized COI representation

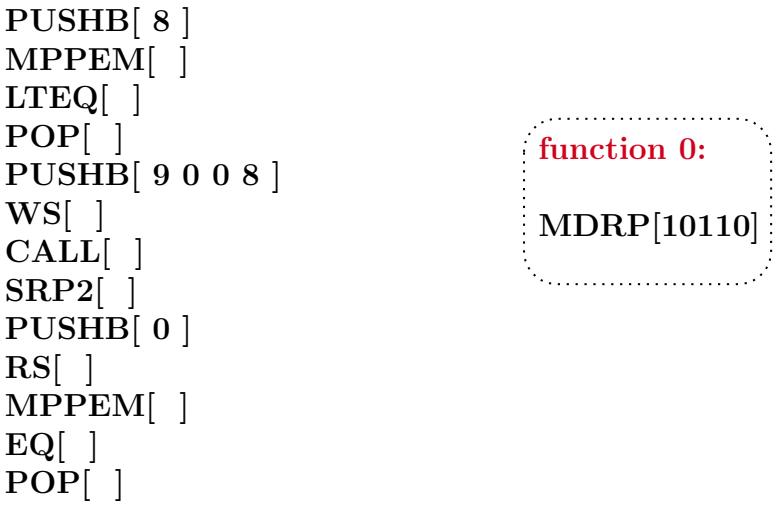

Figure 5.14: Optimized bytecode programs

5.8 Bytecode Debugger

We also developed a graphical tool which loads a font and performs symbolic execution on each single glyph. We can view the values and ranges of variables in the program stack and the graphics state for each single step.

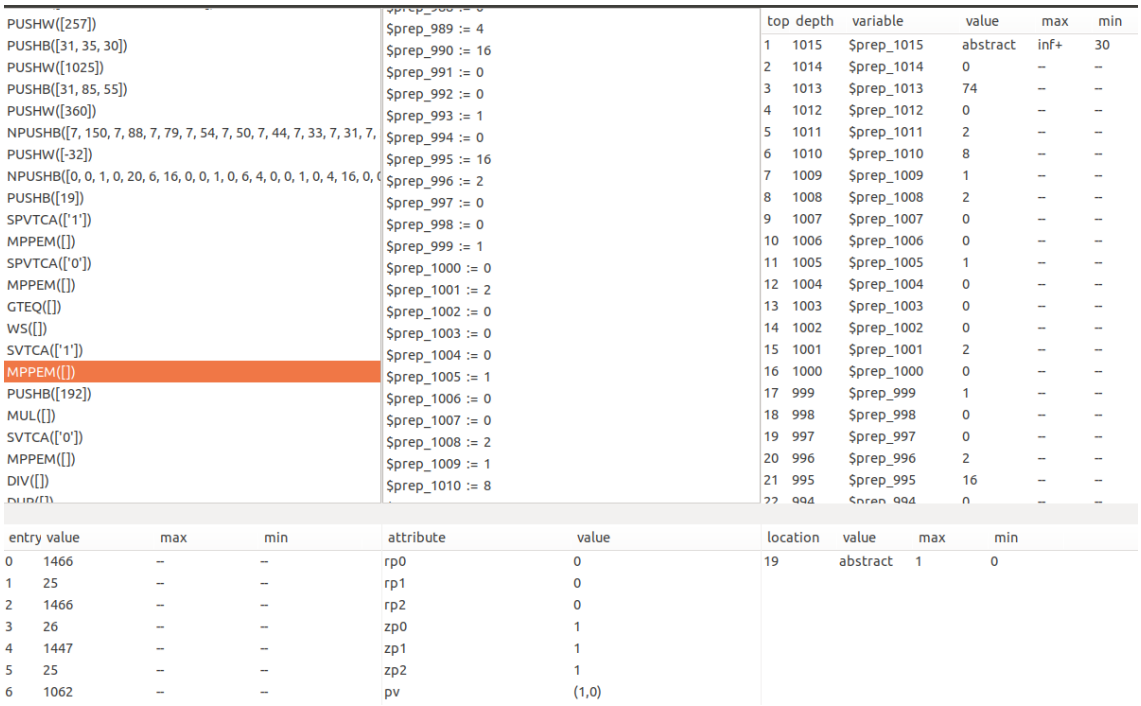

Figure 5.15: A screen shot of our GUI debugger tool.

