3-2023

Improving Accessibility and Efficiency of Analytic Provenance Tools for Reverse Engineering

Caleb W. Richardson

IMPROVING ACCESSIBILITY AND EFFICIENCY OF ANALYTIC PROVENANCE TOOLS FOR REVERSE ENGINEERING

THESIS

Caleb W. Richardson, 2nd Lieutenant, USAF

AFIT-ENG-MS-23-M-054

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-23-M-054

IMPROVING ACCESSIBILITY AND EFFICIENCY OF ANALYTIC PROVENANCE TOOLS FOR REVERSE ENGINEERING

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

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 Science in Computer Science

Caleb W. Richardson, B.S.C.S.

2nd Lieutenant, USAF

March 2023

AFIT-ENG-MS-23-M-054

IMPROVING ACCESSIBILITY AND EFFICIENCY OF ANALYTIC PROVENANCE TOOLS FOR REVERSE ENGINEERING

THESIS

Caleb W. Richardson, B.S.C.S.
2nd Lieutenant, USAF

Committee Membership:

Lt Col Wayne C. Henry, Ph.D.
Chair

Dr. Gilbert Peterson, Ph.D.
Member

Dr. Robert Mills, Ph.D.
Member

AFIT-ENG-MS-23-M-054

Abstract

Reverse engineering is a vital technique for identifying and mitigating cyber threats. Yet, despite its importance, reverse engineering is a time-consuming process. Provenance tools help to improve the workflow of reverse engineers by providing an accessible method of viewing their flow through a binary. The current state-of-the-art provenance tool for reverse engineering software called SensorRE, leverages an external server, web browser, and a large array of javascript libraries.

This thesis presents Provenance Ninja, a software reverse engineering tool developed in Python that runs directly within Binary Ninja. Provenance Ninja captures reverse engineers' provenance data and provides an interactive graph within the reverse engineering environment. The performance of Provenance Ninja is evaluated against SensorRE by measuring functionality and efficiency. This research demonstrates that it is possible to design a provenance tool to run natively in the reverse engineering software that passes all functionality tests when compared to SensorRE and shows statistically significant efficiency improvements at a 95% confidence level in memory utilization and runtime from this approach. The results of this study contribute to the field of software reverse engineering and have the potential to enhance the effectiveness of cyber threat mitigation efforts.

AFIT-ENG-MS-23-M-054

To my parents who inspire me to work harder every single day and become the best possible version of myself.

Table of Contents

Page

Abstract iv

I. Introduction 1

1.1 Problem Background 1
1.2 Problem Statement 2
1.3 Approach 3
1.4 Assumptions 4
1.5 Document Overview 4

II. Background and Literature Review 6

2.1 Overview 6
2.2 Sensemaking 6
2.3 Software Reverse Engineering 7
2.3.1 Static Analysis 8
2.3.2 Dynamic Analysis 9
2.3.3 Sensemaking in Reverse Engineering 9
2.4 Visualization in Analytics 10
2.5 Designing Tools and Evaluations around Requirements 13
2.5.1 Qualitative Evaluations 13
2.5.2 Quantitative Evaluations 14
2.5.3 Evaluation Tools 16
2.6 Provenance 16
2.7 Implementation Technologies 17
2.8 Background Summary 18

III. Design and Methodology 19

3.1 Overview 19
3.2 System Summary 19
3.3 System Description 20
3.3.1 Constructing the Provenance Graph 21
3.3.2 Capturing Input 22
3.3.3 Undo 23
3.3.4 Verbose Mode and Collapsing the Graph 24
3.4 Tools and Implementation Requirements 25
3.5 Usage Scenario 25
3.6 Experiment Hypothesis 29
3.7 Functionalitity Testing Setup 31
3.7.1 Function Test 32
3.7.2 Variable and Type Test 32

Page

3.7.3 Comments and Highlighting Test . . . 33
3.8 Experimental Design Setup . . . 34
3.8.1 Response Time . . . 34
3.8.2 Memory Usage . . . 35
3.8.3 Choice of Statistical Test . . . 35
3.9 Ensuring Isolation . . . 35
3.10 Randomization and Threats to Validity . . . 36
3.11 Methodology Summary . . . 37

IV. Results and Analysis . . . 38

4.1 Overview . . . 38
4.2 Functionality Testing . . . 38
4.3 Performace Testing . . . 39
4.3.1 Gathering Data . . . 40
4.3.2 Memory Analysis Results and Observations . . . 42
4.3.3 Runtime Analysis Results and Observations . . . 45
4.4 Results Summary . . . 47

V. Conclusions . . . 48

5.1 Research Contributions . . . 48
5.2 Research Limitations . . . 49
5.3 Future Work . . . 49
5.4 Concluding Thoughts . . . 50

Appendix A. Software Listings . . . 52

Bibliography . . . 83

IMPROVING ACCESSIBILITY AND EFFICIENCY OF ANALYTIC PROVENANCE TOOLS FOR REVERSE ENGINEERING

I. Introduction

Reverse engineering is a crucial technique for identifying and mitigating cyber threats. It involves analyzing the structure and function of software and hardware systems to understand how they work and identify vulnerabilities. The ability to understand and exploit the inner workings of a system is essential for identifying and addressing cyber threats. The importance of reverse engineering in cybersecurity is recognized at the national level, as evidenced by legislation such as the Cybersecurity Information Sharing Act (CISA) and executive orders such as Executive Order 13636, which emphasizes the need for increased cybersecurity in critical infrastructure [1, 2, 3]. Through reverse engineering, organizations and governments can better protect themselves and the nation from foreign cyber threats by identifying and addressing vulnerabilities in software and systems.

1.1 Problem Background

Reverse engineering is the process of analyzing and understanding the design, structure, and functionality of a given software program. This process is often used to identify and repair vulnerabilities, improve performance, or extract proprietary information from a program. However, this process can be time-consuming and challenging, particularly for large and complex software programs.

One of the key challenges in reverse engineering is understanding the flow of data and interactions within a binary. Analytic provenance is a technique that addresses

this challenge by providing an accessible method of viewing the flow of data and interactions during analysis. However, despite the growing recognition of their importance, there are still limitations in the current state-of-the-art tools.

A systematic study by Perez, et al. [4] found that computational overhead and integration are significant open problems for provenance tools. Classical provenance data techniques and methods are usually data-intensive and time-consuming [5, 6]. While computational overhead in a reverse engineering context is not often considered a limitation, minimizing collection overhead during workflow execution could have a significant impact across a large user base. Furthermore, the reliance on external dependencies can make the tools less accessible, particularly to those working in restricted environments or with limited internet connectivity. These issues highlight the need for a more efficient and accessible solution that can improve the workflow effectiveness of reverse engineers.

SensorRE was developed to support the provenance needs of software engineers during binary analysis [7]. While SensorRE was the first provenance tool for reverse engineers, it is reliant on a multitude of javascript libraries, an external browser, and a server in order to operate with the reverse engineering tool Binary Ninja. This research addresses these limitations.

1.2 Problem Statement

This research addresses the limitations of current state-of-the-art analytic provenance tools in the context of reverse engineering. Specifically, this research develops a provenance plugin that runs directly within Binary Ninja, a popular reverse engineering tool, and evaluates its performance and effectiveness in comparison to SensorRE, an analytic provenance reverse engineering tool. The objectives of this research are to minimize collection overhead during workflow execution, increase accessibility, and

reduce dependency on external servers and web browsers. This research divides its core task into two primary objectives.

1. To design and develop a fully featured reverse engineering provenance tool as a Binary Ninja plugin, Provenance Ninja.

2. To evaluate the performance of Provenance Ninja in terms of memory efficiency and speed compared to SensorRE.

The first part involves the development of the plugin itself. Provenance Ninja is the first provenance tool developed natively within Binary Ninja. This bears the unique challenge of determining how to best display a provenance graph in this environment. After that, the next objective is to determine if this plugin is more efficient than SensorRE. Memory efficiency is measured by recording which programs have their memory affected by each plugin and observing whether Provenance Ninja improves upon the benchmarks of SensorRE. Speed is tracked through the runtime of the plugin. This metric relates to how long it takes from when an action is taken to when it shows up on the graph.

1.3 Approach

The approach of this research is divided into two main phases: development and evaluation. In the development phase, the primary objective is to design and implement a fully-featured reverse engineering provenance tool as a Binary Ninja plugin. The first step in this process is to develop a proof of concept, demonstrating that a more extensive provenance tool can be accomplished. This proof of concept is used to guide the development of the full-fledged plugin. A core design decision during development is the focus on keeping as many of the features present in SensorRE as possible. At the same time, the plan is to expand and evolve those features, utilizing

the rich features in the latest Binary Ninja API.

In the evaluation phase, the newly developed plugin is compared against SensorRE to gauge improvements in memory utilization and response times. This is done by comparing the performance of the two tools on a set of tests that simulate reverse engineering tasks. This experiment assesses how much more efficient and faster Provenance Ninja is compared to SensorRE and provides insight into the gained improvement from developing a reverse engineering provenance tool as a Binary Ninja plugin.

1.4 Assumptions

In this research effort, it is assumed that the Binary Ninja application is the primary reverse engineering tool of interest for developing and testing the provenance plugin. This assumption is based on the existing provenance tool, SensorRE, which was designed for Binary Ninja. Thus, utilizing the same reverse engineering software simplifies the testing process and reduces the potential for confounding variables. Additionally, it is assumed that the primary performance metrics of interest for evaluating the plugin are memory utilization and response time. These metrics were chosen for their straightforward method of measurement, reducing the potential for confounding variables that could complicate the testing data.

1.5 Document Overview

This chapter presents the research problem, objectives, and approach for this thesis. Chapter 2 provides an overview of reverse engineering, provenance, and how the two work together to improve the workflow of reverse engineers. Chapter 3 describes the methodology of how Provenance Ninja is designed and outlines how it is evaluated against SensorRE. Chapter 4 presents the result of the evaluation and analyzes the

results in terms of accessibility and efficiency. Finally, Chapter 5 summarizes the research contributions and provides potential future avenues for research.

II. Background and Literature Review

2.1 Overview

This chapter provides a comprehensive overview of the key concepts and technologies that form the foundation of this research. The chapter begins by discussing the importance of sensemaking in software reverse engineering, followed by an in-depth analysis of the key concepts and techniques used in software reverse engineering. Next, the chapter describes various visualization techniques that are commonly used in analytics and the design of tools for evaluating them. The chapter then provides a detailed overview of the concepts of provenance and the implementation technologies used in this research.

2.2 Sensemaking

Everyone has a different way of processing and understanding information. One person may look at visualization and interpret it slightly or entirely differently than the next person. Understanding sensemaking is the key to ensuring that as many people as possible can extract the core meaning of a medium.

Sensemaking can be defined as "a dynamic process of building or revising an explanation in order to 'figure something out' to ascertain the mechanism underlying a phenomenon in order to resolve a gap or inconsistency in one's understanding" [8]. It is how we build a bridge between what we don't know by using what we already know. This process is described by Pirolli and Card as one that can either occur top-down, or bottom-up [9]. In the case of visualization, the bottom-up sensemaking process is used because a viewer is extrapolating a theory from presented data.

This process has five steps. First, a viewer searches and filters the data they are presented with. Then they read that filtered content and extract evidence and

inferences. Next, a person classifies the data into some sort of schematic, whether it be a mental model or a computer-generated graph. Based on that graph, they then construct a theory that affirms or dissuades their current hypothesis. Finally, they utilize this whole process to present a story (to a client, for example) about what they have concluded [9].

2.3 Software Reverse Engineering

Software reverse engineering is a field in which engineers deconstruct a program to discover how it works. Members of this field are dedicated to "understanding how the program uses the system interface, understanding the program's functions and instruction-level information, and how the program uses data" [10]. Whereas, when a developer compiles a program, they aim for it to work without the user ever questioning the behind-the-scenes code of the software. Engineers utilize reversing techniques to accomplish everything from vulnerability detection to malware detection, and neutralization to protecting intellectual property [7].

Traditionally engineers approach the reversing process in two phases: information extraction and abstraction [11]. However, more recently, the process has been broken down into four steps: asking questions, simulating answers to those questions, recognizing new information based on the simulations, then deciding where to further investigate the source code based on that information [12]. First, reverse engineers ask questions to understand how a program operates and hypothesize about the program's core functionality. This phase is conducted at a high level of abstraction. Then they simulate the software by running the program, using static and dynamic analysis, or combining both. Next, a user extracts information based on observations during the analysis phase. Finally, based on that information, they perform more static/and or dynamic analysis as necessary. Throughout these steps, the core

techniques used are static and dynamic analysis.

2.3.1 Static Analysis

Static analysis involves observing program code structure to determine its function [13]. This process consists of viewing information about a program without actually running it. Engineers commonly utilize hex editors, decompilers, and disassemblers during this phase [7]. While static analysis is a powerful tool, it can potentially be less effective against more complex programs and miss important functionality [13].

Commonly used static analysis tools include IDA Pro [14], Binary Ninja [15], and Ghidra [16]. All three of these allow a reverse engineer to analyze the assembly code of the program they are deconstructing in an easy-to-read manner. Below is an example of Binary Ninja in Figure 1. These tools also support plugins that allow developers to enhance the effectiveness of reverse engineers.

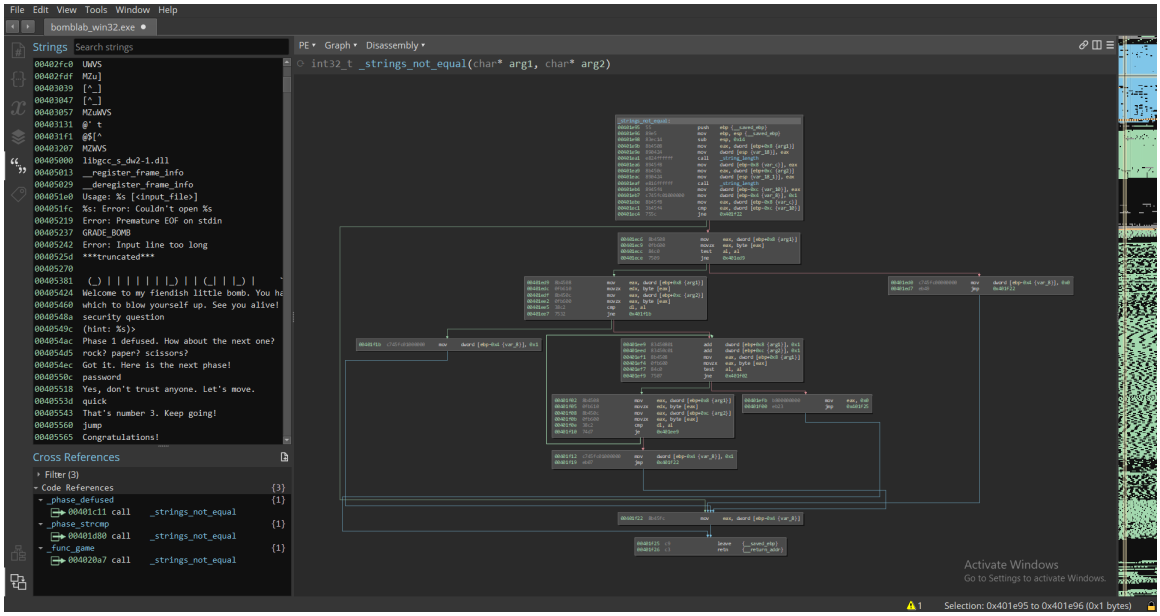


Figure 1: A model reverse engineering workspace within Binary Ninja

2.3.2 Dynamic Analysis

Dynamic analysis is predicated on observing the behavior of a program during its runtime [13]. An engineer can obtain even more information about a program using this analysis. In addition to simply running the program, dynamic instrumentation tools and vitalized environments aid engineers in dynamically analyzing software [7]. Together static and dynamic analysis are powerful methods of determining the functionality of a program.

Tools such as x32dbg [17] and OllyDbg [18] are the most commonly used for dynamic analysis, as well as the debugger built into IDA Pro. This category of tools grants the reverse engineer the capability to step through a program during its runtime and view how various registers within memory are affected by a program's function. Much like static analysis tools, these debuggers also support plugins.

