VISUALIZING MACHINE LEARNING EXPLANATIONS TO SUPPORT SOFTWARE REVERSE ENGINEERS

REPORT

Alec D. McGahee, Second Lieutenant, USAF

AFIT-ENG-MAS-22-S-031

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MAS-22-S-031

VISUALIZING MACHINE LEARNING EXPLANATIONS TO SUPPORT SOFTWARE REVERSE ENGINEERS

REPORT

Presented to the Faculty

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Cyber Systems

Alec D. McGahee, B.S.C.S.

Second Lieutenant, USAF

September 15, 2022

AFIT-ENG-MAS-22-S-031

VISUALIZING MACHINE LEARNING EXPLANATIONS TO SUPPORT SOFTWARE REVERSE ENGINEERS

REPORT

Alec D. McGahee, B.S.C.S.
Second Lieutenant, USAF

Committee Membership:

Wayne C. Henry, Ph.D
Chair

Scott R. Graham, Ph.D
Member

AFIT-ENG-MAS-22-S-031

Abstract

Software reverse engineering is a complex and time-consuming process. As a result, reverse engineers must use various tools to assist them while they look for patterns in binary code. Machine learning is a computing tool that excels in pattern recognition tasks. Researchers have used it in the cyber-security domain for finding patterns in software binaries that indicate the presence of code features such as vulnerabilities. However, the models used for these tasks are rarely explainable. They tend to act as black boxes because their inner workings are obscured and often difficult to understand. This obscurity makes it difficult for a user to determine which parts of the input contributed most to the model's decision. Having explanations of these models may help reverse engineers in analyzing software because it would guide them to portions of the program where the detected pattern exists.

This research develops a novel visualization tool that can display machine learning explanations as heat maps within Binary Ninja. We test the tool's ability to identify `functions starts`, `for loops`, and `switch statements` present in a software binary. We collect the same metrics from Binary Ninja's decompiler based on the representation produced by its high-level intermediate language. Then, we compare the precision statistics from the tool and the decompiler to determine if the tool has an advantage in identifying code features.

The experimental results show that the visualization tool and Binary Ninja's decompiler can identify `function starts` with 100% precision. The tool correctly identifies 76.47% of `for loops` and 91.67% of `switch statements`. Conversely, the decompiler was unable to represent any of the `for loops` accurately and only detected 12.5% of the `switch statements` present in the tested binary.

These results demonstrate that the visualization tool can detect certain code features with higher precision than a widely used decompiler. As a result, it has the potential to increase the efficiency of the reverse engineering process by reducing the time needed to recognize certain patterns in the binary. In future work, this visualization tool could supplement malware detection models and assist in analyzing malicious code.

Acknowledgements

I would like to thank my advisor, Lt Col Wayne Henry, for his continuous guidance and leadership throughout this process. I would also like to thank Daniel Koranek for inspiring me towards this work and helping me along the way. Finally, I am forever grateful for my family and friends who have provided me with constant support throughout my time at AFIT.

Alec D. McGahee

Table of Contents

Page

VISUALIZING MACHINE LEARNING EXPLANATIONS TO SUPPORT SOFTWARE REVERSE ENGINEERS

I. Introduction

1.1 Background and Motivation

Software reverse engineering (RE) is the process of using tools and techniques to make inferences about a software system from its binary code. Source code is not available during this process, so it is extremely difficult to decipher the software's original design. As a result, reverse engineers must use various tools to assist them in extracting information from these systems. RE tools play a paramount role in the productivity of the reverse engineer. Furthermore, tool developers are constantly looking for new ways to improve efficiency within this field.

Pattern recognition is an essential RE task that tools often look to improve. Recently, researchers have used machine learning's exceptional pattern-recognition abilities for RE tasks such as vulnerability detection [1, 2, 3]. However, the models used for these tasks are rarely explainable and their inner workings are often difficult to understand. In machine learning, researchers often refer to these models as "black boxes" because their inputs and outputs are visible but the processes the model uses to reach its decisions are obscured.

Explainability is a term that refers to the methods by which a model might explain itself. Researchers have developed various explainability methods to assist in comprehending machine learning models [4]. Layerwise Relevance Propagation (LRP) is a technique that helps to explain a model's output by assigning relevance scores to

portions of the input based on their importance to the model's decision. For software pattern-recognition tasks, the relevances produced by LRP could aid reverse engineers in understanding where a specific pattern occurs in the software.

1.2 Hypothesis and Research Objective

This capstone's aim is to develop a novel visualization tool that can produce heat maps of machine learning explanations within a popular RE platform. This research hypothesizes that these visualizations can act as a RE aid by providing a visual means of assisting reverse engineers in identifying specific patterns present in the software. To test this hypothesis, this research has the following objectives:

- Develop a RE tool that produces heat map visualizations of machine learning explanations inside a popular RE platform.
- Evaluate whether machine learning explanations can be used as a RE aid.
- Improve efficiency for RE pattern recognition tasks.

1.3 Approach

This research develops a plugin that creates heat maps of LRP relevance values within Binary Ninja's code viewing windows. The plugin uses explanations that originate from a separate research effort aimed at detecting code features in RISC-V binaries via machine learning [5]. After developing the plugin, we conduct an experiment to test its ability to identify `function starts`, `for loops`, and `switch statements` present in the executable for the `yes.c` file from the GNU coreutils package.

The experiment begins by first analyzing the source code for the compiled `yes` binary to determine the amount of program features present. Next, we calculate and present precision metrics from the plugin and Binary Ninja's high level intermediate

language (HLIL). These metrics are then compared to determine which tool has the advantage for detecting program features.

1.4 Research Contributions

To the researcher's knowledge, this is the first work that creates a RE tool using visualizations of machine learning explanations. This capstone demonstrates the potential for reverse engineers to use machine learning explanations as an analysis aid. If successful, this research expands the field of explainable artificial intelligence (AI) by providing a way for explanations to assist in the RE process.

1.5 Capstone Overview

This document is organized as follows. Chapter II provides an overview of relevant background information. Chapter III details the methodology used to develop and evaluate the system. Chapter IV presents and analyzes the experimental results. Finally, Chapter V summarizes the research, presents conclusions, discusses avenues for future work, and outlines limitations.

II. Background and Literature Review

2.1 Overview

Chapter 2 presents a high-level discussion of several relevant fields, including explainable artificial intelligence (AI), software classification and code feature detection, and software reverse engineering (RE).

2.2 Explainable Artificial Intelligence

AI is a broad computing term encompassing a variety of techniques employed by computers to emulate human-level problem-solving and decision-making capabilities [6]. Of these techniques, machine learning has become the most popular. Machine learning makes predictions about data through statistical algorithms [7]. Individual expressions of these machine learning algorithms are called models. Many general model types exist for performing machine learning tasks. Custom models can be created by making variations to existing model types or by combining them [8].

Machine learning's flexibility and accurate prediction measures have made it widely applicable for automating tasks that would otherwise have to be performed by a human [9]. However, the inner workings of many models are difficult to interpret, which means they tend to act as black boxes [10]. Consequently, machine learning's increased use has brought about ethical and legal questions regarding the models' inability to explain their own biases. Bias in AI can stem from many different sources; biased input and training data are the two most significant [11]. When developers skew this data to favor certain characteristics over others, the machine learning algorithm will inevitably produce biased results.

Explainable AI is a research field that has emerged in response to the need for models that are capable of explaining their processes and decisions. Besides providing

a way to understand and eliminate biases, explainability techniques also improve the usability of models. These techniques can help developers improve the model's accuracy while also providing information that will help users verify the model's decision [12].

2.2.1 Interpretability

Researchers often use the term *explainability* broadly to characterize how a machine learning model might explain itself. However, some have noted the importance of differentiating between the terms explainability and *interpretability* as they relate to the comprehension of a machine learning model [13, 4, 14]. Chakraborty et al. describe intepretability as the user's ability to understand and reason about a model's output, whereas explainability refers to the type and completeness of the output [15]. The National Institute of Standards and Technology (NIST) supplies the following definitions [13].

- Intepretability: "the ability to contextualize a model's ouput in a manner that relates it to the system's designed functional purpose, and the goals, values, and preferences of the end users."
- Explainability: "the ability to accurately describe the mechanism, or implementation, that led to an algorithm's ouput, often so that the algorithm can be improved in some way."