Chapter 6

Experimental Results

We tried to run our tool on fonts distributed with Ubuntu 18.04.1. In this thesis, we report our results on 10 of these fonts which our tool successfully analyzed, covering Latin, Arabic and Tibetan scripts. These fonts include Microsoft Core TrueType font Arial, FreeMono-Bold, FreeSans-Bold, Loma-Bold, NotoKufiArabic-Bold, NotoSansTibetan-Bold. We specifically tried to run our tool on dozens of fonts from the NotoSans family which is commissioned by Google. Unfortunately, most of these fonts violate the assertion, and hence our tool does not support those fonts. We execute both the control value program and all the glyph programs of these test fonts. For these fonts, our tool can convert the bytecode programs to the improved COI correctly.

Table 6.1 shows the statistical information on our selected fonts. We performed the optimizations that we discussed in the previous chapter on the COI of the tested fonts. Finally, our tool converted the optimized COI back to TrueType bytecode and replaced the bytecodes in the original font files with the optimized bytecodes. We compared bitmaps generated by the original fonts to the optimized fonts at different resolutions, and their bitmaps match at tested conditions.

Problematic TrueType bytecode Our abstract executor requires that the program stacks have the same depth when merging from different branches. We tried to run our tool on dozens of fonts from NotoSans family which is commissioned by Google, and most of them violate the assertion.

We can fix some simple cases such as the program shown in Listing 6.1. However, we found that sometimes the unbalanced stack depth are recursive, which is not even possible

Font Name	#Glyphs	Size	#Control Value Program Bytecode	#Glyph Program Bytecode	#Function Program Bytecode
Microsoft Arial	1546	359Kb	1031	52191	1204
FreeMono-Bold	1059	294Kb	55	59945	481
NotoMono-Regular	815	106Kb	187	18173	1365
NotoNaskhArabic-Bold	1368	219Kb	107	28058	1502
NotoSansTibetan-Bold	1270	653Kb	263	114270	1489
UbuntuMono-B	1151	187Kb	105	23670	1206
Loma-Bold	282	82Kb	81	18587	481
OpenSans-Bold	786	220Kb	87	16502	1486
OpenSans-SemiboldItalic	773	208Kb	93	9188	1451
NotoKufiArabic-Bold	397	79Kb	56	6419	1489

Table 6.1: Statistical information on selected fonts

to fix. If this happens in the control value program, our tool stops immediately and we conclude that our tool does not support the font. If it is in a glyph program, our tool stops executing the current glyph program and continue executing the next glyph. In this case, we can not remove the uncalled functions, since the symbolic execution is incomplete.

```
1   original bytecode              modified bytecode
2   ...                            ...
3   IF[ ]                          IF[ ]
4      ADD[ ] //stack_size-1           ADD[ ]     // stack_size-1
5      ADD[ ] //stack_size-1           ADD[ ]     // stack_size-1
6   ELSE[ ]                            CLEAR[ ]   // stack_size=0
7      ADD[ ] //stack_size-1       ELSE[ ]
8   EIF[ ]                            ADD[ ]      // stack_size-1
9   CLEAR[ ] //stack_size=0           CLEAR[ ]    // stack_size=0
10  ...                            EIF[ ]
11                                 ...             // stack_size=0
```

Listing 6.1: Fixing a simple unbalanced stack if-else block by copying CLEAR to the end of both branches

In Listing 6.1, the then-branch reduces the depth of the program stack by 2 and the else-branch reduces the stack depth by 1, which seems to violate our assertion. But there is a CLEAR[] right after the EIF[], which empties the program stack. We can copy the CLEAR[] to the end of both branches, such that the stack depth becomes zero after both

branches without influencing to the result.

We present results of our font optimizations on our ten selected test fonts in the following sections

6.1 Uncalled Functions

Table 6.2 shows the number of functions and uncalled functions our abstract executor found in selected fonts. Since the abstract executor was not able to handle jump instructions and uncertain callee situations in previous work, it skipped glyphs which contain these structures. Hence it was not safe to remove the uncalled functions in previous work.

Our enhanced abstract executor now correctly handles both jump instructions and uncertain callees. So, our improved abstract executor identifies uncalled functions accurately, and the bytecodes of the uncalled functions can be safely removed from the TrueType fonts.