2.3.3 Sensemaking in Reverse Engineering

In the realm of reverse engineering specifically, sensemaking is "goal-directed planning" to generate a hypothesis and seek information related to that hypothesis within the context of analyzing software [19]. This type of sensemaking follows a similar process as shown in Figure 2.

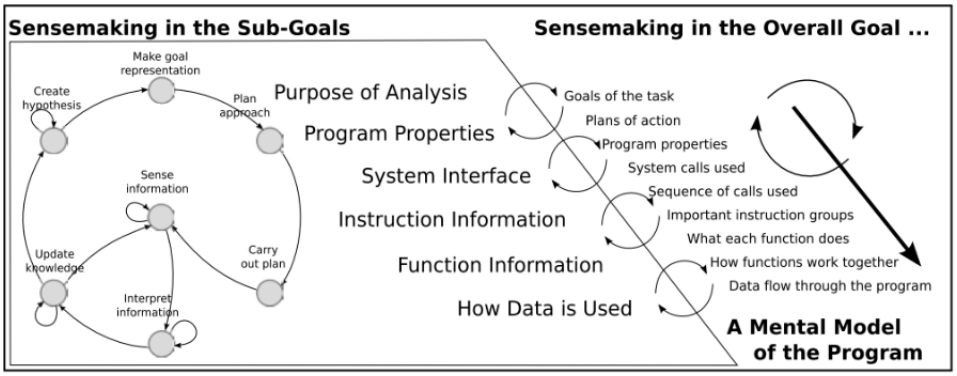


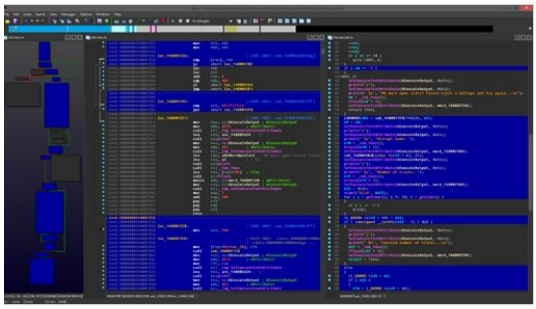
Figure 2: Sensemaking Steps in a Reverse Engineering Environment [20].

Approaching the overall goal, a reverse engineer's mission is to develop a mental model of the program they are analyzing. To begin, they must picture the end goal they are seeking and create sub-goals that will aid them along that path. Once those are established, an engineer develops plans for achieving each sub-goal. These goals come together to give information about the program's properties, system calls, sequence of calls used, important instruction groups, what each function does, and how functions work together. The combination of these results helps the reverse engineer learn how data flows through the program. This knowledge develops into a mental model of the program.

In completing each subgoal, a reverse engineer will often repeat the cycle shown to the left of Figure 2. They first construct a hypothesis based on the current information available. Based upon that, they then construct a goal representation of testing that hypothesis, plan its approach, and carry out the plan. Executing the plan involves sensing and interpreting information and updating their knowledge accordingly. Once new knowledge enters the picture, the cycle continues with the reverse engineer creating new hypotheses based upon their continual knowledge gain until the goal has been met.

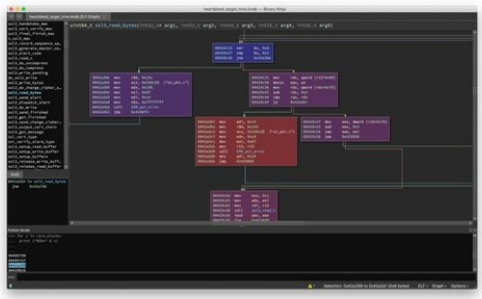
2.4 Visualization in Analytics

Within both the realms of static and dynamic analysis, reverse engineers have tools that aid them in effectively and efficiently analyzing software. These tools include various amounts of detail and visual aids that combine to provide a holistic view of a program's inner workings [7]. Within these tools exist many ways of visualizing the program the engineer is analyzing. These tools are highlighted below in Figure 3.

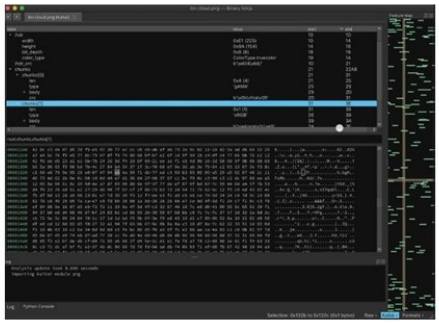


Lighthouse

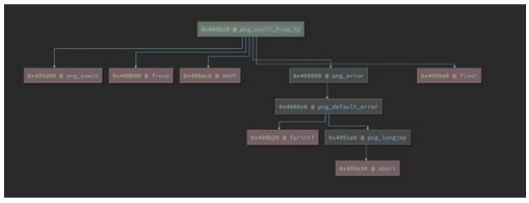
Sourcery Pane



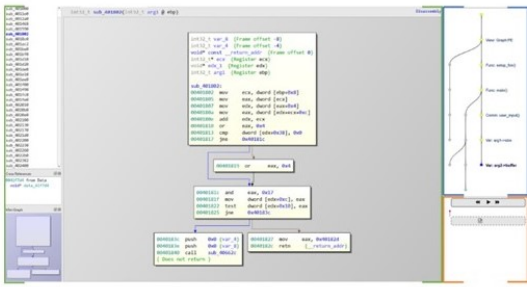
Bncovis



Katai



BNCallgraph



SensorRE

Figure 3: Visualization Plugins for Reverse Engineering Tools [21, 22, 23, 24, 25, 7]

Gaasdelen developed *Lighthouse* as a plugin for IDA Pro and Binary Ninja. This tool allows reverse engineer execution maps for programs [21]. Its coverage painting feature enables a view of coverage data across the disassembly and graph views on both IDA Pro and Binary Ninja, and the decompiler view for IDA Pro [21]. A widget

that docks to the engineer's view provides this visualization once it loads the coverage [21].

Sourcery Pane, a tool designed for Binary Ninja by a developer who goes by mechanicalnull, enables a reverse engineer to have a synchronized view of their disassembly and the source code if the source is available [22]. *Sourcery Pane* was created to visually clarify how source code links to the disassembled code.

Bncovis a scriptable interface developed for Binary Ninja, by the company ForAllSecure, provides coverage information [23]. With this tool, a reverse engineer can view a color-coded map that displays which code sections are covered by the most traces. This allows a reverse engineer to determine a program's functionality efficiently. Additionally, *Bncovis* allows scripting so users can directly interact with the plugin's data structures [23].

Katai provides a visualization tool for hexdumps where a reverse engineer can more easily dissect a binary file. Developed for Binary Ninja by Vector35, this plugin enables a reverse engineer to have a tree view of a program's hex dump [24].

Developer Borzacchiello designed *BNCallgraph* for Binary Ninja as a simple way to view the callgraph of a function [25]. This tool can construct both callgraphs and reverse callgraphs, enabling a reverse engineer to follow the flow of the function they are analyzing and its references. Furthermore, *BNCallgraph* colorizes the branches of the graph to enhance its interpretation.

Finally, *SensorRE* is a plugin for Binary Ninja supporting analytic provenance. That displays a user's provenance history and allows them to recall, replicate and recover actions that they have taken [7]. This tool also uniquely enables reverse engineers to reproduce and present their findings through its storyboard feature.

2.5 Designing Tools and Evaluations around Requirements

There are several methods of evaluating tools that help reverse engineers. Research has been conducted on how to best aid reverse engineers in their analytics, which fundamentally drives the reasoning for developing tools and plugins to augment those tools. Researchers have used both qualitative and quantitative methods in evaluations.

2.5.1 Qualitative Evaluations

One such study by Truede, et al. [26] breaks down the fundamental processes in reverse engineering and the challenges that reverse engineers have in each of those processes. This evaluation was conducted by interviewing engineers and gathering data from a group of structured questions. These questions allowed the researchers to gather data about the most useful tools for reverse engineers and the largest hurdles in utilizing those tools to create a more efficient work environment [26].

Baldwin, et al. conducted a case study on reverse engineers approaching the problem from a psychological frame of reference [27]. They then use the case study results to rank what they determine to be the premier problems surrounding reverse engineers. For this study, questions were asked to two different groups to elicit a holistic survey. The answers from each group were then given a specific point value that was used to order the most important requirements for reverse engineers [27].

Another study by Pirolli and Card interviewed intelligence analysts to find out how they organize information [9]. These findings gave critical insight into engineers' schemes during their analysis. From there, they delved into improving analytics production based on the engineers' responses [9].

Henry interviewed reverse engineers to determine how their current tools could be specifically amplified to aid them in their work. The areas that were focused on

in this study were metadata analysis, disassembly analysis, dynamic analysis, documentation, and collaboration. This study showed the primary needs which needed to be met in developing a provenance tool for binary analysis [7].

2.5.2 Quantitative Evaluations

While other evaluations focus more on quantitative analysis, Maletic proposes the notion of specific tasks that visualization must meet, as shown in Figure 4 [28].

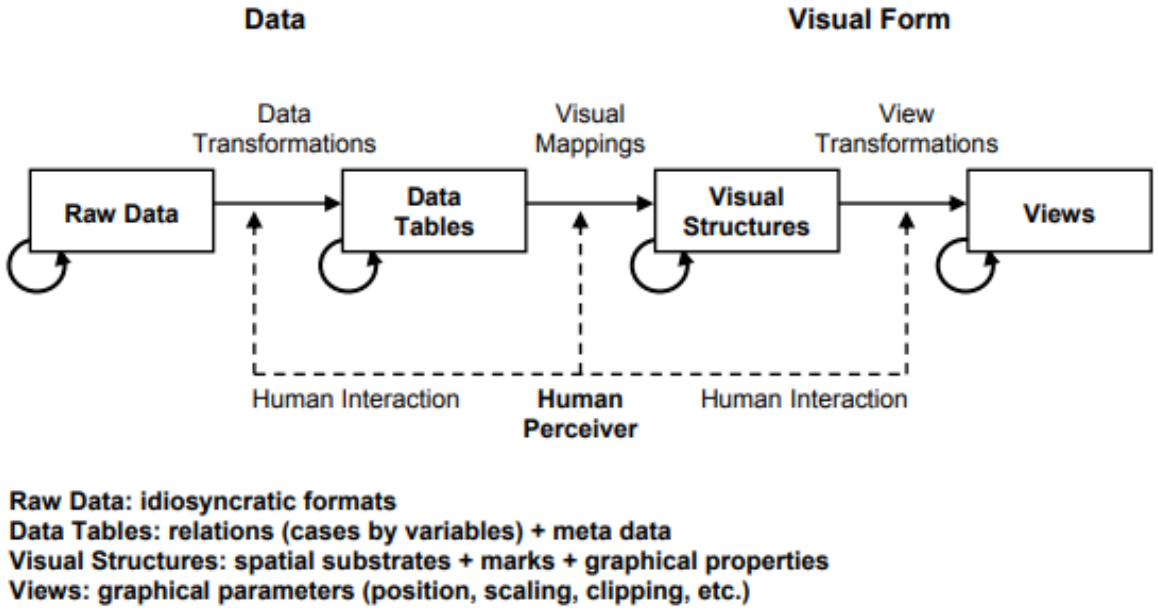


Figure 4: A Reference Model for Visualization [28]

All visualization efforts begin with raw data. This data is then transformed into data tables. Because these tables may not be straightforward for a human to perceive a visualization, they serve to map that data into a visual structure. This is the first requirement of a visualization. Additionally, it must also provide a method for interacting with that data. When visualization is broken down into certain conditions it must meet, it opens the possibility of quantitatively testing them through developing a series of benchmarks [28]. While Maletic does not offer a specific set of benchmarks

for all visualizations, they mention that these criteria will vary between each visual mapping based on how it structures its given data.

Globus and Uselton approach the evaluation of visualization software quantitatively at a high-level [29]. They propose different possible methods of evaluating visualization software beyond interviewing human subjects. These metrics include accuracy, scalability, completeness, its ability to be distributed, size efficiency, time efficiency, and currency. Yet, they also note that the end goal of any visualization is to improve the user experience and that this may require some qualitative analysis to thoroughly test efficacy [29].

Gazis and Katsiri have recently proposed a method to evaluate visualizations based on performance metrics [30]. These benchmarks assess CPU performance through user, system, CPU, and wall time and memory availability during the visualization's runtime. For this evaluation, the tests were performed on a web application and were compared on different frameworks [30].

In reverse engineering, Kienle and Muller evaluate what reversing tools are needed for static and dynamic analysis using quantitative methods [31, 32]. Another study identifies the quality attributes to measure for a software visualization as scalability, interoperability, customizability, interactivity, usability, and adaptability [31]. They then apply these metrics to reverse engineering tools by thoroughly defining what each of those means in the context of a reversing program. This study acknowledges that the evaluation of a tool through human studies is also an essential part of tool development, but not until the benchmarks have first been evaluated in the tool's development [32].

2.5.3 Evaluation Tools

Evaluating the metrics presented by Gazis can be executed using various tools. User and system CPU values and memory efficiency can be measured utilizing the built-in Windows Task Manager, Process Explorer [33], or Performance Monitor [34]. These tools can provide detailed information on the CPU and memory usage of specific processes, including the private working set, virtual memory size, and page file usage. Another way to compare memory efficiency is to use a specialized memory profiler tool such as memprofiler [35]. Memprofiler tracks memory usage, memory leaks, and CPU usage over time. Through this, you can measure and compare the amount of memory used by each application during specific tasks or operations and track changes in memory usage over time. One could also track the number of page faults that occurred to get an idea of the efficiency of memory management.

Wall time, also called runtime, can be measured by leveraging the system clock to measure the time between two actions occurring. This can be accessed from within the code in a given system and printed to the console. The system clock time can be printed before and after a given action, allowing the tester to calculate the total runtime manually. Alternatively, calculations can occur within the code and print the total runtime to the console.

2.6 Provenance

When presenting the model that an analyst has reached through sensemaking, it is often helpful to be able to detail how they reached that model. This is where provenance comes into play. Provenance is the history of a given object and, in a scientific sense, is "the sequence of steps that led to a result" [36]. In the realm of computing, provenance is "a record of past execution (or current execution)" that leads to a given piece or set of data [37].

There are six key reasons why an analyst might desire provenance data as outlined by Ragan, et al. [38]. First, it is essential for recall so that an analyst can look back on work done and remember how they reached their conclusions. Replication is also necessary, as engineers might need to recreate work they have previously done to verify their findings. Action recovery involves being able to undo or redo steps taken along the process, and this is another practical reason to have provenance data. An analyst may also desire this information if they collaborate with others and need a straightforward manner of guiding someone else through the same sensemaking process. Taking that one step further, if they are presenting this data in any way, being able to display provenance information aids their audience in understanding the analyst's thought process. Finally, this data is also helpful for meta-analysis of various trends that might show themselves throughout an engineer's work, allowing them to optimize their efforts [38].

Provenance data can be sorted into two categories. Data provenance relates to how raw data changes and moves [38]. This is most commonly used in fields with large amounts of data to peruse and analyze. Analytic provenance centers around exploring that data and deriving conclusions about what the data represents [7]. Visualizations of reverse engineering data most closely align with this category.

2.7 Implementation Technologies

There are many methods through which analytic provenance data could be incorporated into the reverse engineering process. Binary Ninja is the platform that has been previously targeted for the implementation of this data into the disassembly phase. This platform was chosen due to its rich application programming interface (API) support and its extensive library of external plugins, as highlighted previously.

SensorRE was the first application to support provenance for reverse engineering

[7]. This program did so through a web application platform attached to the Binary Ninja disassembly interface. This enabled scalable vector graphics (SVG) to create more dynamic graph features than would be available in a stand-alone graphing environment.

Another option for implementing such a plugin is utilizing the PySide6 library to create a graph directly within Binary Ninja [39]. This implementation would allow a user to run a provenance tool while using Binary Ninja without having a separate server or web browser running. This should decrease the memory load on the system. Additionally, because inputs do not have to travel through an external server, the response time for updating nodes on the graph should be faster.