NIST also notes that the concepts of interpretation and explanation are closely intertwined in the minds of the user [13]. Depending on the user's background knowledge and willingness to utilize that knowledge, explanations can have different effects on a user's interpretations of a model.

Users who are not well-versed in machine learning may find detailed mechanistic explanations difficult to understand. Therefore, these explanations must be presented

in a understandable and relevant way. This research examines whether visualizing machine learning explanations can support model interpretability without requiring users to have machine learning expertise.

2.2.2 Layerwise Relevance Propagation (LRP)

Many different explainability methods exist to assist users in understanding a model's processes and decisions. This research uses results produced by LRP to create visualizations. LRP is an explainability method that can be added to a trained machine learning model to explain its output by assigning relevance scores to portions of the input based on how much they contribute to the model's decision. To do this, LRP performs a backwards relevance propagation of the output through a layered classifier back to its inputs. Figure 1 demonstrates this propogation.

In many cases, supplementing a machine learning model with explanations produced by LRP could significantly enhance the models' usefulness [17]. Researchers using machine learning to analyze neuroimaging data have used LRP to study associations between brain activity and cognitive state by mapping the contributions of the input to the model's decisions [2]. Similarly, Grezmak et al. used heat-mapping to visualize LRP results to improve fault diagnosis in industrial machinery [18].

These practical implementations demonstrate that LRP has significant potential to improve model effectiveness because of its ability to create associations between inputs and model decisions. Another potential research area where LRP could prove beneficial is software RE. This research investigates whether heat map visualizations of LRP results could benefit reverse engineers analyzing software binaries.

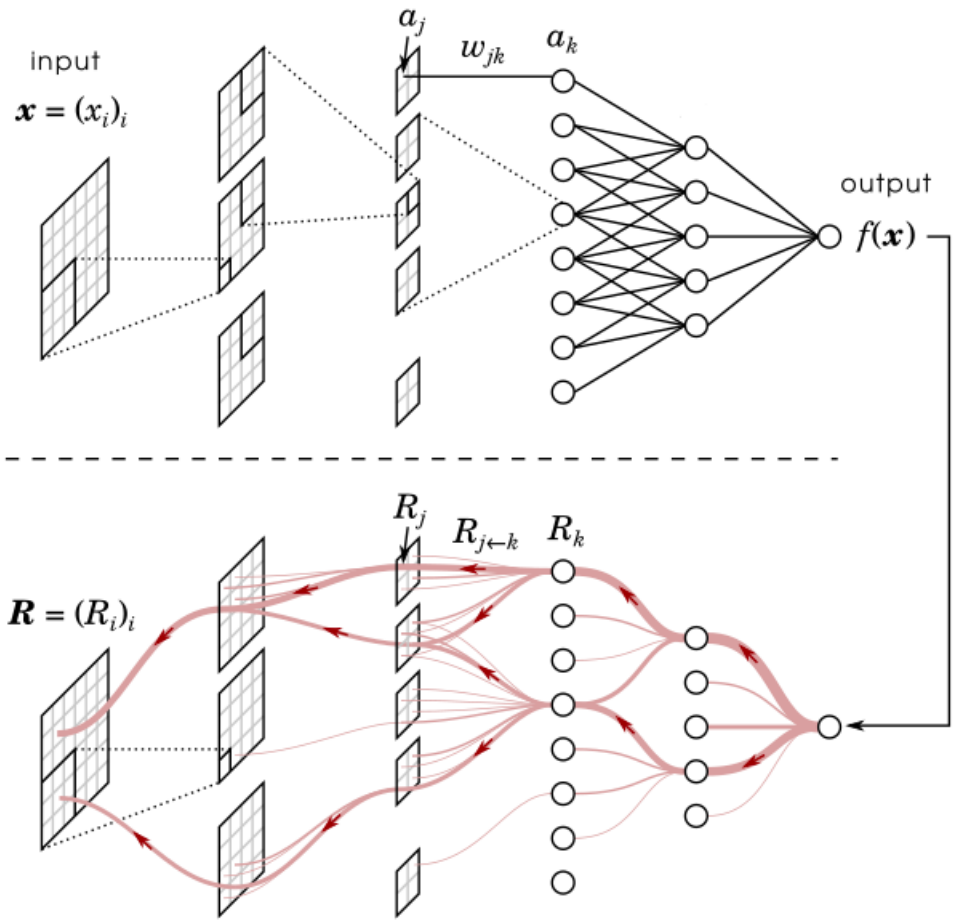

Figure 1: Forward pass and backwards relevance pass on a neural network [16].

2.3 Software Classification and Code Feature Detection

Machine learning, specifically deep learning, is used widely for feature detection tasks. These tasks are most often concerned with recognizing image features [1, 2]. Recently, researchers have used similar techniques for code feature recognition and software classification. Shar et al. used machine learning to detect static code attributes that cause a software system to be vulnerable to SQL injection and cross site scripting attacks [19]. Similarly, Grieco et al. used machine learning to detect security vulnerabilities in computer operating systems [20]. Gavriluţ et al. implemented machine learning to successfully classify files as being malicious or clean [3].

These efforts all use machine learning to recognize specific patterns in code. The models predict whether specific features are present in the software they are analyzing. While this information is essential, explanations that describe what portions of the input contribute most to the model's decision could be valuable data for reverse engineers.

2.3.1 RISC-V

RISC-V is an open-source instruction set architecture that is seeing increased use, especially within embedded devices. RISC-V is advantageous because it doesn't require licensing fees and facilitates customization [21]. RISC-V uses a load-store architecture with either 4-byte or compressed 2-byte instructions that are word-aligned.

Koranek et al. uses execution trace data from RISC-V programs to evaluate different machine learning detection techniques for detecting return-oriented program attacks [5]. This research builds on that work by developing a tool that can visualize explanations from machine learning models designed to detect code features present in RISC-V binaries.

2.4 Reverse Engineering

In their seminal work, Chikofsky and Cross define RE as "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction" [22]. In the context of software, this process can be modeled as an iterative sensemaking process. Bryant defines the *sensemaking process* for RE as "a goal directed planning-based activity, in which the reverse engineer interacts with an executable program using reverse engineering tools to construct a mental model of the functionality of the program" [23]. To construct this mental model, reverse

engineers engage in an iterative process where they must develop hypotheses, create goals, sense information, and update their knowledge base. Figure 2 shows a visual representation of the RE sensemaking process.

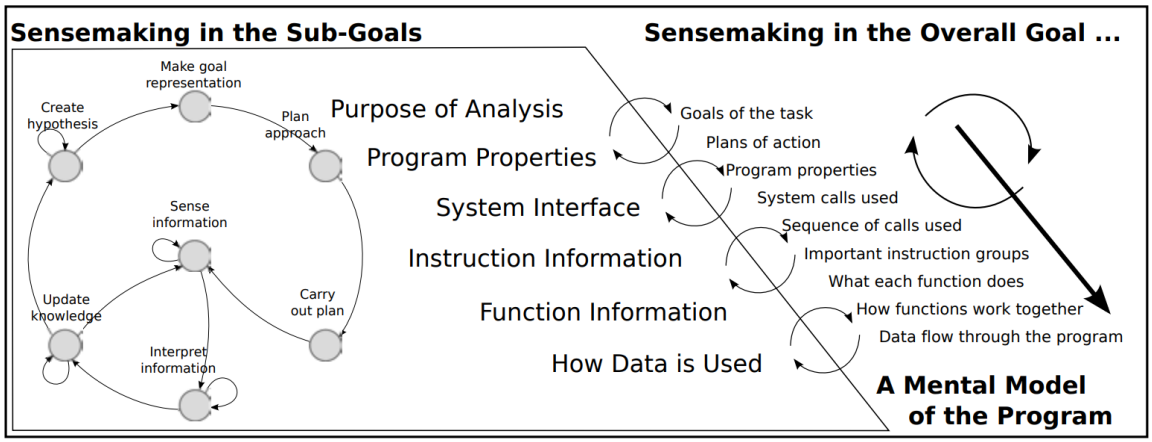

Figure 2: Reverse engineers' iterative sensemaking process [23]

To sense information from a program, analysts develop and test hypotheses through static or dynamic analysis [24]. Static analysis examines a program's code or structure to determine its functionality without executing it [9].