Function Name	#Functions	#Uncalled Functions	#Uncalled Function Code	#Called Functions in Prep	#Called Functions in Glyphs
Microsoft Arial	67	31	415	10	26
FreeMono-Bold	22	5	154	2	15
NotoMono-Regular	69	58	1078	5	6
NotoNaskhArabic-Bold	72	62	1126	4	6
NotoSansTibetan-Bold	71	62	1258	5	4
ubuntuBold-B	63	50	1002	4	9
Loma-Bold	22	5	154	2	15
OpenSans-Bold	71	8	1179	4	4
OpenSans-SemiboldItalic	71	7	1185	4	3
NotoKufiArabic-Bold	71	65	1290	4	2

Table 6.2: Uncalled Functions

We also recorded the number of called functions in both the control value program and glyph programs. This tells whether the subseting of the fonts reduces the function program largely. Even though a function is called by another function, it is marked as called in the control value program if the function is called when executing the control value program,

and the functions called in glyph programs are similar. The fonts of a family share the function program, which causes a large number of functions to be unnecessary in some fonts within the family.

Table 6.2 indicates that a large portion of functions are uncalled in our benchmarks. Note that these functions are uncalled even without assumptions about minimum ppem. We found that some fonts share the same set of global functions, and this may cause a large portion of the functions unused. Most fonts in our benchmarks contain a considerable number of uncalled functions. For example, 86.1% functions of Microsoft Core TrueType font Arial are uncalled, comprising 415 lines of bytecode; 84% functions of NotoMono-Regular are uncalled, comprising 1108 lines of bytecode. We symbolically executed the control value program and all the glyph programs of our test fonts, and identified the functions which are possible to be called correctly, so that we can remove the bytecodes of these uncalled functions safely.

In some test fonts, such as Microsoft Core TrueType font Arial, FreeMono-Bold, most of the called functions are called within glyph programs. More program functions can be removed if subsetting is applied to these fonts.

6.2 Instructions Without Effects

One of the optimizations we performed on test fonts is removing no-effect instructions (section 5.4.2). We focus on the candidate instructions which have potential to be no-effect. Table 6.3 shows the candidate no-effect instructions.

Table 6.3: Instructions which potentially have no effects (the Apple TrueType Font Reference Manual [6])

Number	Instruction	Operation
1	CLEAR	Clears the program stack.
2	FLIPOFF	Sets the auto flip Boolean in the graphics state to FALSE .
3	FLIPON	Sets the auto flip Boolean in the graphics state to TRUE.
4	CEILING	Takes the ceiling of the number at the top of the stack.
5	FLOOR	Takes the floor of the value at the top of the stack.
6	ROLL	Performs a circular shift of top 3 elements on the stack
7	RTG	Sets the round state variable to grid.
8	RTHG	Sets the round state variable to half grid.
9	RTDG	Sets the round state variable to double grid.
10	RUTG	Sets the round state variable to up to grid.
11	RDTG	Sets the round state variable to down to grid.
12	SCVTCI	Sets control value table cut-in to the top element.
13	SDB	Sets delta base to the top element.
14	SDS	Sets delta shift to the top element.
15	SMD	Sets the minimum distance variable to the top element.
16	SRP0	Sets Reference Point 0 to the top element.
17	SRP1	Sets Reference Point 1 to the top element.
18	SRP2	Sets Reference Point 2 to the top element.
19	SWAP	Swaps the top two elements of the program stack.
20	SZPS	Sets all three zone pointers to the top element.
21	SZP0	Sets zone pointer 0 to the top element.
22	SZP1	Sets zone pointer 1 to the top element.
23	SZP2	Sets zone pointer 2 to the top element.

For example, executing a CLEAR[] instruction has no effect if the stack depth is 0. CEILING[] and FLOOR[] do not change anything if the top element on the stack is a multiple of 64. ROLL[] instruction can be removed if the top 3 elements are identical. The instructions which set values in graphics state, such as FLIPOFF[]/FLIPON[], do not have effects if the corresponding values are what they will set them to before executing the instructions.

We chose candidate instructions which:

1. **do not move control points;**
2. **modify the values in the graphics state, or the program stack, but may have no actual effects on the values it intends to modify.**

Because the operations of point moving are absolutely abstract to our executor, we assume all the point moving instructions, such as MIRP[] and MDAP[], are un-removable.