2.8 Background Summary

This chapter overviewed software reverse engineering and its most commonly used tools. It also introduced sensemaking and provenance as methods of interpreting the analytic data gathered through reverse engineering efforts and presenting that information to others. Finally, it explored implementation options for efficiently displaying that data for the analyst.

III. Design and Methodology

3.1 Overview

Provenance tools are helpful for reverse engineers in their day-to-day tasks. Previous research focused on designing an external plugin to support this need. While this solution works well, this research explores a potentially more efficient method of designing a provenance system, one built directly into the Binary Ninja interface.

This chapter introduces Provenance Ninja, the first analytic provenance tool designed directly within Binary Ninja. Provenance Ninja displays a provenance graph in a snappable pane within the Binary Ninja environment. The design and implementation of the plugin are described, followed by a usage scenario. Finally, the means of testing and evaluating the system are presented.

3.2 System Summary

Provenance Ninja is designed to keep the features of SensorRE without requiring a web browser or internet server. Since Binary Ninja only supports Python plugins, the code has to be written solely in Python.

This plugin captures user input in the Binary Ninja interface and then displays that information on a provenance graph. The graph is a widget that can be moved around and scaled within the user's workspace. Aside from running the plugin at the beginning of their session, this tool causes little to no additional work on the reverse engineer's part as it does all of the capturing and displaying automatically. The system diagram for Provenance Ninja can be seen in Figure 5.

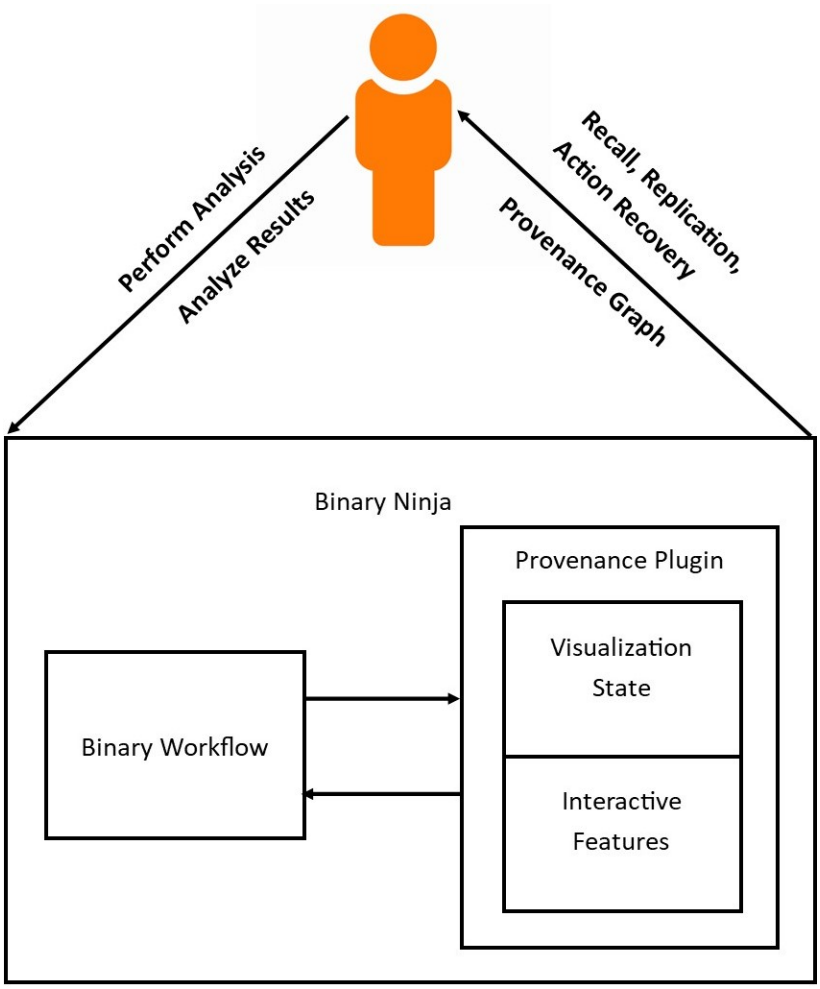


Figure 5: Provenance Ninja System Diagram

3.3 System Description

Provenance Ninja is a Python-based analytic provenance tool. It enables real-time viewing of the changes made to a binary within the Binary Ninja environment. The design of this system is largely modeled after SensorRE, beginning with its core features: provenance display and capture. Each poses unique challenges due to designing the plugin entirely in Python without a JavaScript module.

3.3.1 Constructing the Provenance Graph

The visualization utilizes a tree graph constructed using the PySide6 library [39], where nodes represent actions taken by the user. This Python library provided functionality for several core features. PySide's rich interaction with the native Binary Ninja dock handler is why this library was selected for the plugin. The PySide6 QWidget library is utilized to create a separate window in the user interface in which the plugin resides.

Within this window, a section is added for a graph where users can view their provenance changes in real time. Because of the inherent modularity within the PySide6 widgets, this allows for selection from a wide variety of graphing libraries for the actual graph itself. One such library is the FlowGraph library, which is directly within the Binary Ninja API [15]. To construct the graph itself, first, an array is created to store the nodes. When the plugin calls for the graph to be updated, it adds all the nodes in this array to the graph, regardless of which branch contains which node. Branches are then defined by the edges drawn between each node. A loop ran through each branch and added its corresponding edges to the graph. Finally, the graph is displayed for the user.

Specific nodes and lines are stylized differently to make the graph simpler for users to view. The current node is always green. Because commenting is one of the most commonly utilized tools when analyzing a binary, they are also distinguished. Comments along the active branch, excluding the current node, are colored blue to be easily identifiable. For a user to quickly recognize which actions represent the current state of the binary, the active branch is indicated with solid green arrows, and previous branches use dotted white arrows.

3.3.2 Capturing Input

Nodes on the aforementioned graph are populated by user input. Actions the user conducts within Binary Ninja are tracked and subsequently displayed. Upon receiving a notification that a change has been made within the Binary Ninja environment, the plugin filters the change by multiple types of actions. The actions that trigger a node in the graph are:

- Data Written
- Function Added
- Function Removed
- Function Updated
- Variable Added
- Variable Removed
- Type Defined
- Type Undefined

Once the appropriate input type is determined, a node representing that input is sent to the graph, and the graph is updated. An example of such a graph after a user has added two comments can be seen in Figure 6. As mentioned before, the current node is green, and the previously made comment node is blue.

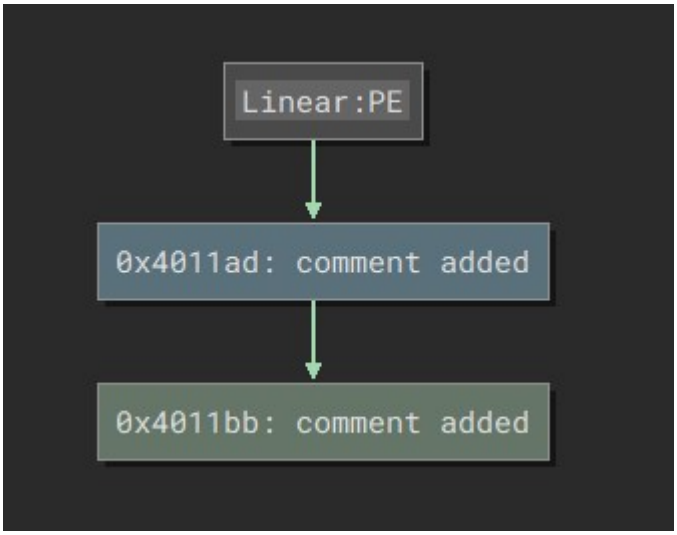


Figure 6: Provenance Graph Representing Added Comment

3.3.3 Undo

Another feature of Provenance Ninja is the ability to undo specific previous actions. Currently, the tool supports undoing the addition and removal of comments. At the bottom of the user interface is a button labeled ”Undo” that a user can select. Upon receiving input from the undo button, the plugin ensures that the action is a valid action to undo. If not, it displays a message box informing the user that the previous action cannot be undone.

If the prior action is undone, the current node is captured, and its contents are processed. Based upon what type of node is being undone, the undo function takes the appropriate action to reverse the course of action most recently done. After undoing the command, variables reflecting the current node on the graph are updated, the non-active branch is noted with a dashed white line, and the graph is redrawn to show its current state. In Figure 7 an example can be seen of the last comment from Figure 6 being undone.

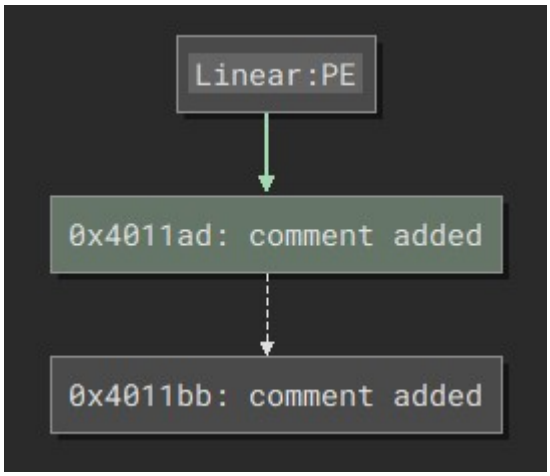


Figure 7: Undoing Comment on Provenance Graph with Verbose Mode Active

3.3.4 Verbose Mode and Collapsing the Graph

In addition to the undo button, the Provenance Ninja interface allows a user to display a verbose version of the graph and collapse the graph. Verbose mode allows a user to display more information about the commands executed, such as the name of an added function or the specific type of a variable changed. Collapsing the graph enables viewing of the current branch and aids in decluttering larger provenance graphs.

The creation of a parallel graph enables verbose mode functionality. In addition to the main graph displayed whenever a command is processed, a second graph is updated with a verbose description of the action that occurred. Suppose the user selects verbose mode, then the plugin switches which graph it displays. This can be toggled back and forth, depending on which version a reverse engineer requires.

A user can also collapse the graph to view only the current branch. At all times, the plugin tracks the current branch in the graph. If the collapse graph feature is toggled, it only draws nodes along the current branch when displaying the graph.

3.4 Tools and Implementation Requirements

All the development is conducted in PyCharm, an integrated development environment (IDE), and executed and tested in Binary Ninja. The IDE is a personal preference. Any code editor would have been sufficient for developing the plugin. The only device requirement is a personal computer running Binary Ninja with the plugin files located in the appropriate folder.

Decreasing the number of external dependencies compared to SensorRE is a design focus. Fewer external dependencies would likely increase the efficiency of the system. Aside from the core Binary Ninja user interface, the only additional dependency it requires is the PySide6 library. Integration requires users to add this library to their Binary Ninja environment. This is an improvement upon SensorRE as it had multiple steps for the user to execute outside of Binary Ninja: starting the SensorRE server from Powershell, installing Javascript, running the Javascript portion of the plugin, and utilizing a web browser. Compared to SensorRE, Provenance Ninja requires fewer steps for a user to execute before running the provenance plugin.

3.5 Usage Scenario

The functionality of Provenance Ninja is demonstrated through a scenario involving a user solving a simple reverse engineering challenge. The provenance graphs for the scenario are shown after following the corresponding actions. The usage scenario simulates how a reverse engineer can use Provenance Ninja to analyze a binary.

First, the user initializes the Provenance Ninja plugin within Binary Ninja, as seen in Figure 8.

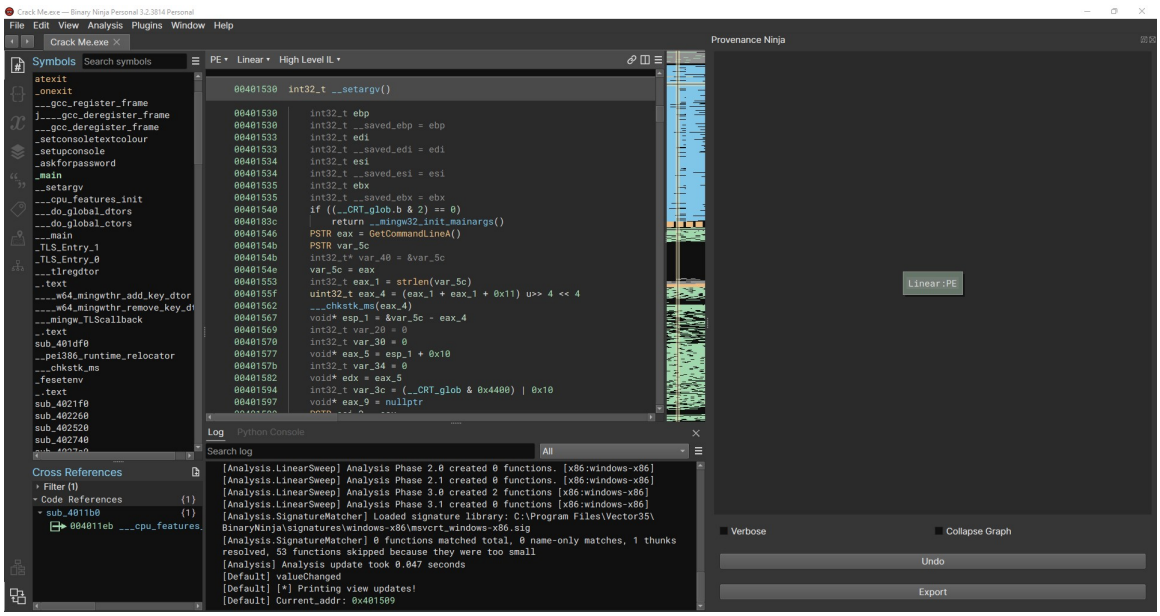


Figure 8: Evaluation Plan for Functionality and Efficiency of Provenance Ninja

Since they are attempting to find a password within this crackme they have begun in the *Linear:PE* view, which is reflected in the graph. Looking through the program, they identify the function that checks for the correct password and the function that calls the password-checking function and other functions. Based upon this information, they deduce that the second function must be the main function and rename it to *main*. Since the first function verifies the user's password, it is named *passVerify*. Next, a comment is added to the variable initially presumed to be the user's entered password. This comment contains the text *"user input?"*. These updates to the graph can be seen in Figure 9.

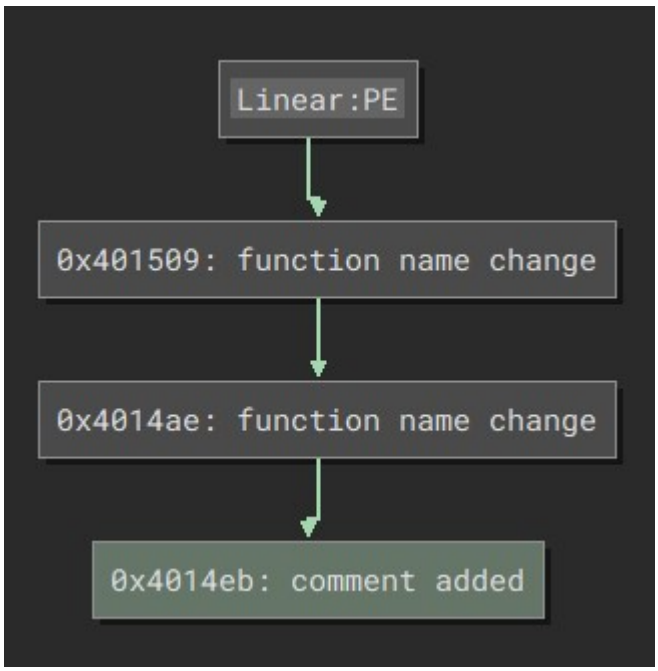


Figure 9: Provenance Graph After First Steps of Usage Scenario

Upon further review of the binary, the user realizes they made an error in their analysis of the user input variable. They undo the comment they previously placed and place the new comment *"user input"* where they eventually determine the location of the variable. They also rename the variable in which the user's input is stored to *"userInput"*. After correctly identifying the input variable, it is now clear where the password against which it is being checked is located. This is noted with the comment *"password"* next to that line, as well as highlighting that line in blue. Also, the reverse engineer notices that a variable labeled as type *void* is supposed to be an *int32t*. These changes are reflected in Figure 10.