Examining a program's code is often broken into two stages: disassembly and decompilation [25]. Both stages involve converting executable machine code into a human readable format. The key differences between the two stages are the format that the tool converts the code to and the level of abstraction involved. Decompilation transforms machine code into a higher level language while disassembly simply creates a one-to-one mapping of processor instruction codes to assembly language instructions [9]. Decompilers provide a more concise and understandable code representation, but disassemblers provide more granularity [26]. Reverse engineers use both extensively.

Static analysis techniques provide a comprehensive view of the program because they can explore all execution paths a program may take [24]. However, the sheer quantity of information provided can be overwhelming and difficult to follow [27].

Dynamically analyzing a program is a more focused approach because it extracts information from program execution traces captured while the program is running [28]. Monitoring the program while it runs allows the analyst to observe behavior that would have been difficult to deduce through static analysis alone [9]. Tools commonly used for dynamic analysis include debuggers, dynamic binary instrumentation tools, and visualization environments [24].

Processor-trace (PT) data is another source of information that can support dynamic analysis. PT is a real-time hardware monitoring functionality that captures events within a central processing unit (CPU), saves the associated data, and presents the data in a human readable form [29]. Many different tools exist for collecting this data. Perf is a Linux tool that uses performance counters on CPUs to profile commonly executed code regions [30]. Intel Processor Trace [31] and ARM CoreSight [32] are other execution monitoring tools that are built into their respective architectures. The data obtained from these monitoring tools can help debug the CPU and enhance program performance. In addition, several research efforts have used this data with machine learning to detect computer attacks [33, 34, 35]. This research effort uses explanations from a machine learning model that analyzes PT data to detect specific code features present in RISC-V programs [5].

2.4.1 Tools

Software RE is a complex and arduous process [36]. Compiling a program causes it to lose much of the syntax and language that its authors wrote it with. Therefore, reverse engineers must perform a painstaking analysis process to make inferences about the program using the methods described above [37]. Such analysis can require reverse engineers to comb through thousands of assembly-level instructions and perform hundreds of steps to understand the program's functionality [24, 36]. As a

result, science and industry are constantly developing tools to support and enhance the efficiency of this process. Reverse engineers use many popular commercial-level static analysis tool sets [38, 39, 40]. This research will primarily focus on Binary Ninja and its ability to disassemble and decompile software binaries.

Binary Ninja is an interactive binary analysis tool capable of disassembly and decompilation [40]. It offers both graph and linear views of disassembled and decompiled code. The graph view allows for the visualization of control flow information, including jumps and function calls. The linear view offers a continuous and complete view of the code. Binary Ninja offers code representations in the following forms: disassembly, low-level intermediate language, medium-level intermediate language, high-level intermediate language, psuedo C, and other advanced intermediate languages.

Along with its disassembler and decompiler, Binary Ninja offers the ability to examine program strings, symbols, and dependencies. Binary Ninja includes robust annotation features that allow highlighting, renaming, and commenting. In addition, its application programming interface (API) supports the development of custom plugins that can be built in C++, python, and rust [40]. A key feature of Binary Ninja is its ability to represent decompiled code as a linear and complete view of the program instead of function by function. As a result, Binary Ninja can propagate annotation features such as highlighting, commenting, and renaming amongst the different representations.

2.4.2 Visualizations

Visual learning is one of the primary avenues through which humans absorb information [41]. Humans' high bandwidth optical system and ability to quickly recognize patterns makes visualizations uniquely beneficial for text saturated domains such as software RE. Many of the most popular RE tools come with visualization features

built-in. For example, Binary Ninja includes a graph view that visualizes program control flow, a color-coded navigation bar to differentiate between program sections, and highlighting features that can add color to individual lines of code [40]. This research uses Binary Ninja's highlighting features to create colored heat maps that support software RE.

Heatmapping is a visualization technique that uses a color-coding system to represent different values [42]. Typically, heatmapping displays values across a color spectrum where one end of the color spectrum represents higher or more important values and the other end of the spectrum represents lower or less important values.

Heat maps are a popular way to visualize results from the LRP explainability method. As discussed in Section 2.2.2, LRP assigns relevance scores to input values based on their importance to a machine learning model's decision. These results act as a heat map without color coding. Several researchers have used colored heat maps in conjunction with LRP to bolster model explainability [43, 44, 45]. However, there has yet to be an effort to use heat maps and LRP as a RE aid. This research effort fills this gap by exploring whether heat maps produced by LRP for a program feature detection model can aid reverse engineers in their analysis process.

2.5 Background Summary

The increased use of machine learning brings about the need for models capable of explaining themselves. Researchers have developed several explainability methods to address this need. LRP is one method that can explain a model's output in terms of its input. Several researchers have used LRP's ability to create associations between a model's decisions and its inputs to support the interpretability of their models [2, 18]. In RE, supplementing a code feature detection model with LRP could have a practical benefit as an analysis aid. To our knowledge, little to no research

has explored whether explanations can be used to identify assembly code features present in software. This research aims to fill this gap by developing and testing a visualization tool that uses explanations to identify code features.

III. Methodology

Introduction

This chapter presents the methodology used to develop and test a visualization tool that produces heat maps of relevance values within a reverse engineering (RE) platform. First, we conduct a platform analysis to choose which platform to build the tool in. Next, we discuss the system's design and present the tool's operation. Finally, we outline the experimental methodology used to test and evaluate the tool.

3.1 Platform Analysis

A preliminary analysis was conducted to the select the most appropriate RE platform. While there are dozens of RE platforms to choose from, we chose to evaluate Ghidra and Binary Ninja because of their robust decompilers. This analysis investigated the following characteristics of each platform:

- **Application Programming Interface (API)**: This characteristic determines whether the tool can the operate the platform's visualization features within the code viewing windows. In particular, the platform's API must support the ability to highlight instructions in the decompiler window.
- **Highlight Consistency**: When the tool highlights instructions, the highlights must be able to propagate amongst the various code representations. This feature allows reverse engineers to access the tool's heat maps in whichever representation they choose to use.
- **Decompilation Window**: The method by which the decompilation window displays functions is important to the tool's use. Preferably, the window will

show a continuous view of the code so users can scroll through the entire program without having to select functions one-by-one.

3.1.1 Ghidra

We evaluated Ghidra first because it is the most robust open-source tool available to reverse engineers. Users begin their analysis in Ghidra by starting a new project and importing a compiled program. Users then open the program to view disassembled and decompiled representations of its code. At this point, users have the option to highlight instructions in the disassembly or decompilation window by right-clicking a line and selecting "colors." However, highlights produced in one window will not propagate to the other. Additionally, the decompiler window will only display the last selected function, meaning users must select the function within the disassembly window or function tree to see its highlights. Figure 3 compares Ghidra's disassembly and decompilation windows. Ghidra's API allows plugins to interact with the interface's highlighting features for the disassembly window but not for the decompilation window.

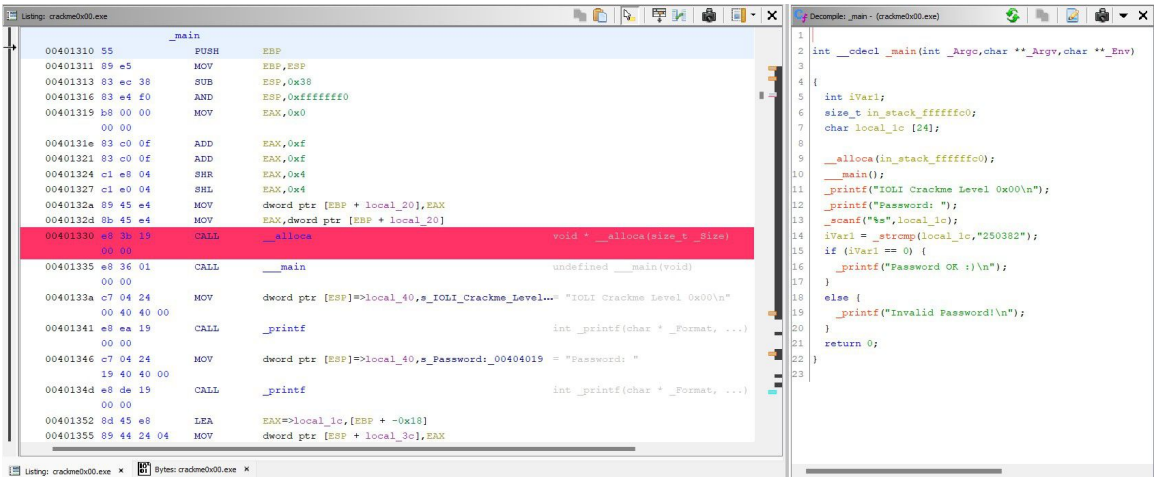

Figure 3: Ghidra's disassembly (left) and decompilation (right) windows

The plugin must be able to produce heat maps in the various code representations

available to the reverse engineer. Ghidra's API does not allow plugins to control highlights within the decompilation window, and it will not translate highlights from one window to the other. These limitations make Ghidra incompatible with the needs of the tool.