Table 6.5 presents the number of candidate instructions and the number of no-effect instructions we can identify in our benchmarks using judgments similar to above. For each test font, the number of candidates is the counts of all candidate instructions in Table 6.3, and the number of no-effect instructions is the number of instructions which actually have no effects.

Font Name	#Candidates	#No-effect Instructions
Microsoft Arial	6084	244
FreeMono-Bold	5761	49
NotoMono-Regular	1748	48
NotoNaskhArabic-Bold	3919	326
NotoSansTibetan-Bold	41105	9597
UbuntuMono-B	2406	17
Loma-Bold	2171	54
OpenSans-Bold	4866	527
OpenSans-SemiboldItalic	1800	520
NotoKufiArabic-Bold	1322	209

Table 6.5: No-effect instructions in test fonts

This optimization reduces the bytecodes significantly in some fonts, but it does not yield significant optimization on all the fonts in our benchmarks. For example, We can locate 9597 instructions out of 41105 candidates in NotoSansTibentan-Bold; and 520 no-effect instructions in OpenSans-SemiboldItalic, which is 28.8% of its candidates. While in font FreeMono-Bold, we can only locate 49 no-effect instructions, which is less than 1% of its total candidate instructions.

Table 6.6 provides more statistical information on no-effect instructions on the test fonts, with respect to each candidate instruction. It shows the number of appearances

of each candidate instructions, as well as the number of no-effect instructions out of the corresponding candidate instruction in our benchmarks. It also shows the percentage of each instruction out of all candidates and its percentage in all no-effect instructions. This tells us what kinds of instructions are more likely to be no-effect.

Table 6.6: Statistical information on no-effect instructions in our benchmarks when lower bound of ppem is 15

Number	**Instruction**	**Counts**	**No-effect**	**% in all Candidates**	**% in all No-effects**
1	CLEAR	0	0	0	0
2	FLIPOFF	167	0	0.23	0
3	FLIPON	185	20	0.26	0.19
4	CEILING	0	0	0	0
5	FLOOR	0	0	0	0
6	ROLL	66	0	0.09	0
7	RTG	6964	6533	9.78	60.34
8	RTHG	297	1	0.42	0.01
9	RTDG	95	0	0.13	0
10	RUTG	4	0	0.01	0
11	RDTG	212	0	0.29	0
12	SCVTCI	0	0	0	0
13	SDB	2845	765	4.00	7.06
14	SDS	2783	2274	3.90	21.00
15	SMD	1138	45	1.60	0.42
16	SRP0	696	749	9.70	6.90
17	SRP1	34693	237	48.70	2.20
18	SRP2	14739	202	20.70	1.86
19	SWAP	34	0	0.05	0
20	SZPS	0	0	0	0
21	SZP0	0	0	0	0
22	SZP1	0	0	0	0
23	SZP2	0	0	0	0

We set the lower bound of ppem to 15 using our variable range system. We found 64918

candidate instructions in all the tested bytecode programs, and 10825 of them are actual no-effect instructions, which is 16.67% of the total size. Thus, these instructions can be removed from the glyph programs.

According to our experiments, RTG[], SDS[] and SDB[] contribute a lot to no-effect instructions. About 88.4% of no-effect instructions are one of the three instructions. To our surprise, about 93.8% of RTG[] (sets the round state variable to grid) have no effects to the graphics state.

Some instructions, such as FLIPON[], SRP0[], SRP1[], SPR2[], SMD[], also contribute to no-effect instructions, but the percentages are not so significant.

We also tested the performance of no-effect instruction removal with different lower bounds of ppem. The lower bounds we chose are 10, 15, 20, 30, 100. To our surprise, the number of no-effect instructions are the same with different lower bounds we tested.

6.3 Dead Block Elimination

Another optimization we have implemented is dead block elimination (section 5.4.3). We tested the efficiency of this optimization with different lower bounds of ppem. According to our experiments, the efficiency of dead block elimination is relative to the lower bound of ppem.

We split the instances of unreachable blocks into 2 groups. One of the groups contains the unreachable blocks in the control value program or the glyph programs, and the other group contains the ones in functions. We can erase the unreachable branches in the first group, as well as the their IF[]/ELSE[]/EIF[] instructions as soon as we see them, but we can not remove a branch inside of a function until we ensure that no call to this function ever enters the branch.