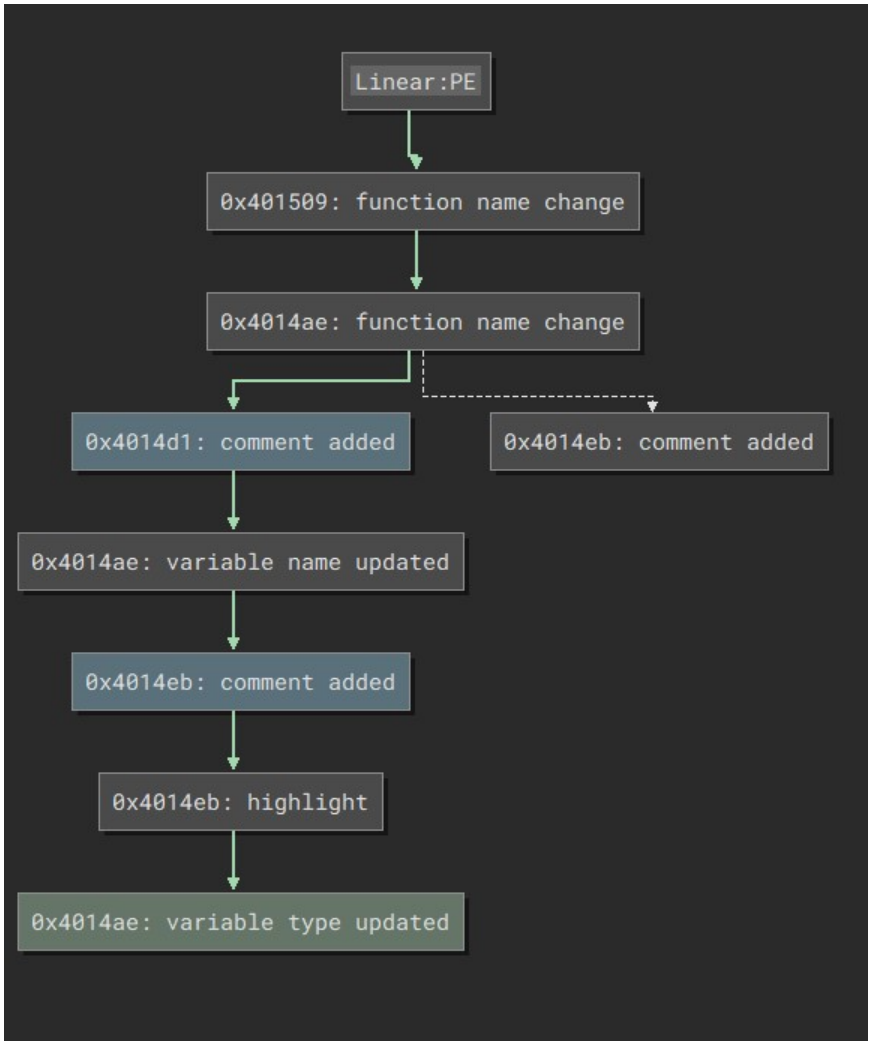


Figure 10: Provenance Graph After Further Steps Taken Usage Scenario

Finally, the analysis of the binary is complete. However, before reporting the work that has been accomplished, the reverse engineer places the graph in verbose mode and collapses it, as seen in Figure 11. This ensures that it is easier to identify the flow of the analysis and view exactly which steps are taken.

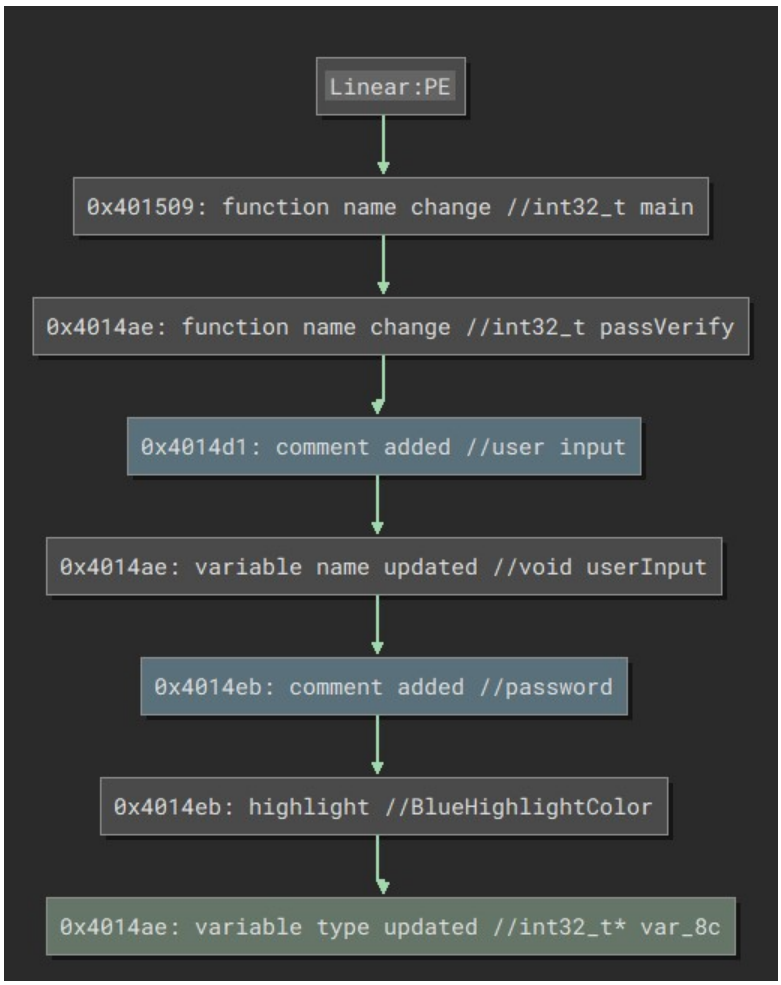


Figure 11: Reportable Provenance Graph From Usage Scenario

3.6 Experiment Hypothesis

The purpose of the experiment is to compare usage to SensorRE. SensorRE is the current standard for a Binary Ninja provenance tool. Thus, the most effective manner of evaluating Provenance Ninja is to compare the two systems. The inherent difference between the two plugins is that one natively runs within Binary Ninja, and the other utilizes an external system. Our experimental hypothesis is that A provenance tool directly within Binary Ninja will have lower memory usage and faster response times than an external system.

First, a functionality test is performed. This compares the feature set of Provenance Ninja to that of the existing tool, SensorRE. It is important to note that this research does not include user testing, as that aspect has already been thoroughly examined in previous research by Henry et al. [7]. Our focus is on the technical capabilities of the tool and its integration with Binary Ninja. Functionality testing ensures Provenance Ninja is a comparable tool to be analyzed against SensorRE.

Next, two metrics tested in this hypothesis are memory usage and response time. These factors provide a quantitative measure of whether the difference in efficiency between the two plugins is statistically significant. If the average memory usage and response time of Provenance Ninja is lower than that of SensorRE to a statistically significant measure, then the hypothesis will fail to be rejected. The system under test diagram in Figure 12 details how these metrics are measured.

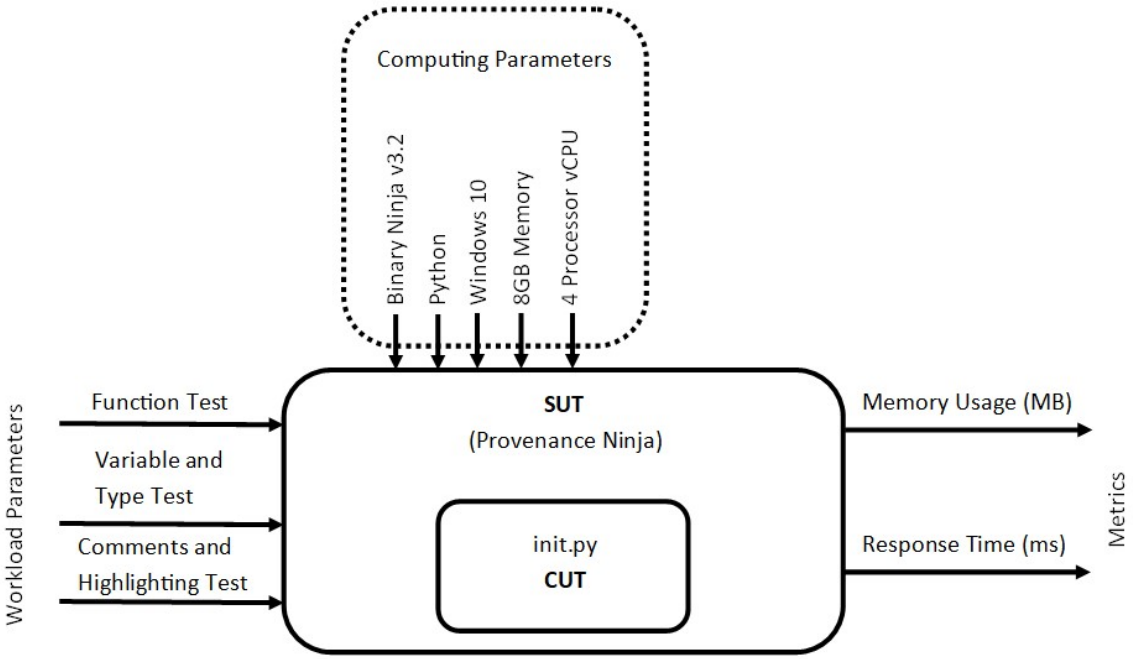


Figure 12: System Under Test Diagram

3.7 Functionalitity Testing Setup

Identical to SensorRE, Provenance Ninja captures five core functions within Binary Ninja, as shown in Figure 13.

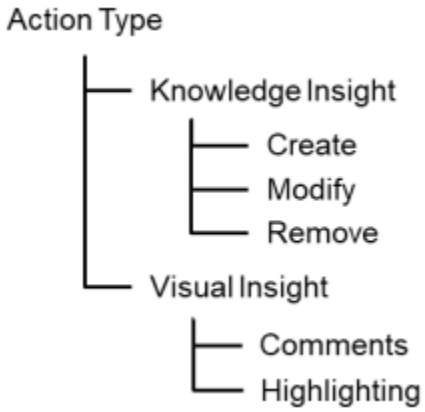


Figure 13: Primary Capturable Actions within Binary Ninja [7]

The primary actions that Binary Ninja tracks are creating, modifying, and removing read-only data, variables, type information, and functions [15]. These are the insights that reflect knowledge acquired by the user when analyzing the binary [7]. Capturing comments and highlighting reflects visual insights, which are how users visually interpret the knowledge insights they are analyzing. SensorRE is capable of visualizing:

- Creation of Read-only data
- Creation and Removal of Data Variables
- Modification of Variables
- Defining and Undefining of Data Types
- Creation, Modification, and Removal of Functions
- Creation and Removal of Comments

- Creation of Highlighting

Thus, functionality testing must show that Provenance Ninja can track all user session actions. The unit tests for evaluating memory usage and runtime analysis show that Provenance Ninja can accomplish those tasks. Because the plugin has these capabilities, it meets the primary goal of improving a reverse engineer's workflow.

3.7.1 Function Test

The first unit test run focuses on functions. In this test, a single branch ten-node graph is created by:

- Creating a function at a random address
- Changing the name of the new function to a different value than its default title
- Changing the type of the new function to a different value than its default title
- Changing the name and type of the new function to values unique from its defaults and the previously used testing values
- Removing the function from the binary

3.7.2 Variable and Type Test

Variables and type information are the focus of the second unit test. It also uses the feature to undo comments within the plugin to test two branches. A single-branch eight-node graph is created by:

- Creating a read-only variable at a random location
- Creating a data variable at a random location
- Removing the new data variable
- Defining a new data type

- Undefining the new data type
- Changing an existing variable's name to a different value than its default name
- Changing the same variable's name to a different value than its default type
- Changing the name and type of the same variable to values unique from its defaults and the previously used testing values

3.7.3 Comments and Highlighting Test

Finally, the third test focuses on comments and highlighting and handling multiple branches. A three-branch fifteen-node graph is created by:

- Creating five comments at unique random locations
- Undoing the most recently created comment
- Highlighting a random line
- Creating two comments at unique random locations
- Removing the two created comments
- Undoing the removal of the comments utilizing the undo button within the Provenance Ninja user interface
- Highlighting a random line
- Creation of two comments at unique random locations

Note that undoing a comment through the plugin does not create a new node while removing the comment in the binary does. These three tests ensure that the functionality of Provenance Ninja is comparable to that of SensorRE.

3.8 Experimental Design Setup

Two core response variables factor into this experiment: memory usage and response time. Each of these variables considers the size of the binary being tested and the number of operations performed on the given binary. Testing is conducted on a virtual machine (VM) running Windows 10 with 8 gigabytes (GB) of RAM, 4 processors, a 60 GB hard drive, and running Binary Ninja version 3.2.

3.8.1 Response Time

Response time is a factor because a faster response time to user input is an expected outcome of designing the plugin to operate directly within Binary Ninja. This is the time from when a user records an input until the corresponding node is drawn on the provenance graph. To track this metric, extra code must be added to both plugins' codebases.

For Provenance Ninja, response time is measured by adding code to the core python framework that utilizes the system time to detect how much time occurs between a user entering input and the provenance graph displaying the proper output. This metric is measured in milliseconds.

Because SensorRE operates partially in Binary Ninja and partially in a web browser, a slightly different approach is taken. The initial time is still captured in the python plugin. However, the time when the node is displayed is printed to the console in the browser in the typescript execution. Due to system time being the same, regardless of which programming language is used, this yields an effective result.

3.8.2 Memory Usage

Another expected benefit of designing the plugin within the Binary Ninja infrastructure is that it utilizes less memory than one that operates within a web browser via JavaScript. To evaluate this, the memory utilization (in megabytes) of Binary Ninja, the web browser, and the PowerShell program is captured in the task manager. The latter two only apply to SensorRE, and the first is relevant to both plugins.

3.8.3 Choice of Statistical Test

For statistically measuring the results of these tests, a multivariate analysis of variance (MANOVA) test is used. This test seeks to determine the effect of two different provenance plugins on response time and memory usage. Unit tests are replicated on small, medium, and large graphs ten times each for 30 trials for each tool. The results of these tests are then measured with an ANOVA test. Each ANOVA test is run at a 95 percent confidence level. If the p-value of the ANOVA test for each factor is below 0.05, then there is a statistically significant improvement in that factor.

3.9 Ensuring Isolation

To ensure that the results gathered in the experiment are isolated solely to what is being tested, both the operating system and Binary Ninja Version are held constant. Windows 10 is the sole operating system for testing to reduce unknown variance. The latest performance update to Binary Ninja at the time of testing is version 3.2 [15], so this is the version used.

The most significant potential nuisance factor in this test is the inability to properly measure the memory usage of the plugins due to having to separate their workload from the innate workload of Binary Ninja and, for SensorRE, the web browser as

well. This is handled by visually observing which programs in the task manager are affected by each plugin running before conducting tests. For SensorRE, Binary Ninja, the Google Chrome web browser, the Powershell window, and the Node.JS Server-side Javascript programs are all affected. When running Provenance Ninja, the only change in memory that this test is concerned with is that of Binary Ninja. Testing measures the memory values of these programs before any plugins are launched, again after launch, and continues to monitor them after each node is added to the graph.

3.10 Randomization and Threats to Validity

Randomization is added to this experiment through the design of the unit tests. Each test focuses on testing different functions of the provenance plugins, ensuring that all major features are covered. The memory locations within the binary at which the actions are taken are randomized for each test. This randomization ensures the values gathered are as average as possible.

There are two primary threats to the validity of this experiment. Externally, a concern is that the tests may not be typical of an actual reverse engineering environment. It may seem that the commands are selected to produce optimal results. However, this is remedied in the user tests by writing them to display the core features of Binary Ninja on the provenance graph. These are the same features that reverse engineers use in their day-to-day tasks. Internally, a potential factor could be personal bias. However, in constructing these tests around the core features of Binary Ninja, the personal bias of which factors are tested is reduced. Additionally, testing metrics are defined in such a way that any researcher may utilize this metric in comparing the efficiency of two different provenance plugins.