3.1.2 Binary Ninja

Binary Ninja is a close-source RE platform developed by Vector35 [40]. After users select a file to analyze, they have the option to highlight instructions by right-clicking and selecting "highlight instruction." Unlike Ghidra, highlights in Binary Ninja propagate among the different code representations. Highlighting an instruction in the disassembly view will highlight the closest related instruction in higher-level representations. Figure 4 demonstrates this functionality.

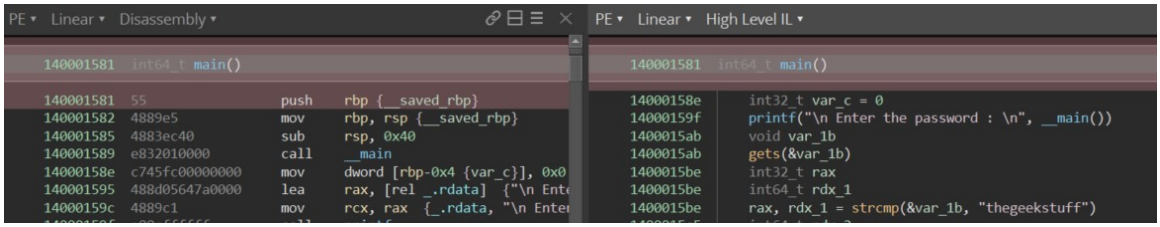

Figure 4: Binary Ninja's disassembly (left) and decompilation (right) windows

Binary Ninja's decompiler views display a continuous stream of code rather than a single function. This feature allows users to scroll through an entire program and see every decompiled function without having to select it first (as in the case of Ghidra). Binary Ninja's API allows Python plugins to operate the interface's highlighting features by referencing specific instruction addresses. Thus, plugins can produce highlights that are displayed in every code representation as long as the referenced addresses appear in each representation. Overall, Binary Ninja's API, highlight consistency, and decompiler windows all support the needs of the tool, and was therefore chosen for tool development in this research effort.

3.2 System Design

This section outlines a plugin that produces heat maps of Layerwise Relevance Propagation (LRP) relevance values within Binary Ninja. Figure 5 shows an overview of the system. We build the plugin in Python and interface it with Binary Ninja using the API. The plugin takes a `.pickle` file that contains a serialized relevance dictionary as input. Then, it calculates a color value within the Red-Green-Blue (RGB) spectrum for each instruction in the dictionary. Finally, it uses Binary Ninja's API to highlight the instructions based on their color value.

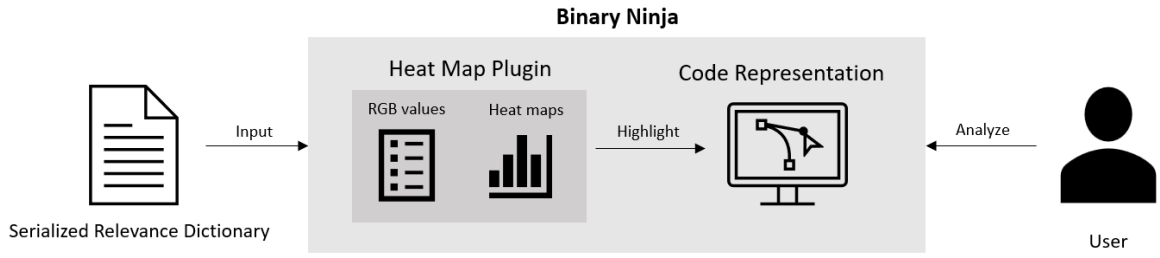

Figure 5: System Design

3.2.1 Input

LRP data is provided as a serialized relevance dictionary in a `.pickle` file. Pickle is a Python library that converts an object in memory into a byte stream that users can store as a file on disk. When a `.pickle` file is loaded into a program, the Pickle library can also convert the file back into a Python object. The dictionary used for this research contains instruction addresses with assigned relevance values.

3.2.2 Calculating Color Values

Once the plugin extracts the dictionary from the `.pickle` file, it finds the highest relevance score by iterating through the dictionary. The plugin then assigns the

instruction with the highest relevance score an RGB red value of 255. The plugin iterates through the dictionary again and calculates a red color value for each instruction based on its ratio to the highest relevance score. The plugin pairs these color values with their associated instructions in a new Python dictionary. Figure 6 displays the plugin's code for calculating color values.

```
    #Calculate Color Values
    for x in relevances:

        #ignore first value
        if x == "name":
            continue

        #throw away values less than 0
        elif float(relevances[x]) < 0:
            continue

        #if value greater than zero, calculate rgb score and add to
           dict.
        else:
            rgbVal = (float(relevances[x])/base)*255
            heatmap[x]=rgbVal
```

Figure 6: Calculating color values

3.2.3 Generating Heat Maps

The plugin uses the color values dictionary to produce highlights within Binary Ninja. For each instruction in the dictionary, the plugin saves the function it is contained within using the `get_function_at` method in Binary Ninja's `binaryview` module. The plugin can then act on the saved function as a Binary Ninja `function` module. A Binary Ninja `function` module contains several methods, one of which is the `set_user_instr_highlight` method. The plugin passes the instruction's address and red color value as arguments into the `set_user_instr_highlight` method to highlight the instruction. Every instruction in the dictionary is highlighted with this method. Figure 7 displays the code for generating heat maps.

```
#Generate heat maps
for x in heatmap:

    currentAddr=int(x,16)
    previousFunction = bv.get_previous_function_start_before(
        currentAddr)
    cfunc=bv.get_function_at(previousFunction)
    print(cfunc)
    print(math.ceil(heatmap[x]))
    cfunc.set_user_instr_highlight(currentAddr, highlight.
        HighlightColor(red = math.ceil(heatmap[x]), blue = 0,
        green = 0))
```

Figure 7: Generating heat maps

This highlighting process creates a heat map within Binary Ninja with color values from 0 to 255. Figure 8 shows an example heat map within Binary Ninja. The intensity of the color directly relates to the value of the relevance score. More intense highlights represent higher relevances, while less intense highlights represent lower relevances. The full source code for the plugin can be found in Appendix A.