6.3.1 Dead Blocks in Functions

Table 6.8 illustrates the number of visits of if-then-else blocks with concrete conditions in global functions under the chosen lower bounds of ppem.

As shown in Table 6.8, the number of visited if-then-else blocks with concrete conditions increases with respect to the lower bound of ppem. We remove the branches in functions which are never visited. Table 6.9 illustrates the results of dead block removal in functions with respect to selected lower bounds of ppem. However, the results are not as good as we

Font Name	10	15	20	30	100
Microsoft Arial	771	3168	4989	5537	6247
FreeMono-Bold	10024	10046	10068	10109	12862
NotoMono-Regular	1	93	216	396	398
NotoNaskhArabic-Bold	0	123	302	514	592
NotoSansTibetan-Bold	0	140	545	685	722
UbuntuMono-B	29	241	435	584	632
Loma-Bold	2498	2499	2501	2051	3304
OpenSans-Bold	0	2	2	4	6
OpenSans-SemiboldItalic	0	9	23	32	33
NotoKufiArabic-Bold	0	0	7	10	12

Table 6.8: Visits of if-then-else blocks in functions, where the conditions are concrete

expected. We found that even though a lot of function calls only visit one of the branches of an if-then-else block, but other calls to this function visit the other branch.

Font Name	10	15	20	30	100
Microsoft Arial	0	0	0	0	10
FreeMono-Bold	23	23	23	23	57
NotoMono-Regular	0	0	0	12	12
NotoNaskhArabic-Bold	0	0	0	0	0
NotoSansTibetan-Bold	0	0	8	8	17
UbuntuMono-B	5	5	5	5	14
Loma-Bold	23	23	23	23	48
OpenSans-Bold	0	4	4	4	4
OpenSans-SemiboldItalic	0	0	0	0	4
NotoKufiArabic-Bold	0	0	0	0	9

Table 6.9: Removable dead code in function programs

Table 6.9 shows that we can only remove very limited lines of dead code from the function program. Note that the number of removable bytecodes increases with respect to lower bound of ppem.

6.3.2 Dead Blocks in Control Value Program and Glyph Programs

Table 6.10 illustrates the dead code in the control value programs and the glyph programs of our test fonts with respect to selected lower bounds of ppem. Unlike global functions, the control value programs and the glyph programs are executed exactly once, so all the unreachable branches detected can be removed directly.

Font Name	10	15	20	30	100
Microsoft Arial	0	6	31	108	251
FreeMono-Bold	5	5	5	8	8
NotoMono-Regular	0	0	0	0	6
NotoNaskhArabic-Bold	0	0	0	0	6
NotoSansTibetan-Bold	0	0	0	0	6
UbuntuMono-B	0	0	0	3	3
Loma-Bold	5	5	5	8	8
OpenSans-Bold	0	0	0	0	6
OpenSans-SemiboldItalic	0	9	0	0	6
NotoKufiArabic-Bold	0	0	0	0	6

Table 6.10: Removable dead code in control value programs and glyph programs

For Microsoft Core TrueType font Arial, we can remove 108 lines of code in the control value program and the glyph programs by dead block elimination when the lower bound of ppem is 30, and the number becomes 251 if we increase the lower bound of ppem to 100. Dead code elimination is not so effective on other fonts in our benchmarks. In fact, some test fonts, such as FreeMono-Bold and Loma-Bold, do not even contain if-then-else blocks in their control value programs or glyph programs, which impedes our optimization of dead code elimination.

Even though the effectiveness of dead block elimination is limited in our benchmarks, it is still valuable. Our abstract executor only executes one branch of the if-then-else block if the condition is concrete, which may make some variables concrete after the block. This may help us locate more no-effect instructions.

6.4 Font Subsetting

Some large fonts contain thousands of glyphs. In our benchmarks, Microsoft Core TrueType font Arial contains 1546 glyphs, and NotoSansTibetan-Bold contains 1270 glyphs. Web pages usually use a small portion of glyphs in these fonts. Font subsetting technologies are always applied to the fonts before the required fonts are attached to the webpage to save limited network bandwidth. However, existing subsetting software only subset the glyphs [18]. They do not subset the function programs, nor perform optimizations on the bytecode programs. This may cause the subset fonts to contain un-necessary bytecodes, which conflicts with the purpose of font subsetting.