3.11 Methodology Summary

This chapter presented Provenance Ninja, the unique provenance tool that runs natively within Binary Ninja as a Python plugin. It detailed its development, including important details such as which features were chosen to be added, how the graph was constructed, and how some of the more complex features were implemented. The chapter then presented an example of how reverse engineers could use Provenance Ninja in their daily workflow. Finally, the plan for testing the plugin against SensorRE was outlined.

IV. Results and Analysis

4.1 Overview

This chapter provides a comprehensive overview of the key concepts and technologies that form the foundation of this research. The chapter begins by discussing the importance of sensemaking in software reverse engineering, followed by an in-depth analysis of the key concepts and techniques used in software reverse engineering. Next, the chapter describes various visualization techniques commonly used in analytics and the design of tools for evaluating them. The chapter then provides a detailed overview of the concepts of provenance and the implementation technologies used in this research.

4.2 Functionality Testing

Functionality testing shows that all desired performance is present within Provenance Ninja. For the performance testing, the 3 tests outlined in Chapter 3 are ran 10 times each. To validate functionality, a provenance graph from one of each of those tests is displayed in Figure 14. This figure shows that all the intended capturable actions from the function tests are properly received by Provenance Ninja and displayed to the user in a graph.

This demonstrates that Provenance Ninja is functionally comparable to SensorRE. It is a fully featured provenance tool that can display commands useful to a reverse engineer when analyzing a binary. Now that functionality is verified, the performance tests can begin.

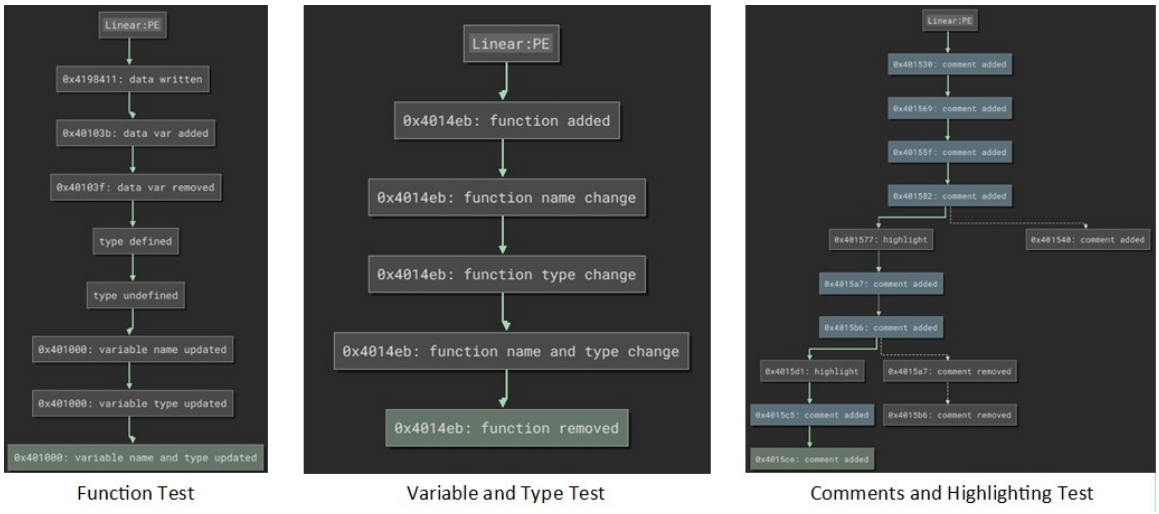


Figure 14: Results of Functionality Testing

4.3 Performace Testing

The performance testing assesses the memory efficiency and response time of Provenance Ninja versus SensorRE. As mentioned in Chapter 3, an ANOVA test is run on each factor to determine if there is a statistically significant difference between the two plugins at a 95 percent confidence level. If the research hypothesis that Provenance Ninja utilizes less memory and operates with faster response times compared to SensorRE fails to be rejected, tests indicate a performance-related benefit to developing a plugin natively within a reverse engineering tool. Otherwise, it shows no significant performance increase from a native plugin.

Two Windows 10 virtual machines (VMs) are created to set up the testing environment. Both VMs are loaded with Binary Ninja; one is set up to run the SensorRE plugin, and the other is to run Provenance Ninja. The tests are run in separate VMs from clean Windows installations to ensure that both plugins can run uninhibited.

To set up SensorRE, the steps from its GitHub page are followed. The source code is first installed from the repository. Then the necessary dependencies of NodeJS

version 8.12.0 and yarn are installed using Powershell. Next, the plugin is placed into the Binary Ninja plugins folder. A simple crackme from crackmes.one is opened with Binary Ninja, and the XML server is started. In the shell, the SensorRE server is run. Finally, the provenance system is booted in a separate Powershell window, and a Chrome web browser is opened to display the SensorRE graph at the address http://localhost:8080/.

For the Provenance Ninja setup, the code is first placed directly into the Binary Ninja plugin folder. Then, the networkx library is added to the Python framework Binary Ninja uses to run plugins. Python is installed on the VM, and in the Binary Ninja settings, the installation of python is set as the interpreter. Now that the plugin can be run, it is ready for testing.

4.3.1 Gathering Data

Data is gathered in each plugin, one set of tests at a time. Each test is run 10 times for a total of 30 trials. For each test, the environment is reset to ensure no residual data affects the memory values. The relevant memory values from the task manager are recorded prior to the plugin being launched. Once the plugin is launched, the baseline memory values are recorded as seen in the left of Figure 15. There are two Powershell instances, a Google Chrome window and Binary Ninja. Then, the tests are run with the memory values tracked after each node is added and the time it takes the plugin to display the node after an input is received. The image to the right of Figure 15 shows how the memory values increased for SensorRE after one node is placed on the graph. The memory recording for Provenance Ninja works the same way, just without Powershell and Google Chrome.

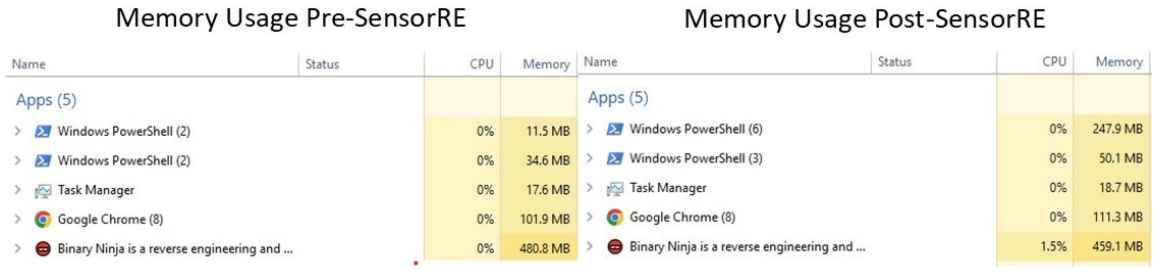


Figure 15: Memory Usage of SensorRE

Within both plugins, code is added to track the system time when an input is received and when a node is printed to the graph for the user to view. For SensorRE, the start time is tracked in python and printed to the console of Binary Ninja as shown in the top section of Figure 16. The finish time is recorded within the typescript and displayed in the console in the web browser as seen in the lower part of Figure 16.

The current memory address and comment content are noted in the Binary Ninja console and corroborated in the Google Chrome console. This ensures that the start and stop times are associated with the same action. Despite being measured in different programming languages, both measurements utilize system time, which is the same regardless of the runtime environment. Both values are recorded, and the start time is subtracted from the finish time to acquire the runtime. In Provenance Ninja, both happen within the python code, so the subtraction is done in the code and printed directly to the console.

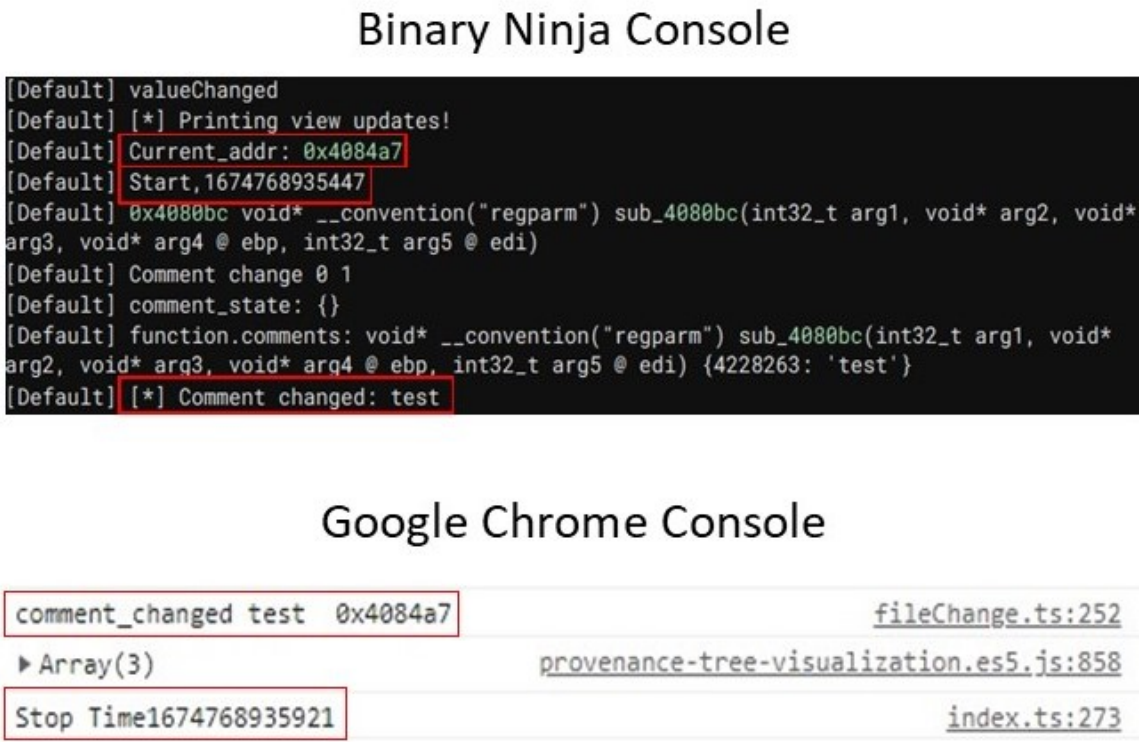


Figure 16: Runtime of SensorRE

4.3.2 Memory Analysis Results and Observations

Memory testing shows several vital insights. Across all three unit tests, there is no statistically significant difference between the memory usage of Binary Ninja itself. Across all tests for both plugins, the memory usage of Binary Ninja decreases as more nodes are added to the graph. With this number similar across both plugins, the improvements in memory are found in the external programs SensorRE requires to run.

The total value for the amount of memory used across the two plugins as the number of nodes on the graph increases is seen in Figure 17. The Provenance Ninja number is representative of solely the memory usage of Binary Ninja. For SensorRE the total includes Powershell and Google Chrome.

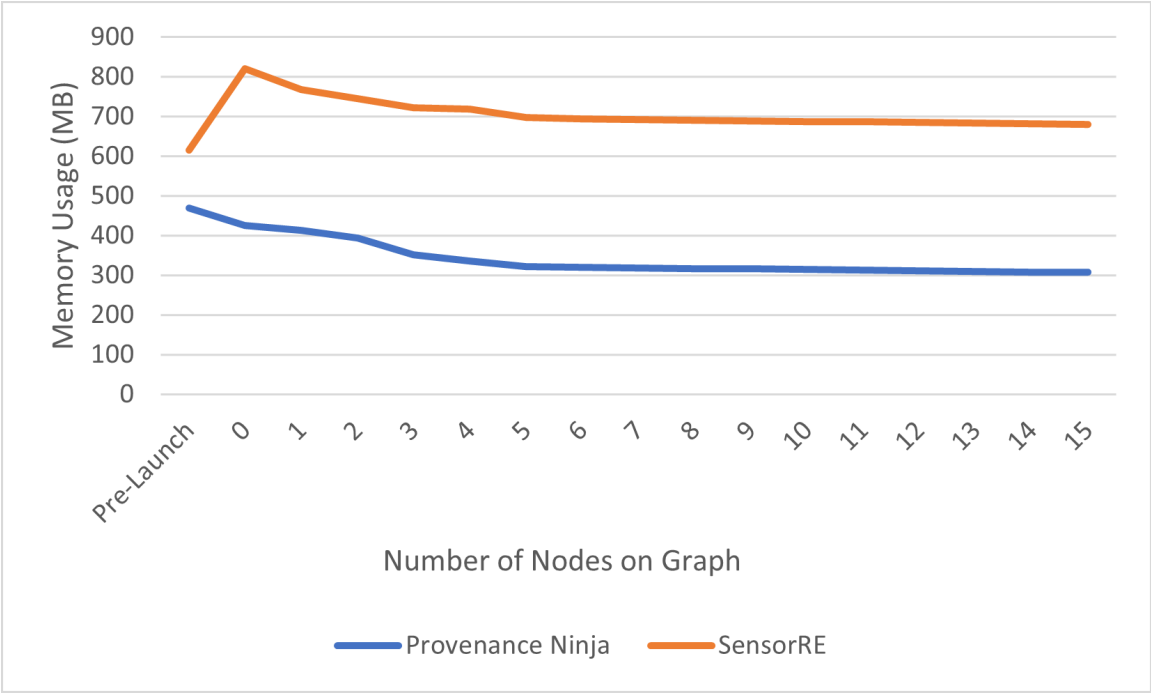


Figure 17: Average Memory Usage by Number of Nodes

SensorRE has two main external programs supporting it: Powershell and Google Chrome. Powershell utilizes a consistent memory value once the program is started that remains consistent as more nodes are added. Chrome has a variable memory usage when started; however, once connected to the server, it stabilizes at a lower rate.

The reason for the memory values across all programs remaining stable, regardless of how many nodes or branches are added to the graph, is likely that both plugins utilize low graphically intensive visualizations. A potential reason for this is that the information stored in the graphs is text strings, which do not require a high memory capacity. Due to this, even when commands are undone, the memory load on the system is not significantly affected.

For Powershell, it makes sense that it would experience a spike in memory once the server is being utilized. It then remains stable as no information is being stored

in the server, which merely serves as a conduit between Binary Ninja and Google Chrome. The web browser memory values remain stable throughout stages of the visualization as, similarly to what is observed with Binary Ninja, the text strings are not capacitively demanding.

To determine if there is an improvement in the memory usage of Provenance Ninja compared to SensorRE the data is consolidated into the average memory usage of each test as seen in Figure 18

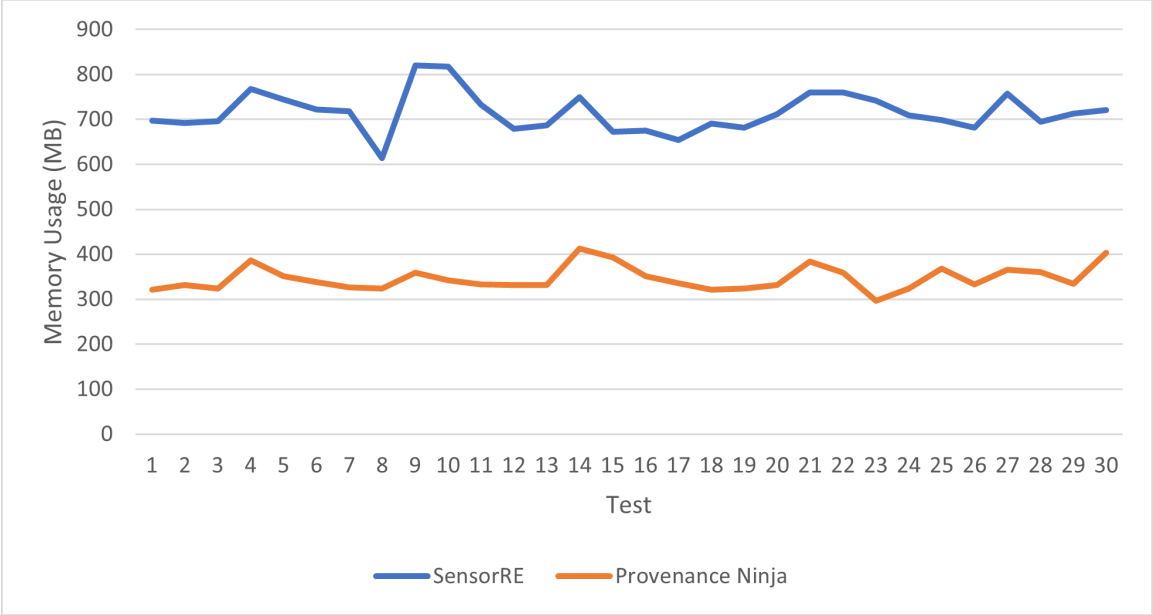


Figure 18: Average Memory Usage Across Tests

This graph shows a notable difference between the memory usage of SensorRE and Provenance Ninja in line with the average memory usage by the number of nodes. Each average has few points that deviate from linearity, with test 8 on SensorRE being the only point straying from the norm. Because of this, an ANOVA test is run on this dataset. The p-value from this test is 4.637E-21. This value is less than 0.05 showing that there is a statistically significant difference between the memory usage of Provenance Ninja and SensorRE. This shows a proven improvement in memory

from developing a provenance plugin to run within the Binary Ninja environment instead of utilizing an external server. The confidence intervals for the averages from this test are shown in Figure 19.