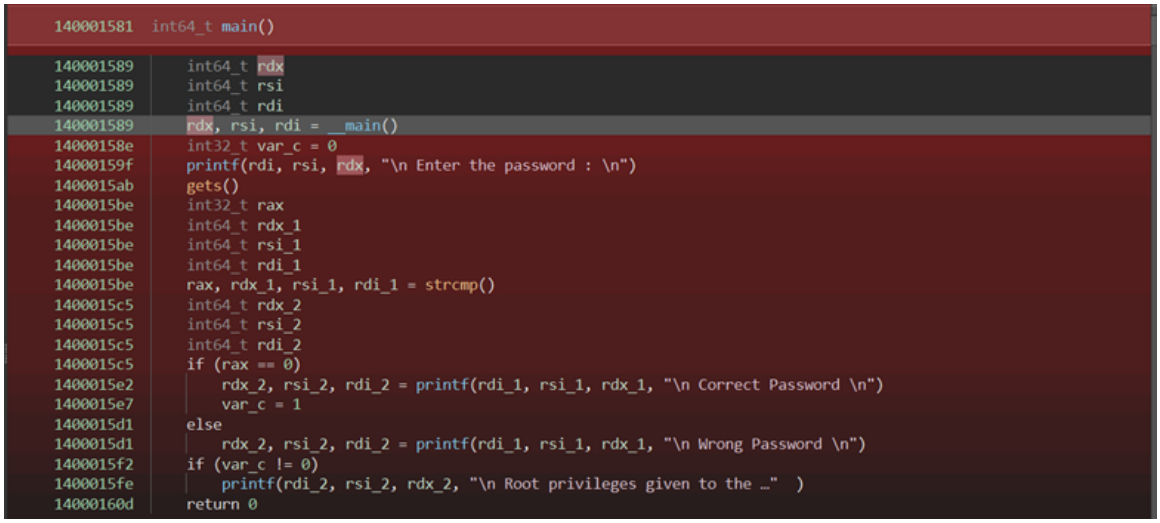

Figure 8: Example heat map produced in Binary Ninja

3.3 Experimental Methodology

3.3.1 Problem/Objective

This research evaluates whether heat maps of LRP relevance values from a program feature detection model can act as a RE aid. RE is a complex and time-consuming process. Therefore, any information that helps reverse engineers recognize patterns in a program has the potential to be a valuable addition to their toolset.

LRP assigns relevance values to portions of an input based on how much that portion contributes to the model's decision. If a model's output correlates with something that a reverse engineer is looking for, visualizations describing how the model reached its decision could be of great use to the reverse engineer.

To evaluate whether heat maps of LRP relevance values have any potential benefit to reverse engineers, we test whether the heat maps can identify low-level program features. In addition, we calculate how precisely Binary Ninja's high level intermediate language (HLIL) can identify these program features to determine whether the plugin has any advantage over Binary Ninja's decompiler.

3.3.2 System Under Test

The System Under Test (SUT) is the Binary Ninja heat map plugin, shown in Figure 9. The system comprises Binary Ninja's HLIL, the program's source code, and the plugin's heat maps, the Component Under Test (CUT).

3.3.3 Metrics

The experiment produces the following metrics to measure and assess the system.

- **Source Code Program Features**: The source code for each function in the `yes` executable is analyzed to count the number of program features present.

- **HLIL Program Features**: The functions within the `yes` executable are analyzed within the HLIL view in Binary Ninja to determine how many program features Binary Ninja's decompiler can recognize.
- **Heat Map Program Features**: Once the plugin is run, we measure the number of program features the heat maps correctly identify.

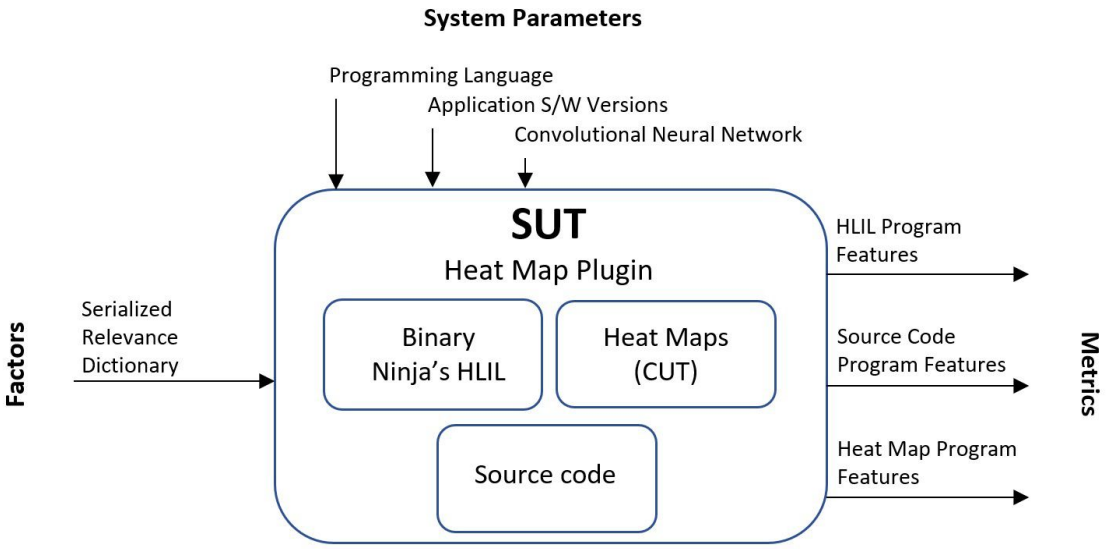

Figure 9: System Under Test (SUT)

3.3.4 Factors

The experiment provides the following inputs to the system.

- **Serialized Relevance Dictionary**: The relevance data for each program feature is formatted as a dictionary that is serialized into a `.pickle` file. Each program feature has a separate `.pickle` file. The experiment produces precision statistics for `function starts`, `for loops`, and `switch statements`.

3.3.5 System Parameters

The following system parameters are kept constant throughout the experiment.

- **Binary Ninja Version**: The experiment uses Binary Ninja version 3.1.3469 Personal.
- **Programming Language**: Python 3.9 is used to develop and run the plugin.
- **Convolutional Neural Network (CNN)**: The CNN used to produce the relevance data has five convolutional layers, a max pooling layer, and three dense layers. This CNN was built for a separate research effort that uses machine learning to detect program features in RISC-V binaries [5].

3.3.6 Uncontrolled Variables

Currently, Binary Ninja does not provide native support for the RISC-V architecture. Bn-riscv is a community plugin that allows Binary Ninja to load RISC-V binaries in the Executable and Linking Format [46]. Because Binary Ninja can only load RISC-V binaries via a community plugin, we cannot give high confidence in its ability to decompile RISC-V binaries to the same degree it can for those compiled in architectures that Binary Ninja natively supports.

3.3.7 Experimental Design

The experiment is performed on a Hewlett-Packard laptop with an Intel Core i7-8665U central processing unit (CPU), 32 GB of random access memory (RAM), and a 64-bit Windows 11 Home operating system. To start, we download the `yes.c` file from the GNU coreutils repository on Github to a Ubuntu 21.04 virtual machine. Once downloaded, we compile the `yes.c` file using a RISC-V version of the gcc compiler with the options `-g` and `-Wno-error`. Next, we create an objdump file of the compiled `yes` executable using the following command:

```
riscv64-linux-gnu-objdump -l --source-comment yes | grep -e '^\/' -e
'^#' > yes.obj
```

This command produces a printout of the various functions present in the `yes` executable. It also provides the location of the source code for each function. We then analyze each function's source code to count the number of program features present.

Once we identify the number of source code program features, we open the executable in the HLIL view within Binary Ninja on the experiment laptop. Next, we manually analyze the decompiled program to count the number of program features that Binary Ninja's HLIL can correctly represent. Finally, we run the heat map plugin for each program feature. Each program feature has a separate `.pickle` file containing its associated relevance data. We run the plugin for each `.pickle` file, resulting in three test runs total. For each test run, an analyst measures the amount of the program features that the heat map identified within each function.

3.3.8 Statistical Analysis

One of the ways that machine learning performance is measured is through the relative proportion of samples classified correctly and incorrectly. In a statistical context, *accuracy*, *precision*, and *recall* are ways of representing these proportions. Accuracy is defined as

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

while precision is defined as

$$Precision = \frac{TP}{TP + FP}$$

and recall is defined as

$$Recall = \frac{TP}{TP + FN}$$

True positives (TP) and false positives (FP) refer to whether a prediction matches a positive value within the ground truth. True negatives (TN) and false negatives (FN) refer to whether a prediction matches a negative value within the ground truth. For this research, a positive value represents the existence of a program feature, while a negative value represents the absence of a program feature. The ground truth is the source code for the binary being analyzed.