Table 6.11 shows how much more bytecodes our tool can remove from the subset fonts than font subsetting software. We tested the font subsetting optimizations of 10%, 20% and 50% subsettings. We set the lower bound of ppem to 30 in this test.

Font Name	50%	20%	10%
Microsoft Arial	652	657	653
FreeMono-Bold	219	224	244
NotoMono-Regular	1107	1110	1110
NotoNaskhArabic-Bold	1162	1162	1229
NotoSansTibetan-Bold	1281	1281	1281
UbuntuMono-B	1019	1019	1019
Loma-Bold	210	215	219
OpenSans-Bold	1198	1198	1198
OpenSans-SemiboldItalic	1195	1195	1195
NotoKufiArabic-Bold	1305	1305	1305

Table 6.11: Bytecode removal with respect to font subsettings when the lower bound of ppem is 30

Table 6.12 illustrates the number of uncalled functions with respect to the percentage of subsetting. More functions of a font become uncalled if the font contains smaller subset of its glyphs for some test fonts.

So, our tool can optimize the bytecode, which enhanced the effectiveness of current font subsetting technologies.

Font Name	50%	20%	10%
Microsoft Arial	31	37	38
FreeMono-Bold	5	5	5
NotoMono-Regular	59	60	60
NotoNaskhArabic-Bold	63	63	65
NotoSansTibetan-Bold	62	62	71
UbuntuMono-B	50	54	50
Loma-Bold	5	5	5
OpenSans-Bold	64	64	64
OpenSans-SemiboldItalic	64	64	64
NotoKufiArabic-Bold	65	65	65

Table 6.12: Uncalled functions with respect to the percentage of subsetting, when the lower bound of ppem is 30

6.5 Overall Results

In this section, we show the overall results of our optimizations on our benchmarks. Figure 6.13 illustrates the performance of every optimization we implemented on our test fonts when we set the lower bound of ppem to 30. The table also shows the execution time of a round trip for each test font in seconds. Our new implementation runs much faster than the previous tool. Our improved tool runs more than 100 times faster, and consumes less than 1/50 memory as the prototype tool.

Our tool can optimize some fonts in our benchmarks significantly, with basic optimization methods widely used in compilers. As shown in Table 6.13, we can reduce the bytecodes of NotoKufiArabic-Bold by 18.82% , OpenSans-SemiboldItalic by 15.88%, and NotoSansTibetan-Bold by 9.36%. We erased over 10,000 bytecodes of NotoSansTibetan-Bold. However, our optimizations are not so successful on all fonts. For example, we only reduced the bytecodes of FreeMono-Bold by 0.37%, and Loma-Bold by 1.20%. So, our future work will focus on finding better optimization schemes which are designed specifically for TrueType bytecode.

Font Name	time (sec)	#code removed	% of code removed	#code in uncalled functions	#no-effect code	#code of unreachable blocks
Microsoft Arial	212	659	1.21	415	244	0
FreeMono-Bold	46	226	0.37	154	49	23
NotoMono-Regular	8	1138	5.76	1078	48	12
NotoNaskhArabic-Bold	19	1452	4.89	1126	326	0
NotoSansTibetan-Bold	96	10863	9.36	1258	9597	8
UbuntuMono-B	21	1024	4.09	1002	17	5
Loma-Bold	11	231	1.20	154	54	23
OpenSans-Bold	7	1710	9.47	1179	527	4
OpenSans-SemiboldItalic	6	1705	15.88	1185	520	0
NotoKufiArabic-Bold	3	1499	18.82	1290	209	0

Table 6.13: Overall effectiveness of optimizations when lower bound of ppem is 30

Chapter 7

Conclusion

This work advances the state of the art in the area of TrueType bytecode analysis. We improved the design of the abstract executor and the three address code. Our latest abstract executor handles all if-then-else blocks and jump instructions in practice, and we solved uncertain callee problems. Our tool successfully generates COI for TrueType bytecode programs in more cases. We introduced range propagation and a comparison system to our abstract executor. We performed some standard analysis and optimizations on COI with the assistance of the range propagation system. We finally completed the cycle by implementing the conversion from COI back to TrueType bytecode.