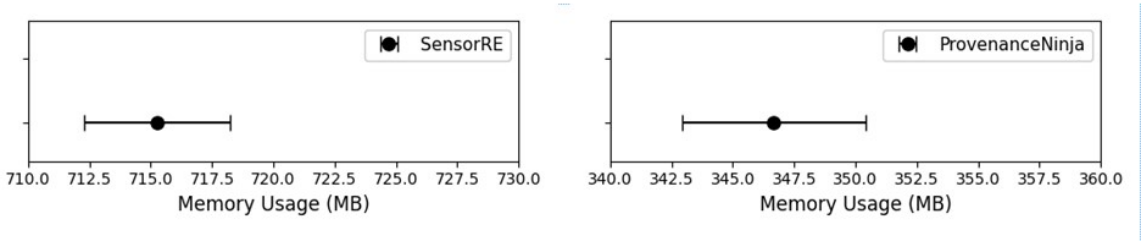


Figure 19: Average Memory Confidence Intervals

The confidence intervals are shown on two separate graphs due to the disparity between the averages. This shows that, at a 95 percent confidence interval, there is no chance of the memory usage of the two plugins overlapping, further reinforcing the results of the ANOVA test.

4.3.3 Runtime Analysis Results and Observations

The runtime tests also yield profitable results. On the surface, it is difficult to tell if there is a margin of improvement when observing the average runtimes across all tests.

There is no notable difference in the runtime of different commands between the two provenance tools. This is most likely because once it is determined that a change to the binary has been made, both plugins follow a similar logic process. The improvement comes in the time it takes to send that information to the graph. SensorRE transmits this data through an external server, while Provenance Ninja does not have this restraint. Because of this, the average can be taken of each test runtime without regard to which command was run during that test. The memory values from each test are averaged and compiled in Figure 20.

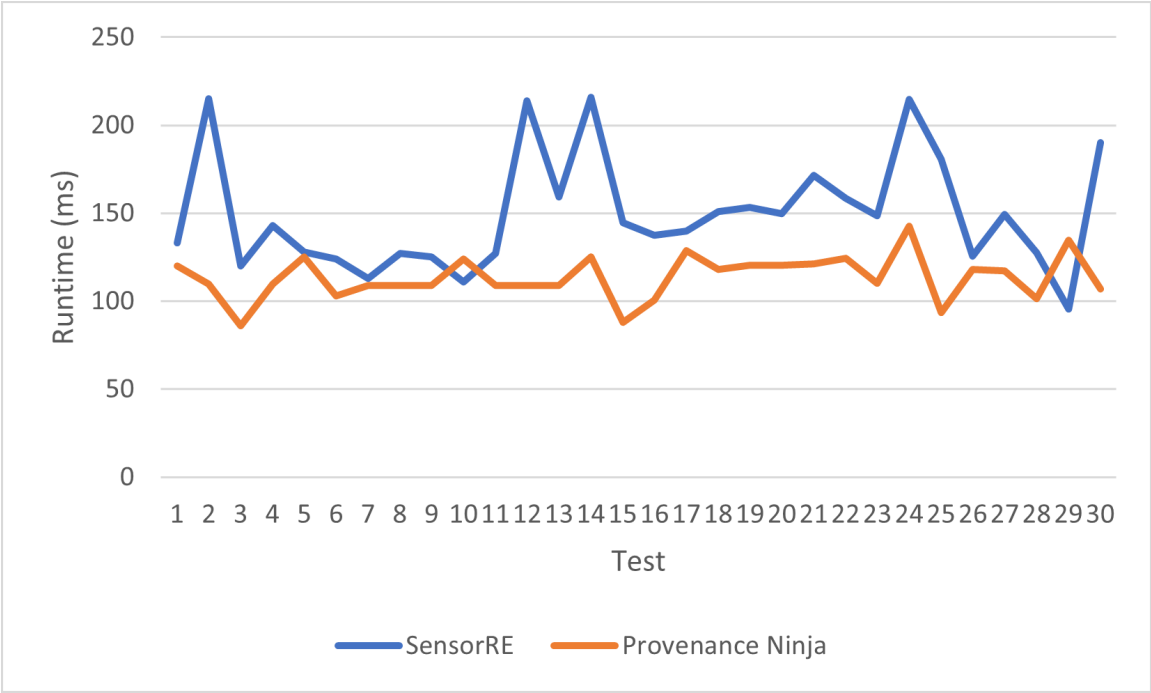


Figure 20: Average Runtime by Test

From this graph, it is not apparent if there is a notable difference between the memory values of SensorRE and Provenance Ninja. To determine if there is any improvement, an ANOVA test is run on the values from Figure 20. The ANOVA test returns a p-value of 0.02692. As this is less than the selected confidence interval of 0.05, the improvement in runtime is of a statistically significant margin. However, the normality of the data is not verified from the average runtime. To ensure a lack of overlap between the average runtimes of the two plugins, given their standard deviations, a confidence interval graph is used as shown in Figure 21.

From this, which utilizes the mean and standard deviation of the two data sets, it can be more clearly seen that there is little chance of the average runtime of either plugin reaching a point that invalidates the statistical significance of the ANOVA test.

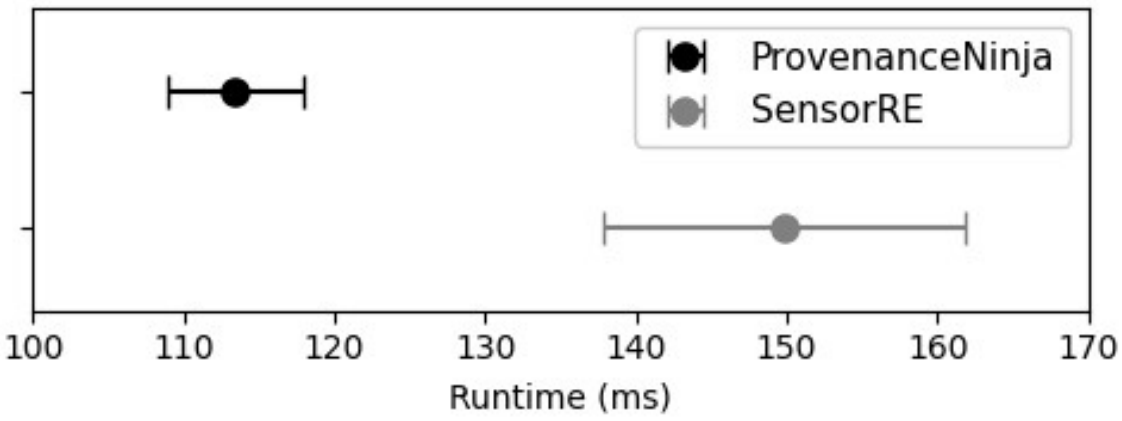


Figure 21: Average Runtime Confidence Intervals

This confirms the testing hypothesis and makes logical sense. When SensorRE receives input from the user, it first sends that information through the server before it is received and displayed by the web browser. However, Provenance Ninja receives the input, adds nodes to the graph, and visualizes them all from within the Python plugin running in Binary Ninja. The testing shows there is a benefit to this, as Provenance Ninja benefits from faster runtimes.

4.4 Results Summary

This chapter analyzed the results of the memory usage and runtime testing on Provenance Ninja and SensorRE. In both metrics, Provenance Ninja shows statistically significant improvement. The decrease in memory usage derives from Provenance Ninja wielding unique freedom from Powershell and an external web browser. These freedoms also lower the runtime by removing the need to transmit data across an external server.

V. Conclusions

Provenance tools benefit reverse engineers in a myriad of ways. These tools enable users to view changes made to a binary in real time, undo previous actions, locate where prior changes were made, and quickly summarize their findings upon completion. Reducing the time taken for software analysis in this way holds benefits for any user, especially in time-critical tasks.

Research presented the development of a novel Python-based provenance plugin called Provenance Ninja and implemented it directly within Binary Ninja. Testing confirmed that this tool was more memory efficient and faster than SensorRE. This chapter summarizes the research done and its contribution to reverse engineering.

5.1 Research Contributions

Provenance Ninja is a first-of-its-kind provenance tool for Binary Ninja that runs natively within reverse engineering software. This meets the first research objective and shows that a fully featured reverse engineering provenance tool can be designed as a Binary Ninja plugin. Several libraries within Python and nuanced design choices were used to make this a reality. Because Python plugins can be installed directly from within Binary Ninja, this increases the accessibility of provenance tools for reverse engineers.

Not only does this research show that such a tool can be created, but also that it is more efficient than SensorRE. As the current state-of-the-art reverse engineering provenance tool, SensorRE is a natural candidate for benchmark comparison. The contents of Chapter 4 show that Provenance Ninja is indeed more memory efficient and faster than SensorRE, answering the second research question. This confirms the research hypothesis that designing a provenance tool to operate as a Python plugin

within Binary Ninja is more efficient than one that utilizes external programs.

For the reverse engineering community, this has the effect of improving their workflow on a day-to-day basis. Improving this process can directly lead to critical tasks being solved more efficiently. Additionally, designing the tool to be more accessible ensures a wider reach of reverse engineers can use this tool.

5.2 Research Limitations

This research focused on capturing essential functions of SensorRE and focused on the attributes that would most directly affect memory and runtime. Because of that, there were features of SensorRE that were not implemented into Provenance Ninja. The most notable limitation is the ability to undo any activity recorded. For testing, only commenting was given the functionality to be undone with a popup textbox displaying if another command is executed before pressing the undo button.

Another feature of SensorRE not present in Provenance Ninja is the ability to save a provenance graph. After researching this area, there is no currently established method of saving the graph created with the Python libraries utilized. Such an effort would have to be done manually and was not within the scope of this research.

5.3 Future Work

Based on the conclusions of this research, there are several avenues for further research.

1. Other performance metrics can be analyzed, such as CPU performance, graphics performance, and the amount of disk space within Binary Ninja that the plugin takes up. While these metrics may bear little effect on some users, in environments where resources are tight, even a small improvement in one of these metrics could go a long way.

2. Research can be conducted on incorporating multiple ways of viewing provenance data within Provenance Ninja. Currently, it only provides the linear graph view. However, if user studies prove that additional views, such as a table view, are useful to reverse engineers, that capacity can be added.

3. The ability to recreate the state of a binary from previous provenance data is an area that can be helpful to reverse engineers. Future research can explore the viability of this area and incorporate that feature into Provenance Ninja.

4. There are capabilities present in SensorRE not currently implemented in Provenance Ninja. Future research could focus on expanding and improving the functionality of the plugin.

5. When conducting testing, it was noted that adding nodes decreased the memory usage of Binary Ninja. The reason for this is currently unknown. Further exploration into this area is a potential future research topic.

6. This research chose to use Windows Task manager to evaluate memory performance. Further exploration could use a tool such as memprofiler to determine if memory leaks hinder provenance tools.

7. While this research only explored provenance analytics, this approach could be applied to other domains to build reverse engineering tools for a variety of purposes within Binary Ninja or other applications.

5.4 Concluding Thoughts

Analyzing a binary is an intensive process. Provenance tools are an effective method of making that endeavor easier for a reverse engineer to process. This research improved the current software reverse engineering provenance resources by

developing a Python-based plugin for Binary Ninja. Provenance Ninja displays real-time information to a user about comments, functions, variables, and highlighting. It does so in a manner that is more resource efficient than prior software while operating natively within the Binary Ninja environment. Provenance Ninja represents a significant advancement in the field of reverse engineering and we believe that it will be a valuable resource for practitioners in the field.