The experiment measures how well the heat maps produced by our Binary Ninja plugin correlate with the source code regarding the prevalence of certain program features. Therefore, the primary concern is positive values. We use precision to evaluate the system by dividing the amount of program features the heat maps or Binary Ninja's HLIL can identify (true positives) by the total amount present in the source code (overall positives). These ratios are given in the form of percentages. The experiment measures how precisely the plugin and the HLIL can identify all three program features, resulting in six total precision metrics.

```
00001a14  int64_t emit_ancillary_info(int64_t arg1, int64_t arg2, void* arg3, int64_t arg4, int64_t arg5, int64_t arg6)

00001a16      int64_t ra
00001a16      arg_a8 = ra
00001a18      int64_t s0
00001a18      arg_a0 = s0
00001a1c      int64_t a0
00001a1c      arg_8 = a0
00001a2a      arg_98 = **0x199b8
00001a46      sub_17a0()
00001a4e      arg_10 = arg1
00001a56      arg_18 = &arg_28
00001a6c      while (*arg3 != 0)
```

Figure 10: True positive `function start`

For the plugin, true positives are counted by determining whether highlights are produced in the functions that contain the program features. Figure 10 displays `function start` that is considered a true positive.

For Binary Ninja's HLIL, true positives are counted by identifying the locations in the psuedo-code where it was able to reproduce the program feature. Figure 11 displays a `switch statement` that is considered to be a true positive for Binary Ninja's HLIL.

```
000044d4        switch (arg1)
000044fc            case 1
000044fc                sub_1670()
00004506                *arg2
00004510                a0_11 = sub_1650()
0000451e            case 2
0000451e                int64_t a0_16
0000451e                void* a4_6
0000451e                a0_16, a4_6 = sub_1670()
00004528                *a4_6
00004530                *(arg2 + 8)
0000453a                a0_11 = sub_1650()
```

Figure 11: True positive `switch statement`

Our assessment of precision was based on correlating heat maps with Binary Ninja's HLIL representation, and then cross-checking that with source code. However, correlation of these data sources was not always possible. True negatives were difficult to establish because it required authoritatively stating that a code feature did not exist at an assembly location. For this reason, we avoid measures of performance that require measuring true negatives.

IV. Results and Analysis

Overview

This chapter presents the experimental results of the Binary Ninja heat map plugin presented in Chapter 3. First, we present how many program features are in the source code of the `yes` executable. Next, we discuss and analyze precisions for each program feature category. These precision statistics are calculated using the true positive results from the plugin's heat maps and Binary Ninja's high level intermediate language (HLIL).

4.1 System Precision

After producing the objdump for the `yes` executable, we discovered that the executable contains 116 functions originating from 25 different source files. After analyzing each function's individual source code, we counted 17 `for loops` and 24 `switch statements` present in the entire executable. These metrics serve as the ground truth positive values that will be used in the precision statistics.

4.1.1 Function Start Precision

Both the plugin and Binary Ninja's HLIL detect all 116 `function starts`, resulting in 100% precision. We assume that if Binary Ninja's HLIL classifies each function as a separate entity, it can detect where a function starts and ends. Table 1 summarizes the `function start` precision results.

Table 1: `Function Start` Precision

Precision	Function starts
Plugin	100%
Binary Ninja HLIL	100%

4.1.2 For Loop Precision

The plugin's heat maps correctly identify 13 of 17 `for loops`, resulting in 76.47% precision. The heat map failed to identify `for loops` in the `safe_write function`, the `rpl_mbrtowc` function, and the `setlocale_null_unlocked` function. The `safe-read.c` file labels the source code for the `safe_write` function as `safe_rw`. It contains a `for loop` that the authors wrote with no initialization, condition, or incrementation, which is likely why the heat map cannot detect it. Figure 12 displays this anomaly.

```
size_t
safe_rw (int fd, void const *buf, size_t count)
{
  for (;;)
    {
      ssize_t result = rw (fd, buf, count);

      if (0 <= result)
        return result;
      else if (IS_EINTR (errno))
        continue;
      else if (errno == EINVAL && SYS_BUFSIZE_MAX < count)
        count = SYS_BUFSIZE_MAX;
      else
        return result;
    }
}
```

Figure 12: Malformed `for loop` in the `safe_rw` function

The `for loop` inside the `rpl_mbrtowc` function is nested inside an `if statement` and contains no initialization, which may affect the heat maps' detection. Binary Ninja's HLIL representation of the `setlocale_null_unlocked` function does not align with its source code. The order of the instructions is misaligned, and the decompiled version is missing several other program features in the source code such as `if statements`. Therefore, the heat maps' inability to detect this program feature is likely due to the way Binary Ninja decompiled the executable. As noted before, Binary Ninja does not come with native support for the RISC-V architecture. Thus,

there is likely deficiencies in its ability to decompile some RISC-V code.

Binary Ninja misrepresents every `for loop` as a `while loop`, resulting in 0% `for loop` precision. These failures are likely due to how Binary Ninja represents loops, however they demonstrate that the heat map plugin may have an advantage for identifying some program features compared to Binary Ninja's HLIL. Table 2 summarizes the `for loop` precision results.

Table 2: `For Loop` Precision

Precision	**For Loops**
Plugin	76.47%
Binary Ninja HLIL	0%

4.1.3 Switch Statement Precision

The plugin correctly identifies 22 of 24 `switch statements`, resulting in 91.67% precision. The plugin failed to identify `switch statements` in the `setlocale_null_androidfix` function and the `setlocale_null` function. Both functions originate from the `setlocale_null.c` file. Similar to the `setlocale_null_unlocked` function, Binary Ninja's decompiled versions of these functions do not align with their source code. Both contain a few variable initializations and a call to another function, but they are missing a large portion of the instructions present in the source code. Figure 13 compares Binary Ninja's HLIL representation of `setlocale_null_unlocked` with its source code.

The heat maps' inability to identify `switch statements` within these two functions is likely due to Binary Ninja's decompilation rather than deficiencies in the Convolutional Neural Network (CNN) detection technique. Overall, the `setlocale_null.c` file alone contained approximately 67% of the program features the heat maps failed to detect.

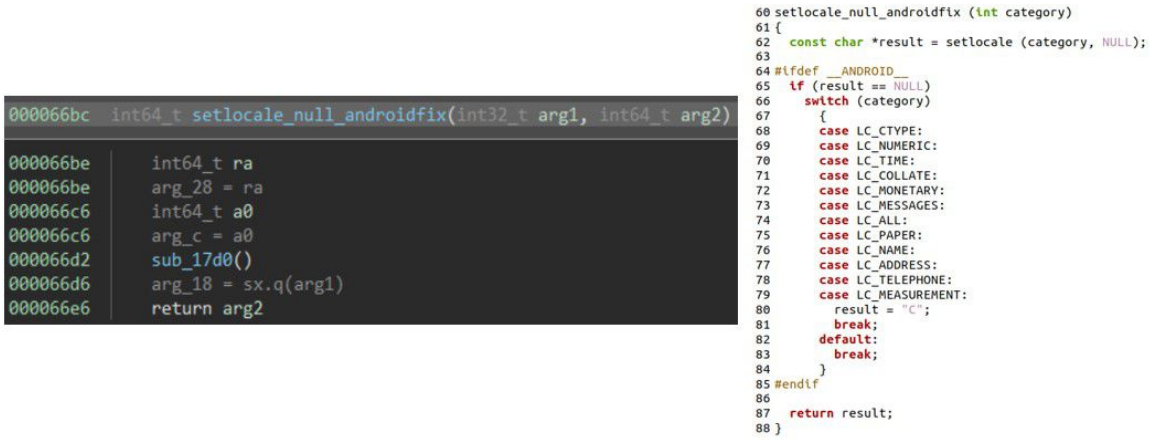

Figure 13: Comparison of Binary Ninja's HLIL (left) and source code (right)

Binary Ninja identifies 3 out of 24 `switch statements`, resulting in 12.5% precision. The HLIL misrepresents the rest of the `switch statements` as a series of nested `if statements`. Unlike `for loops`, Binary Ninja has shown the ability to detect `switch statements`. However, the heat map demonstrates a greater capacity to detect this feature within the `yes` executable. Table 3 summarizes the `switch statement` precision results.

Table 3: `Switch Statement` Precision

Precision	**Switch Statements**
Plugin	91.67%
Binary Ninja HLIL	12.5%

4.2 Results Summary

This chapter presents and analyzes program feature detection precisions from a custom heat map plugin and Binary Ninja's HLIL. Both the plugin and the HLIL demonstrate the ability to identify `function starts` with high precision. The plugin has relatively high precision for detecting `for loops` and `switch statements` when accounting for abnormal coding practices and decompilation issues. Con-

versely, Binary Ninja's HLIL performs poorly in representing `for loops` and `switch statements`.