We tested our tool on 10 test fonts from different font families, including Microsoft Core TrueType font Arial, NotoSansTibetan-Bold, Loma-Bold, and it can successfully reduce the sizes of the bytecode program by 0.37% to 18.82%. Our tool reduces the bytecode significantly of some test fonts. For example, we can erase over 10,000 bytecodes from NotoSansTibetan-Bold, and we can reduce bytecode size by 18.82% of NotoKufiArabic-Bold. On average, we can reduce the size of bytecode of our test fonts by 7.10%.

To put our work in context, note that web traffic accounts for 17% of all Internet traffic [21]. Furthermore, almost 70% of websites include custom fonts, and the average website serves 95 KB of fonts per webpage [12]. We have observed that, for some popular fonts, bytecode accounts for almost half of their size—46% for Microsoft Core TrueType font Arial and 45% of NotoSansTibetan-Bold. We can calculate that our techniques can save about 0.5% of font size. Projected across all Internet traffic, even this modest 0.5% savings results in much bandwidth world-wide.

Chapter 8

Future Work

We found that a huge amount of bytecodes consist of point moving instructions, which are order sensitive. That means any modification (removal, insertion, order switching) to these instructions may largely affect the resulting bitmaps. The point moving operations are totally abstract for our abstract executor. It might be a good idea to explore how font engines adjust glyph outlines, and try to find methods to optimize these point moving instructions in the future.

For example, we could perform 'symbolic drawing' on an abstract canvas, along with the symbolic execution of bytecode programs. We could then replace sequences of point moving operations by more efficient sequences of instructions.

References

[1] The fonttools project. https://github.com/behdad/fonttools/.

[2] The freetype 2 project. https://www.freetype.org/. Accessed: 1-Oct-2018.

[3] Microsoft Visual TrueType (VTT). https://docs.microsoft.com/en-us/typography/tools/vtt/. Accessed: 2-Nov-2018.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, techniques, and tools (2nd edition), 1986.

[5] Saswat Anand, Corina S Păsăreanu, and Willem Visser. JPF–SE: A symbolic execution extension to Java PathFinder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138. Springer, 2007.

[6] Apple Inc. TrueType reference manual. https://developer.apple.com/fonts/TrueType-Reference-Manual/. Accessed: 2-Nov-2018.

[7] William Blume and Rudolf Eigenmann. Symbolic range propagation. In *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pages 357–363. IEEE, 1995.

[8] William R Bush, Jonathan D Pincus, and David J Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[9] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international conference on Software engineering*, pages 281–290. ACM, 2008.

[10] Delphine Demange, Thomas Jensen, and David Pichardie. A provably correct stackless intermediate representation for Java bytecode. In *Asian Symposium on Programming Languages and Systems*, pages 97–113. Springer, 2010.

[11] Ana M Erosa and Laurie J Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Computer Languages, 1994., Proceedings of the 1994 International Conference on*, pages 229–240. IEEE, 1994.

[12] Tammy Everts. The average web page is 3MB. how much should we care? https://speedcurve.com/blog/web-performance-page-bloat/. Accessed: 17-Dec-2018.

[13] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[14] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *International Static Analysis Symposium*, pages 95–111. Springer, 2011.

[15] Wenzhu Man. *COI: A First Step towards TrueType Bytecode Analysis*. Electrical and Computer Engineering, University of Waterloo, 2015.

[16] Corina S Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International journal on software tools for technology transfer*, 11(4):339, 2009.

[17] W Wesley Peterson, Tadao Kasami, and Nobuki Tokura. On the capabilities of while, repeat, and exit statements. *Communications of the ACM*, 16(8):503–512, 1973.

[18] Alberto Pettarin. Subsetting fonts with glyphIgo. https://www.albertopettarin.it/blog/2014/09/16/subsetting-fonts-with-glyphigo.html. Accessed: 3-Nov-2018.

[19] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 225–236. ACM, 2009.

[20] Daryl Shannon, Sukant Hajra, Alison Lee, Daiqian Zhan, and Sarfraz Khurshid. Abstracting symbolic execution with string analysis. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 13–22. IEEE, 2007.

[21] Todd Spangler. Netflix eats up 15% of all internet downstream traffic worldwide (study). *Variety*. Available at https://variety.com/2018/digital/news/netflix-15-percent-internet-bandwidth-worldwide-study-1202963207/.

[22] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[23] Clark Verbrugge, Laurie Hendren, et al. Generalized constant propagation a study in C. In *International Conference on Compiler Construction*, pages 74–90. Springer, 1996.