Appendix A. Software Listings

init.py

```
1  import threading
2  from binaryninja import *
3  from binaryninjaui import DockHandler, DockContextHandler,
       UIActionHandler, FlowGraphWidget, ViewFrame
4  from PySide6.QtCore import Qt
5  from PySide6.QtWidgets import (QApplication, QHBoxLayout, QVBoxLayout,
       QWidget,
6                                 QPlainTextEdit, QPushButton, QCheckBox)
7  from PySide6.QtGui import QFont, QFontMetrics
8  import gc
9  import networkx as nx
10 import time
11 from collections import defaultdict, OrderedDict
12
13 ######################################################################
14 ################### GLOBAL VARIABLES #################################
15 widgets = []
16
17 current_addr = 0x401000
18 current_view = "Graph:PE"
19 var_state = []
20 prev_var_name = []
21 prev_var_type = []
22 func_name = None
23 comment_state = None
24 highlight_state = None
25 data_state = None
```

```
26 dict_funcs = None
27 func_type = None
28 eventfunc = time.time()
29 prov_graph = nx.Graph()
30 verbose_prov_graph = nx.Graph()
31 curr_node = 0
32
33 ##TESTING
34 start_time = 0
35 stop_time = 0
36
37 #initializing list of -1 values for tracking branching
38 branches = [[-1]*100 for i in range(100)]
39 current_branch = 0
40 num_branches = 1
41 branch_counter = 0
42
43 #variable to track whether an undo is currently occuring
44 undo_occurring = False
45
46 ##########################################################################
47 ###################### CREATION OF THE WIDGET ############################
48 #Creates the widget
49 class ProvenanceNinjaDockWidget(QWidget, DockContextHandler):
50     # initializer for entire Dock Widget
51     def __init__(self, parent, name, bv, data):
52
53         global widgets, branch_counter, current_branch
54         widgets.append(self)
55
56         # initializes the QWidget
57         QWidget.__init__(self, parent)
```

```
58
59          # initializes the dock handler
60          DockContextHandler.__init__(self, self, name)
61          self.actionHandler = UIActionHandler()
62          self.actionHandler.setupActionHandler(self)
63
64          # creates a main display widget
65          textbox_layout = QVBoxLayout()
66          self.textbox = QPlainTextEdit()
67          self.textbox.setLineWrapMode(QPlainTextEdit.LineWrapMode.NoWrap)
68          self.textbox.setReadOnly(True)
69          font = QFont("Helvetica")
70          self.textbox.setFont(font)
71          font = QFontMetrics(font)
72          self.textbox.setMinimumWidth(40 * font.averageCharWidth())
73          self.textbox.setMinimumHeight(30 * font.lineSpacing())
74          # textbox_layout.addWidget(self.textbox)
75
76          #initializes the graph
77          self.flow_graph = FlowGraphWidget(None, data)
78          graph = FlowGraph()
79
80          #(temporary) graph
81          prov_graph.add_node(1, data=bv.file.view)
82          verbose_prov_graph.add_node(1, data=bv.file.view)
83
84          #creates (temporary) intial nodes
85          node_a = FlowGraphNode(graph)
86          node_a.lines = [prov_graph.nodes[1]['data']]
87          node_a.highlight = enums.HighlightStandardColor.
                GreenHighlightColor
88          graph.append(node_a)
```

```
89          #first branch, first spot, node 0
90          branches[current_branch][branch_counter] = 0
91          branch_counter += 1
92
93          self.flow_graph.setGraph(graph)
94          textbox_layout.addWidget(self.flow_graph)
95
96          #add undo button
97          self.buttonsWidget = QWidget()
98          self.buttonsWidgetLayout = QHBoxLayout(self.buttonsWidget)
99          self.undo_button = QPushButton("Undo")
100         self.undo_button.clicked.connect(self.undo)
101         self.buttonsWidgetLayout.addWidget(self.undo_button)
102
103         #add the verbose and collapse checkboxes
104         self.checkboxWidget = QWidget()
105         self.checkboxWidgetLayout = QHBoxLayout(self.checkboxWidget)
106         self.verbose_check = QCheckBox("Verbose")
107         self.collapse_check = QCheckBox("Collapse Graph")
108         self.verbose_check.stateChanged.connect(self.update_graph)
109         self.collapse_check.stateChanged.connect(self.update_graph)
110         self.checkboxWidgetLayout.addWidget(self.verbose_check)
111         self.checkboxWidgetLayout.addWidget(self.collapse_check)
112
113         #add export button
114         self.exportWidget = QWidget()
115         self.exportWidgetLayout = QHBoxLayout(self.exportWidget)
116         self.exportButton = QPushButton("Export")
117         #self.exportButton.clicked.connect(self.undo)
118         self.exportWidgetLayout.addWidget(self.exportButton)
119
120         textbox_layout.addWidget(self.checkboxWidget)
```

```
121         textbox_layout.addWidget(self.buttonsWidget)
122         textbox_layout.addWidget(self.exportWidget)
123         #undo_button.clicked.connect(temp_func)
124 
125 
126         # Putting all the child layouts together
127         layout = QVBoxLayout()
128         layout.addLayout(textbox_layout)
129         self.setLayout(layout)
130 
131 
132         # Actual storage variables
133         self.bv = bv
134         self.filename = None
135 
136     def set_text(self, text):
137         self.textbox.setPlainText(text)
138 
139     def set_line(self, text):
140         self.line_info.setText(text)
141 
142     def set_function(self, text):
143         self.function_info.setText(text)
144 
145     def shouldBeVisible(self, view_frame):
146         if view_frame is None:
147             return False
148         else:
149             return True
150 
151     def notifyViewChanged(self, view_frame):
152         if view_frame is None:
```

```
153             self.datatype.setText("None")
154             self.data = None
155         else:
156             self.datatype.setText(view_frame.getCurrentDataType())
157             view = view_frame.getCurrentViewInterface()
158             self.data = view.getData()
159 
160     def contextMenuEvent(self, event):
161         self.m_contextMenuManager.show(self.m_menu, self.actionHandler)
162 
163     def undo(self):
164         global curr_node, undo_occurring
165 
166 
167         #if it is a valid node to be undone
168         if curr_node > 0:
169             # sets flag so that no graph nodes are added as a result of
                    undoing actions
170             undo_occurring = True
171 
172             # captures the number of the current node
173             undo_command = -1
174             for i in range(len(branches[current_branch])):
175                 if branches[current_branch][i] == curr_node:
176                     undo_command = branches[current_branch][
                            branch_counter - 1]
177                     break
178             # captures the text from that node
179             undo_node = verbose_prov_graph.nodes[undo_command + 1]['data
                    ']
180             # isolates the address where the action occurred
181             split = undo_node.split(':')
```

```
182             address = split[0]
183
184             #filters based upon what the action was
185             valid_action = False
186             if 'comment added' in split[1]:
187                 # gets the function in which the comment was made
188                 func = self.bv.get_functions_containing(int(address, 16)
                        )
189                 # removes the comment at that location
190                 func[0].set_comment_at(int(address, 16), "")
191                 print("undid comment at " + address)
192                 valid_action = True
193             elif 'comment removed' in split[1]:
194                 # gets the function in which the comment was made
195                 func = self.bv.get_functions_containing(int(address, 16)
                        )
196                 # removes the comment at that location
197                 print(split[1])
198                 comment = split[1].split("//")
199                 print(comment)
200                 func[0].set_comment_at(int(address, 16), comment[1])
201                 print("added comment at " + address)
202                 valid_action = True
203             else:
204                 show_message_box("Error", "Cannot undo this command",
205                                  MessageBoxButtonSet.OKButtonSet,
                                     MessageBoxIcon.InformationIcon)
206                 print("Cannot undo this command")
207
208             if valid_action:
209                 #resets the position of the current node
210                 for i in range(99):
```

```
211                     if branches[current_branch][i+1] < 1:
212                         curr_node = 0
213                         break
214                     elif curr_node == branches[current_branch][i+1]:
215                         curr_node = branches[current_branch][i]
216                         break
217 
218                 #update graph to reflect undone action
219                 self.update_graph(0)
220 
221                 #jump to most current action (the new current node)
222                 curr_action = prov_graph.nodes[curr_node+1]['data']
223                 #if it is an action and not the first node in the graph
224                 if '0x' in curr_action:
225                     split = curr_action.split(':')
226                     address = split[0]
227                     self.bv.file.navigate(self.bv.file.view, int(address
                            , 16))
228 
229         undo_occurring = False
230 
231     def update_graph(self, id = None):
232         #Update graph to display a certain state
233 
234         graph = FlowGraph()
235 
236         nodes = []
237         for i in range(prov_graph.number_of_nodes()):
238             # check if the graph should be collapsed
239             #print("i is " + str(i) + "curr branch contains " + str(
                    branches[current_branch]))
240             if self.collapse_check.isChecked():
```

```
241                 if i not in branches[current_branch] or i > curr_node:
242                     continue
243             n = FlowGraphNode(graph)
244             if (i == curr_node):
245                 n.highlight = enums.HighlightStandardColor.
                        GreenHighlightColor
246             elif i > curr_node or i not in branches[current_branch]:
247                 n.highlight = enums.HighlightStandardColor.
                        NoHighlightColor
248             elif 'comment' in prov_graph.nodes[i+1]['data']:
249                 n.highlight = enums.HighlightStandardColor.
                        BlueHighlightColor
250             #decide between the verbose text and the simplified text
251             if self.verbose_check.isChecked():
252                 n.lines = verbose_prov_graph.nodes[i+1]['data']
253             else:
254                 n.lines = prov_graph.nodes[i+1]['data']
255             graph.append(n)
256             nodes.append(n)
257
258
259
260         current_edge = EdgeStyle(EdgePenStyle.SolidLine, 2, ThemeColor.
                GreenStandardHighlightColor)
261         past_edge = EdgeStyle(EdgePenStyle.DashLine, 1, ThemeColor.
                WhiteStandardHighlightColor)
262         outgoing_edges = [[0]*len(nodes) for i in range(len(nodes))]
263         num_nodes = len(nodes)
264
265         if self.collapse_check.isChecked():
266             for i in range(len(nodes) -1):
267                 nodes[i].add_outgoing_edge(BranchType.UserDefinedBranch,
```

```
                    nodes[i+1], current_edge)
268                 outgoing_edges[i][i+1] = 1
269         else:
270             for i in range(num_branches):
271                 for j in range(99):
272                     branch_drawn = num_branches - (i+1)
273                     x = branches[branch_drawn][j]
274                     y = branches[branch_drawn][j+1]
275                     if x > -1 and y > -1 and y < num_nodes:
276                         if branch_drawn == current_branch and x <
                                curr_node:
277                             nodes[x].add_outgoing_edge(BranchType.
                                    UserDefinedBranch, nodes[y],
                                    current_edge)
278                             outgoing_edges[x][y] = 1
279                         else:
280                             if outgoing_edges[x][y] == 0:
281                                 nodes[x].add_outgoing_edge(BranchType.
                                        UserDefinedBranch, nodes[y],
                                        past_edge)
282                                 outgoing_edges[x][y] = 1
283                     else:
284                         break
285         self.flow_graph.setGraph(graph)
286
287     @staticmethod
288     def create_widget(name, parent, bv, data=None):
289         return ProvenanceNinjaDockWidget(parent, name, bv, data)
290
291     # Creates the graph portion of the widget
292     class FlowGraphWidget(FlowGraphWidget):
293         # initializes the graph portion of the widget
```

```
294         def __init__(self, parent: QWidget, view: BinaryView, graph:
                FlowGraph = None):
295             super().__init__(parent, view, graph)
296             self.bv = view
297
298 ##########################################################################
299 ################### REGISTERING NOTIFICATIONS ###################
300 #Register Notifications and tells plugin where to go for various
        notification types
301 class myNotification(binaryninja.BinaryDataNotification, QWidget):
302     def __init__(self, view, dock_widget):
303         self.view = view
304         self.dock_widget = dock_widget
305         pass
306
307     def data_written(self, view, offset, length):
308         print("data_written: ", view, offset, length)
309         data_written(view, offset, length, self.dock_widget)
310         pass
311
312     def data_inserted(self, view, offset, length):
313         print("data_inserted: ", view, offset, length)
314         pass
315
316     def data_removed(self, view, offset, length):
317         print("data_removed: ", offset, length)
318         pass
319
320     def function_added(self, view, func):
321         print("function_added: ", func)
322         func_added(view, func, self.dock_widget)
323         pass
```

```
324
325     def function_removed(self, view, func):
326         print("function_removed: ", func)
327         func_removed(view, func, self.dock_widget)
328         pass
329
330     def function_updated(self, view, func):
331         print("function_updated", hex(int(func.start)))
332         func_updated(view, func, self.dock_widget)
333         pass
334
335     def data_var_added(self, view, var):
336         data_var_added(view, var, self.dock_widget)
337         print("var_added: ", var)
338         pass
339
340     def data_var_removed(self, view, var):
341         data_var_removed(view, var, self.dock_widget)
342         print("var_removed: ", var)
343         pass
344
345     def data_var_updated(self, view, var):
346         print("var_updated: ", var)
347         pass
348
349     def string_found(self, view, string_type, offset, length):
350         print("string_found: ", string_type, offset, length)
351         pass
352
353     def string_removed(self, view, string_type, offset, length):
354         print("string_removed: ", string_type, offset, length)
355         pass
```

```
356
357     def type_defined(self, view, name, type):
358         global type_flag
359         type_defined(view, str(name), type, self.dock_widget)
360         print("type_defined: ", name, type)
361         pass
362
363     def type_undefined(self, view, name, type):
364         global type_flag
365         type_undefined(view, str(name), type, self.dock_widget)
366         print("type_undefined: ", name, type)
367         pass
368
369
370 ########################################################################
371 ################### INPUT PARSING ######################################
372
373 def func_updated(bv, function, dockwidget):
374     global eventfunc, var_state, prev_var_name, prev_var_type, func_name
        , comment_state, \
375         highlight_state, dict_funcs, branch_counter, current_branch,
            num_branches, curr_node
376     temp_name = 0
377     temp_type = 0
378
379     try:
380         print("{} {}".format(hex(int(bv.get_functions_containing(
            current_addr)[0].start)), str(function)))
381         if (eventfunc + 1 < time.time()):
382             # Check for var name collision (caused spurious var entries)
383             if (str(bv.get_functions_containing(current_addr)) != str(
                function)):
```

```
384                 var_state = function.vars.copy()
385
386             # Local Var name/type change
387             if(len(var_state) == len(prev_var_name) == len(prev_var_type
                    )):
388                 for item, var in enumerate(var_state):
389                     if (str(prev_var_name[item]) != str(function.vars[
                            item].name)) and temp_name == 0:
390                         print("[] Name change: {} {}".format(function.
                                vars[item].name, item))
391                         var_type_new, var_name_new, index = function.
                                vars[item].type, function.vars[item].name,
                                item
392                         var_type_old, var_name_old = prev_var_type[item
                                ], prev_var_name[item]
393                         temp_name = 1
394                     if (str(prev_var_type[item]) != str(function.vars[
                            item].type)) and temp_type == 0:
395                         print("[] Type change: {} {}".format(function.
                                vars[item].type, item))
396                         var_type_new, var_name_new, index = function.
                                vars[item].type, function.vars[item].name,
                                item
397                         var_type_old, var_name_old = prev_var_type[item
                                ], prev_var_name[item]
398                         temp_type = 1
399
400             # Local Var name/type change
401             if (temp_name == 1 and temp_type == 1):
402                 print(
403                     "[*] Var_Updated: func:{} func_addr:{} var_name_new
                            :{} var_type_new:{} var_name_old:{} var_type_old
```

```
                    :{}"
404                 .format(function.symbol.name, str(function)[11:-1],
                        var_name_new, var_type_new, var_name_old,
405                         var_type_old))
406             address = function.start
407             graph_text = '0x{:x}'.format(int(address)) + ": variable
                    name and type updated"
408             verbose_graph_text = '0x{:x}'.format(int(address)) + ":
                    variable name and type updated //" \
409                                 + str(var_type_new) + str(
                                        var_name_new)
410             addNodeToGraph(graph_text, dockwidget,
                    verbose_graph_text)
411
412         elif temp_type == 1:
413             address = function.start
414             graph_text = '0x{:x}'.format(int(address)) + ": variable
                    type updated"
415             verbose_graph_text = '0x{:x}'.format(int(address)) + ":
                    variable type updated //" \
416                                 + str(var_type_new) + " " + str(
                                        var_name_new)
417             addNodeToGraph(graph_text, dockwidget,
                    verbose_graph_text)
418         elif temp_name == 1:
419             address = function.start
420             graph_text = '0x{:x}'.format(int(address)) + ": variable
                    name updated"
421             verbose_graph_text = '0x{:x}'.format(int(address)) + ":
                    variable name updated //" \
422                                 + str(var_type_new) + " " + str(
                                        var_name_new)
```

```
423                 addNodeToGraph(graph_text, dockwidget,
                        verbose_graph_text)
424             var_state = function.vars
425             prev_var_name = []
426             prev_var_type = []
427             for i in range(len(var_state)):
428                 prev_var_name.append(var_state[i].name)
429                 prev_var_type.append(var_state[i].type)
430             print(prev_var_name)
431
432             # Function name/type change
433             # Remember what the func name/type were first (dict_funcs),
                    then send new and old
434             dict_funcs_new = func_types(bv)
435             new_key_change, new_key_diff, new_set_diff = diff_func_types
                    (dict_funcs_new, dict_funcs)
436             print("New key diff: " + str(new_key_diff))
437             #if (new_key_diff > 0):
438             if (new_key_diff != 0):
439                 print('[*] Updating function name {}'.format(function))
440
441                 print("func_new: {} {}".format(new_key_diff,
                        new_set_diff))
442                 old_key_change, old_key_diff, old_set_diff =
                        diff_func_types(dict_funcs, dict_funcs_new)
443                 print("func_old: {} {}".format(old_key_diff,
                        old_set_diff))
444                 fname = function.name
445                 ftype = function.return_type
446
447                 if (old_key_change != 0 and new_key_change != 0):
448                     # eventfunc2 = time.time()  #skip next updates (call
```

```
                updates in other functions)
449             # Name change
450             print("keydiff: {} {}".format(old_key_diff,
                    new_key_diff))
451             if str(ftype) in str(old_key_diff):
452                 print("**** Name updated ****")
453
454                 if fname in new_key_diff:
455                     address = function.start
456                     graph_text = '0x{:x}'.format(int(address)) +
                            ": function name change"
457                     verbose_graph_text = '0x{:x}'.format(int(
                            address)) + ": function name change //"
                            + str(ftype) + " " +  fname
458                     addNodeToGraph(graph_text, dockwidget,
                            verbose_graph_text)
459
460             # Type change
461             elif str(fname) in str(old_key_diff):
462                 print("**** Type updated ****")
463
464                 if fname in new_key_diff:
465                     address = function.start
466                     graph_text = '0x{:x}'.format(int(address)) +
                            ": function type change"
467                     verbose_graph_text = '0x{:x}'.format(int(
                            address)) + ": function type change //"
                            + str(ftype) + " " + fname
468                     addNodeToGraph(graph_text, dockwidget,
                            verbose_graph_text)
469
470             # Name and Type change
```

```
471                     else:  # if (str(list(old_set_diff)[1]) != str(list(
                            new_set_diff)[1])):
472                         print("**** Name and Type updated ****")
473
474                         if fname in new_key_diff:
475                             address = function.start
476
477                             graph_text = '0x{:x}'.format(int(address)) +
                                    ": function name and type change"
478                             verbose_graph_text = \
479                                 '0x{:x}'.format(int(address)) + ":
                                        function name and type change //" +
                                        str(ftype) + " " + fname
480                             addNodeToGraph(graph_text, dockwidget,
                                    verbose_graph_text)
481
482                 dict_funcs = dict_funcs_new
483                 eventfunc = time.time()
484                 # func_name = function.symbol.name
485
486             # Comment state change
487             if ((comment_state != function.comments)):
488                 address, comment, comment_text = None, None, None
489                 comment_state_len = len(comment_state)
490                 new_comment_len = len(function.comments)
491                 print("Comment change {} {}".format(comment_state_len,
                        new_comment_len))
492                 print("comment_state: {}".format(comment_state))
493                 print("function.comments: {} {}".format(function,
                        function.comments))
494                 # Added
495                 if (comment_state_len < new_comment_len or
```

```
                comment_state_len == new_comment_len):
496              for item in function.comments.items():
497                  if item not in comment_state.items():
498                      address = item[0]
499                      comment = item[1]
500                      comment_text = "comment_changed"
501                      print("[*] Comment changed: {}".format(
                             comment))
502                      graph_text = '0x{:x}'.format(int(address)) +
                             ": comment added"
503                      verbose_graph_text = '0x{:x}'.format(int(
                             address)) + ": comment added //" +
                             comment
504                      addNodeToGraph(graph_text, dockwidget,
                             verbose_graph_text)
505
506          # Removed comment
507          elif (comment_state_len > new_comment_len):
508              for item in comment_state.items():
509                  if item not in function.comments.items():
510                      address = item[0]
511                      comment = item[1]
512                      comment_text = "comment_removed"
513                      print("[*] Comment removed: {}".format(
                             comment))
514                      graph_text = '0x{:x}'.format(int(address)) +
                             ": comment removed"
515                      verbose_graph_text = '0x{:x}'.format(int(
                             address)) + ": comment removed //" +
                             comment
516                      addNodeToGraph(graph_text, dockwidget,
                             verbose_graph_text)
```

```
517                 comment_state = function.comments
518
519             # Highlight change:
520             if (str(highlight_state) != str(function.get_instr_highlight
                    (current_addr))):
521                 print("[*] Highlight change: {} {}".format(hex(int(
                        current_addr)),
522                                                            function.
                                                                get_instr_highlight
                                                                (
                                                                current_addr
                                                                )))
523                 print("highlight_state: {}".format(highlight_state))
524                 print("get_instr_highlight: {}".format(function.
                        get_instr_highlight(current_addr)))
525
526                 color_old, color_new = color_matching(str(
                        highlight_state),
527                                                       str(function.
                                                           get_instr_highlight
                                                           (current_addr)
                                                           ))
528
529                 graph_text = '0x{:x}'.format(int(current_addr)) + ":
                        highlight"
530                 verbose_graph_text = '0x{:x}'.format(int(current_addr))
                        + ": highlight //" + str(color_new)
531                 addNodeToGraph(graph_text, dockwidget,
                        verbose_graph_text)
532
533                 highlight_state = function.get_instr_highlight(
                        current_addr)
```

```
534
535             else:
536                 print("Skipping func_updated1")
537         except Exception as e:
538             print("Exception: skipping func_update: {}".format(e))
539
540
541 def func_added(bv, function, dockwidget):
542     global eventfunc
543
544     if (eventfunc + 2 < time.time()):
545         data = OrderedDict()
546         print("[*] Function Added: {}".format(function.symbol.name))
547
548         address = function.start
549         graph_text = '0x{:x}'.format(int(address)) + ": function added"
550         verbose_graph_text = '0x{:x}'.format(int(address)) + ": function
                added //" + function.symbol.name
551         addNodeToGraph(graph_text, dockwidget, verbose_graph_text)
552
553         eventfunc = time.time()
554
555
556 def func_removed(bv, function, dockwidget):
557     global eventfunc
558
559     if (eventfunc + 1 < time.time()):
560         data = OrderedDict()
561         print("[*] Function Removed: {}".format(function.symbol.name))
562
563         address = function.start
564         graph_text = '0x{:x}'.format(int(address)) + ": function removed
```

```
            "
565         verbose_graph_text = '0x{:x}'.format(int(address)) + ": function
                removed //" + function.symbol.name
566         addNodeToGraph(graph_text, dockwidget, verbose_graph_text)
567
568         eventfunc = time.time()
569
570
571 def data_written(bv, address, length, dockwidget):
572     global eventfunc, data_state
573
574     if (eventfunc + 1 < time.time()):
575         data = OrderedDict()
576         print('[*] Data Written <0x{name:x}> {length}'.format(name=
                address, length=length))
577         data_new = bv.read(address, 1)
578         print("new data: {} old data: {}".format(data_new, data_state))
579
580         graph_text = "0x" + str(address) + ": data written"
581         verbose_graph_text = '0x{:x}'.format(int(address)) + ": data
                written //" + str(data_new)
582         addNodeToGraph(graph_text, dockwidget, verbose_graph_text)
583
584         eventfunc = time.time()
585
586     else:
587         pass
588
589
590 def type_defined(bv, name, type, dockwidget):
591     global event, eventfunc
592
```

```
593     if (event + 1 < time.time()):
594         if (eventfunc + 1 < time.time()):
595             data = OrderedDict()
596             print('[*] Type Defined')
597 
598             graph_text = "type defined"
599             verbose_graph_text = "type defined //" + str(type)
600             addNodeToGraph(graph_text, dockwidget, verbose_graph_text)
601             event = time.time()
602 
603     else:
604         print('[*] Type Defined - skipping')
605         pass
606 
607 
608 def type_undefined(bv, name, type, dockwidget):
609     global event, eventfunc
610 
611     if (event + 1 < time.time()):
612         if (eventfunc + 1 < time.time()):
613             data = OrderedDict()
614             print('[*] Type Undefined')
615 
616             graph_text = "type undefined"
617             verbose_graph_text = "type undefined //" + str(type)
618             addNodeToGraph(graph_text, dockwidget, verbose_graph_text)
619 
620     else:
621         print('[*] Type Undefined - skipping')
622         pass
623 
624 
```

```
625 def data_var_added(bv, var, dockwidget):
626     global eventfunc
627 
628     if (eventfunc + 1 < time.time()):
629         data = OrderedDict()
630         print('[*] Data_var_added')
631 
632         address = var.address
633         graph_text = '0x{:x}'.format(int(address)) + ": data var added"
634         verbose_graph_text = '0x{:x}'.format(int(address)) + ": data var
                added //" + str(var)
635         addNodeToGraph(graph_text, dockwidget, verbose_graph_text)
636 
637         eventfunc = time.time()
638 
639     else:
640         print('[*] Data Var Added - skipping')
641         pass
642 
643 
644 def data_var_removed(bv, var, dockwidget):
645     global eventfunc
646 
647     if (eventfunc + 1 < time.time()):
648         data = OrderedDict()
649         print('[*] Data_var_removed')
650 
651         address = var.address
652         graph_text = '0x{:x}'.format(int(address)) + ": data var removed
              "
653         verbose_graph_text = '0x{:x}'.format(int(address)) + ": data var
                removed //" + str(var)
```

```
654         addNodeToGraph(graph_text, dockwidget, verbose_graph_text)
655
656         eventfunc = time.time()
657     else:
658         print('[*] Data Var Removed - skipping')
659         pass
660
661 def addNodeToGraph(graph_text, dockwidget, verbose_graph_text = None):
662     global curr_node, num_branches, current_branch, branch_counter,
            undo_occurring
663
664     if not undo_occurring:
665         print("Branching")
666         if curr_node < prov_graph.number_of_nodes() - 1:
667             num_branches += 1
668             current_branch += 1
669             branch_counter = 0
670             i = 0
671             while branches[current_branch-1][i] != curr_node:
672                 branches[current_branch][i] = branches[current_branch
                        -1][i]
673                 branch_counter += 1
674                 i += 1
675             branches[current_branch][i] = curr_node
676             branch_counter +=1
677             print("new branch has " + str((i+1)) + " nodes")
678
679
680         prov_graph.add_node(prov_graph.number_of_nodes() + 1, data=
                graph_text)
681         verbose_prov_graph.add_node(verbose_prov_graph.number_of_nodes()
                + 1, data=verbose_graph_text)
```

```
682         print("current branch: ", current_branch, "branch counter: ",
                branch_counter)
683         branches[current_branch][branch_counter] = prov_graph.
                number_of_nodes() - 1
684         branch_counter += 1
685         curr_node = prov_graph.number_of_nodes() - 1
686         dockwidget.update_graph((0))
687
688 ######################################################################
689 ################### SETUP FUNCTIONS  ##################################
690 # create dictionary of address, type, name for binary
691 def func_types(bv):
692     s = []
693     tup_master = ()
694     for func in bv.functions:
695         tup_temp = (str(func)[11:-1], str(func.return_type))
696         s.append(tup_temp)
697         tup_temp2 = (str(func)[11:-1], str(func.name))
698         s.append(tup_temp2)
699
700     d = defaultdict(list)
701     for k, v in s:
702         d[k].append(v)
703     return d
704
705 def diff_func_types(a, b):
706     # Change function type
707     diff_change = 0
708     set_diff = None
709     for i in a:
710         diff = set(a[i]) - set(b[i])
711         if (len(diff) > 0):
```

```
712             diff_change = diff
713             set_diff = set(a[i])
714
715     #Change func name
716     diff_keys = a.keys() - b.keys()
717     func_renamed = diff_keys
718     #return diff_change, set_diff, func_renamed
719     return func_renamed, diff_change, set_diff
720
721 class OrderedSet(collections.Set):
722     def __init__(self, iterable=()):
723         self.d = collections.OrderedDict.fromkeys(iterable)
724
725     def __len__(self):
726         return len(self.d)
727
728     def __contains__(self, element):
729         return element in self.d
730
731     def __iter__(self):
732         return iter(self.d)
733
734
735 def type_lookup(var_type):
736     type_list = ['int16_t', 'int24_t', 'int32_t', 'char', 'void', '
        uint16_t', 'uint24_t', 'uint32_t',
737                  'float8', 'float16', 'float24', 'float', 'double', '
                     float72', 'long double',
738                  'void*', 'void* const', 'void* volatile', 'void&', '
                     int32_t*']
739
740     if (type_list.count(var_type) > 0):
```

```
741             print("found var: {}".format(var_type))
742             return True
743         else:
744             return False
745 
746 def color_matching(color_old, color_new):
747         colors_dict = {'none': 'NoHighlightColor', 'black': '
                BlackHighlightColor', 'blue': 'BlueHighlightColor',
748                        'cyan': 'CyanHighlightColor', 'green': '
                           GreenHighlightColor', 'magenta': '
                           MagentaHighlightColor',
749                        'orange': 'OrangeHighlightColor', 'red': '
                           RedHighlightColor', 'white': 'WhiteHighlightColor
                           ',
750                        'yellow': 'YellowHighlightColor'}
751 
752         _old_color = color_old.split(':')[1].split()[0][:-1]
753         _new_color = color_new.split(':')[1].split()[0][:-1]
754 
755         print(_old_color, _new_color)
756         old_color = colors_dict.get(_old_color, color_old)
757         new_color = colors_dict.get(_new_color, color_new)
758 
759         return old_color, new_color
760 
761 def setValue(bip, bv):
762         global current_addr, selChanged, current_view, var_state, func_name,
                comment_state, highlight_state, data_state, dict_funcs,
                func_type, var_list
763         valueChanged = current_addr != bip.current_addr
764         # print("{} {} {}".format(valueChanged, current_addr, bip.
                current_addr))
```

```
765     # viewChanged = current_view != bv.file.view
766     if (valueChanged):  # or viewChanged):
767 
768         print("valueChanged")
769         #var_state = bip.current_func.vars.copy()
770         update_ns(bip, bv)
771         # eventfunc2  = time.time()
772     current_addr = bip.current_addr
773     current_view = bv.file.view
774     try:
775         if (
776                 bv.file.view == "Graph:PE" or bv.file.view == "Linear:PE
                    "):  # Eventually, determine if bip is in the .text
                    section
777             var_state = bip.current_func.vars
778             func_name = bip.current_func.symbol.name
779             func_type = bip.current_func.return_type
780             comment_state = bip.current_func.comments
781             highlight_state = bip.current_func.get_instr_highlight(
                    current_addr)
782 
783             # if ('call' in bv.get_disassembly(current_addr)):
784             # get list of functions
785             dict_funcs = func_types(bv)
786             # print("-- selected a call")
787 
788         if (bv.file.view == "Hex:PE"):
789             data_state = bv.read(bip.current_addr, 1)
790     except Exception as e:
791         print("Found setValue exception {}".format(e))
792         # sys.exit()            # Close thread
793         # undefined functions have no vars
```

```
794
795 #Updates the namespace of the running kernel with the binja magic
        variables
796 def update_ns(bip, bv):
797     global current_addr, current_view
798
799     print("[*] Printing view updates!")
800     current_addr = hex(int(bip.current_addr))
801     print("Current_addr: " + current_addr)
802
803     current_view = bv.file.view
804
805     return
806
807 #Begins looking for updates in the Binja interface
808 def start_watch(bv):
809     obj = [o for o in gc.get_objects() if isinstance(o,
            scriptingprovider.PythonScriptingInstance.InterpreterThread)]
810     if len(obj) == 1:
811         bip = obj[0]
812     else:
813         raise Exception("Couldn't find scriptingprovider. Sure you are
                in the right kernel?")
814
815     setValue(bip, bv)
816     # update_ns(bip, bv)
817     threading.Timer(1, start_watch, [bv]).start()
818
819 #Adds the provenance widget to the Binja interface
820 def addStaticDockWidget(bv):
821     mw = QApplication.allWidgets()[0].window()
822     dock_handler = mw.findChild(DockHandler, '__DockHandler')
```

```
823     dock_widget = ProvenanceNinjaDockWidget.create_widget("Provenance
            Ninja", dock_handler.parent(), bv)
824     dock_handler.addDockWidget(dock_widget, Qt.RightDockWidgetArea, Qt.
            Vertical, True)
825     notification = myNotification(bv, dock_widget)
826     bv.register_notification(notification)
827     start_watch(bv)
828
829 #Sets the name of the plugin as it appears in the Binja UI
830 binaryninja.PluginCommand.register("Provenance Ninja", "Generate
        Provenance Graph", addStaticDockWidget)
```