V. Conclusions

The field of explainable artificial intelligence (AI) has emerged in response to the need for models capable of explaining themselves. Layerwise Relevance Propagation (LRP) is an explainability technique that can explain a machine learning model's decision in terms of its input. These explanations provide useful data to the model's developers and users. For reverse engineers, explainability information from LRP can have practical use as a reverse engineering (RE) aid. If developers design a machine learning model that can detect a certain pattern in a binary that a reverse engineer is looking for, information pointing to where the pattern occurs could be crucial for the reverse engineer's analysis. However, little research thus far has tested whether these hypotheses are true.

5.1 Research Conclusions

To fill this gap, this research develops a plugin for Binary Ninja that produces heat map visualizations of LRP relevance values. The plugin takes a serialized relevance dictionary as input, calculates color values for each instruction present in the dictionary, and then uses these colors values to highlight the instructions in Binary Ninja.

This research conducts an experiment to measure the amount of `function starts`, `for loops`, and `switch statements` that the heat maps and Binary Ninja's high level intermediate language (HLIL) can identify. We then produce precision statistics by dividing the true positive results for each tool by the number of program features present in the binary's source code.

Overall, the experimental results show that the plugin's heat maps can identify 100% of `function starts`, 76.47% of `for loops`, and 91.67% of `switch statements`

in the tested RISC-V binary. Binary Ninja's HLIL was also able to identify `function starts` with 100% precision. However, it could not represent any of the `for loops` and only detected 12.5% of the `switch statements`.

These statistics demonstrate that the plugin's heat maps can identify `for loops` and `switch statements` with higher precision than Binary Ninja's built-in decompiler. Based on these results, we deduce that the plugin can act as a RE aid because it maintains a superior ability to identify program features when compared to a popular RE platform.

5.2 Future Work

The following efforts are potential avenues for future work:

- **Advanced Program Feature Detection**: This plugin is a script that users can modify to read in any `.pickle` file. Therefore, researchers can use the script with other models designed to detect more advanced program features such as software vulnerabilities.

- **Graphical User Interface (GUI)**: The plugin only performs the basic functions outlined in Section 3.2. A GUI would allow users to select between multiple program features and different models without having to make edits to the plugin's code. This improvement would make the plugin more user-friendly and efficient.

- **Automated Analysis**: The experiment for this research effort required thorough manual analysis to identify program features present in the source code and to measure those identified by the heat map and Binary Ninja's HLIL. An analysis system that can automate these measurements would reduce the time required to evaluate the plugin.

- **User Study**: A user study would benefit this research because it would allow experts to provide feedback regarding the quality and usability of the heat map visualizations.

- **Platform Flexibility**: The plugin presented in this research only works within Binary Ninja. Each RE platform employs different techniques for disassembling and decompiling code. In addition, reverse engineers have preferences for which interfaces they prefer. Therefore, increasing platform diversity would increase the plugin's reach and usability.

5.3 Countermeasures and Limitations

Code obfuscation is a preventive measure many program authors take to hide attributes in their code. Both legitimate actors and malware authors may employ this measure to disrupt the RE process. No matter their intent, code obfuscation techniques will negatively affect the plugin's ability to heat map program features because they would prevent Binary Ninja from accurately disassembling and decompiling binary code.

As noted previously, the visualization tool is built for Binary Ninja. Experiments have not been performed to test how the plugin's heat maps perform in other platforms or how they compare with other popular decompilers.

5.4 Research Significance

This research serves as a stepping stone for other efforts to use machine learning explainability techniques as a RE aid. The experimental results demonstrate that heat maps visualizations of LRP results can detect low-level program features with relatively high precision. This capability signifies that reverse engineers can use these

heat maps as a visual means of directing their focus to features in the code that they are looking for.

In addition, this research demonstrates that these visualizations can perform better than Binary Ninja at representing certain features that exist in the source code. Therefore, the heat maps have the potential to significantly increase the efficiency of the RE process by reducing the time needed to recognize certain patterns in disassembled or decompiled code.

Appendix A. Software Listing

Binary Ninja Plugin - __init.py__

```
#author: Alec D. McGahee
#date: 15 September 2022

from binaryninja import *
import pickle
import math

def plugin_main(bv, function):

    #Uncomment pickle file based on desired feature

    #Function Starts
    relevances = pickle.load(open("C:\\Users\\Admin\\Documents\\
        yes_CNNModelv6_function_start.pickle", "rb"))

    #For Loops
    #relevances = pickle.load(open("C:\\Users\\Admin\\Documents\\
        yes_CNNModelv6_for.pickle", "rb"))

    #Switch Statements
    #relevances = pickle.load(open("C:\\Users\\Admin\\Documents\\
        yes_CNNModelv6_switch.pickle", "rb"))

    base = 0
3
    #iterate through dict to find highest relevance
    for x in relevances:
        if x == "name":
            continue
```

```
27          elif float(relevances[x]) > base:
28              base = relevances[x]
29
30              #save instruction with highest relevance
31              highRel = x
32
33      #Format address for instruction with highest relevance
34      highRel = "0x" + highRel
35
36      #Creates dictionary to hold RGB color values for instructions
            with positive relevances
37      heatmap = {}
38
39      #Calculate Color Values
40      for x in relevances:
41
42          #ignore first value
43          if x == "name":
44              continue
45
46          #throw away values less than 0
47          elif float(relevances[x]) < 0:
48              continue
49
50          #if value greater than zero, calculate rgb score and add to
                dict.
51          else:
52              rgbVal = (float(relevances[x])/base)*255
53              heatmap[x]=rgbVal
54
55      #Generate heat maps
56      for x in heatmap:
```

```
57
58          currentAddr=int(x,16)
59          previousFunction = bv.get_previous_function_start_before(
                currentAddr)
60          cfunc=bv.get_function_at(previousFunction)
61          print(cfunc)
62          print(math.ceil(heatmap[x]))
63          cfunc.set_user_instr_highlight(currentAddr, highlight.
                HighlightColor(red = math.ceil(heatmap[x]), blue = 0,
                green = 0))
64
65  #Binary Ninja API use
66  PluginCommand.register_for_address("Heat Map", "Displays color
        spectrum onto function (main for now)", plugin_main)
```

Bibliography

1. V. Iglovikov, S. Mushinskiy, and V. Osin, "Satellite imagery feature detection using deep convolutional neural network: A kaggle competition," *arXiv preprint arXiv:1706.06169*, 2017.

2. A. W. Thomas, H. R. Heekeren, K.-R. Müller, and W. Samek, "Analyzing neuroimaging data through recurrent deep learning models," *Frontiers in neuroscience*, p. 1321, 2019.

3. D. Gavriluţ, M. Cimpoeşu, D. Anton, and L. Ciortuz, "Malware detection using machine learning," in *2009 International Multiconference on Computer Science and Information Technology*. IEEE, 2009, pp. 735–741.

4. W. Samek, G. Montavon, S. Lapuschkin, C. J. Anders, and K.-R. Müller, "Explaining deep neural networks and beyond: A review of methods and applications," *Proceedings of the IEEE*, vol. 109, no. 3, pp. 247–278, 2021.

5. D. F. Koranek, S. R. Graham, B. J. Borghetti, and W. C. Henry, "Identification of return-oriented programming attacks using risc-v instruction trace data," *IEEE Access*, vol. 10, pp. 45 347–45 364, 2022.

6. I. Education, "What is artificial intelligence (ai)?" 2022. [Online]. Available: https://www.ibm.com/cloud/learn/what-is-artificial-intelligence?

7. M. Copeland, "The difference between ai, machine learning, and deep learning?" Jul 2016. [Online]. Available: https://blogs.nvidia.com/blog/2016/07/29/whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/

8. C. Parsons, "What's a machine learning model?" Aug 2021. [Online]. Available: https://blogs.nvidia.com/blog/2021/08/16/what-is-a-machine-learning-model/?msclkid=d3a8f346c65911eca0f08a36ddbbd27b

9. M. Sikorski and A. Honig, *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

10. P. Hall and N. Gill, *An introduction to machine learning interpretability*. O'Reilly Media, Incorporated, 2019.

11. D. Roselli, J. Matthews, and N. Talagala, "Managing bias in ai," in *Companion Proceedings of The 2019 World Wide Web Conference*, 2019, pp. 539–544.

12. S. Spreeuwenberg, *AIX: Artificial Intelligence Needs Explanation*. LibRT BV, 2019.

13. D. A. Broniatowski, "Psychological Foundations of Explainability and Interpretability in Artificial Intelligence," 2021. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8367.pdf

14. P. Linardatos, V. Papastefanopoulos, and S. Kotsiantis, "Explainable ai: A review of machine learning interpretability methods," *Entropy*, vol. 23, no. 1, p. 18, 2020.

15. S. Chakraborty, R. Tomsett, R. Raghavendra, D. Harborne, M. Alzantot, F. Cerutti, M. Srivastava, A. Preece, S. Julier, R. M. Rao *et al.*, "Interpretability of deep learning models: A survey of results," in *2017 IEEE smartworld, ubiquitous intelligence & computing, advanced & trusted computed, scalable computing & communications, cloud & big data computing, Internet of people and smart city innovation (smartworld/SCALCOM/UIC/ATC/CBDcom/IOP/SCI)*. IEEE, 2017, pp. 1–6.

16. Heatmapping. 2021. [Online]. Available: http://www.heatmapping.org

17. S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek, "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation," *PloS one*, vol. 10, no. 7, p. e0130140, 2015.

18. J. Grezmak, J. Zhang, P. Wang, K. A. Loparo, and R. X. Gao, "Interpretable convolutional neural network through layer-wise relevance propagation for machine fault diagnosis," *IEEE Sensors Journal*, vol. 20, no. 6, pp. 3172–3181, 2020.

19. L. K. Shar and H. B. K. Tan, "Predicting sql injection and cross site scripting vulnerabilities through mining input sanitization patterns," *Information and Software Technology*, vol. 55, no. 10, pp. 1767–1780, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584913000852

20. G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," ser. CODASPY '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 85–96. [Online]. Available: https://doi.org/10.1145/2857705.2857720

21. J. Tranter, "What is risc-v and why is it important?" May 2021. [Online]. Available: https://www.ics.com/blog/what-risc-v-and-why-it-important

22. E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE software*, vol. 7, no. 1, pp. 13–17, 1990.

23. A. R. Bryant, *Understanding how reverse engineers make sense of programs from assembly language representations.* Air Force Institute of Technology, 2012.

24. W. C. Henry and G. L. Peterson, "Sensorre: Provenance support for software reverse engineers," *Computers & Security*, vol. 95, p. 101865, 2020.

25. H. S. Anderson, A. Kharkar, B. Filar, and P. Roth, "Evading machine learning malware detection," *black Hat*, vol. 2017, 2017.

26. Introduction to decompilation vs. disassembly. [Online]. Available: https://www.hex-rays.com/decompiler/decompilation_vs_disassembly/

27. J. Baldwin, P. Sinha, M. Salois, and Y. Coady, "Progressive user interfaces for regressive analysis: making tracks with large, low-level systems," in *Proceedings of the Twelfth Australasian User Interface Conference-Volume 117*. Citeseer, 2011, pp. 47–56.

28. G. Canfora, M. Di Penta, and L. Cerulo, "Achievements and challenges in software reverse engineering," *Communications of the ACM*, vol. 54, no. 4, pp. 142–151, 2011.

29. A. S. Mutschler, "Using processor trace at the system level," Apr 2020. [Online]. Available: https://semiengineering.com/using-processor-trace-at-the-system-level/

30. "Perf: Linux profiling with performance counters." [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

31. J. Batra, "Collecting intel® processor trace (intel® pt) in intel® system debugger." [Online]. Available: https://www.intel.com/content/www/us/en/developer/videos/collecting-processor-trace-in-intel-system-debugger.html

32. "Coresignt architecture." [Online]. Available: https://developer.arm.com/Architectures/CoreSight%20Architecture

33. L. Chen, S. Sultana, and R. Sahita, "Henet: A deep learning approach on intel® processor trace for effective exploit detection," in *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2018, pp. 109–115.

34. D. Pfaff, S. Hack, and C. Hammer, "Learning how to prevent return-oriented programming efficiently," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2015, pp. 68–85.

35. J. Zhang, W. Chen, and Y. Niu, "Deepcheck: A non-intrusive control-flow integrity checking based on deep learning," *arXiv preprint arXiv:1905.01858*, 2019.

36. J. Cowley, "Job analysis results for malicious-code reverse engineers: A case study," CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, Tech. Rep., 2014.

37. M. K. Tennor, "Reverse engineering cognition," MITRE CORP MCLEAN VA, Tech. Rep., 2015.

38. "Ghidra." [Online]. Available: https://github.com/NationalSecurityAgency/ghidra

39. "Ida pro." [Online]. Available: https://hex-rays.com/ida-pro/

40. "Binary ninja." [Online]. Available: https://binary.ninja/

41. R. Koschke, "Software visualization in software maintenance, reverse engineering, and re-engineering: A research survey," *Journal of Software Maintenance and Evolution*, vol. 15, no. 2, pp. 87–109, 2003.

42. "Heatmap." [Online]. Available: https://www.optimizely.com/optimization-glossary/heatmap/

43. Y.-J. Jung, S.-H. Han, and H.-J. Choi, "Explaining cnn and rnn using selective layer-wise relevance propagation," *IEEE Access*, vol. 9, pp. 18 670–18 681, 2021.

44. A. Binder, S. Bach, G. Montavon, K.-R. Müller, and W. Samek, "Layer-wise relevance propagation for deep neural network architectures," in *Information science and applications (ICISA) 2016*. Springer, 2016, pp. 913–922.

45. M. Böhle, F. Eitel, M. Weygandt, and K. Ritter, "Layer-wise relevance propagation for explaining deep neural network decisions in mri-based alzheimer's disease classification," *Frontiers in aging neuroscience*, p. 194, 2019.

46. Bn-riscv: Binary Ninja Plugin for RISC-V. 2020. [Online]. Available: https://github.com/uni-due-syssec/bn-riscv

Acronyms

AI artificial intelligence. 3, 4, 31

API application programming interface. 11, 14–17

CNN Convolutional Neural Network. 22, 28

CPU central processing unit. 10, 22

CUT Component Under Test. 20

GUI Graphical User Interface. 32

HLIL high level intermediate language. 2, 3, 20, 21, 23, 25–32

LRP Layerwise Relevance Propagation. vii, 1, 2, 6, 12, 17, 20, 31, 33

RAM random access memory. 22

RE reverse engineering. 1–4, 6, 8–12, 14, 16, 20, 31–34

RGB Red-Green-Blue. 17, 18

SUT System Under Test. 20, 21

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
15–09–2022	Master's Capstone	Sept 2021 — Sept 2022

4. TITLE AND SUBTITLE

Visualizing Machine Learning Explanations
to Support Software Reverse Engineers

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

McGahee, Alec, 2nd Lt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MAS-22-S-031

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Software reverse engineering is a complex and time-consuming process. Machine learning is a tool that has potential to increase the efficiency of this process. In particular, machine learning explanations allow models to explain themselves in a way that can provide beneficial information to reverse engineers. This research develops a novel visualization tool that can heat map machine learning explanations within Binary Ninja. We determine how precisely the tool can identify function starts, for loops, and switch statements present in a compiled binary's source code. We also collect precision metrics from Binary Ninja's high-level intermediate language and compare these with the tool's. Overall, the experimental results show the tool and Binary Ninja can identify functions starts with 100% accuracy. The tool was able detect for loops with 76.47% accuracy and switch statements with 91.67%. Meanwhile, Binary Ninja's high-level intermediate view detected 0% of the for loops and only 12.5% of the switch statements.

15. SUBJECT TERMS

software reverse engineering (RE), convolutional neural network (CNN), explainable artificial intelligence (AI), visualizations, machine learning, deep learning, Layerwise Relevance Propagation (LRP), heat maps

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Lt Col Wayne Henry, AFIT/ENG
U	U	U	UU	54	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x7243; wayne.henry@afit.edu