Bibliography

1. L. Craig, "U.s. passes new cybersecurity legislation in june 2022," https://www.eccouncil.org/cybersecurity-exchange/career-and-leadership/federal-cybersecurity-laws-june-2022/, 06 2022, accessed: 2022-12-05.

2. C. on Homeland Security and G. Affairs, Jun 2022. [Online]. Available: https://www.congress.gov/bill/117th-congress/senate-bill/2520/text

3. p. 7, 2013.

4. B. Pérez, J. Rubio, and C. Sáenz-Adán, "A systematic review of provenance systems," *Knowledge and Information Systems*, vol. 57, pp. 495–543, 2018.

5. L. Carata, S. Akoush, N. Balakrishnan, T. Bytheway, R. Sohan, M. Seltzer, and A. Hopper, "A primer on provenance," *Communications of the ACM*, vol. 57, no. 5, pp. 52–60, 2014.

6. A. M. Cuzzocrea *et al.*, "Big data provenance: State-of-the-art analysis and emerging research challenges," in *CEUR WORKSHOP PROCEEDINGS*, vol. 1558. CEUR-WS, 2016.

7. W. C. Henry, "Analytic Provenance for Software Reverse Engineers," Air Force Institute of Technology, Tech. Rep., sep 2020. [Online]. Available: https://apps.dtic.mil/sti/citations/AD1108805

8. T. O. B. Odden and R. S. Russ, "Defining sensemaking: Bringing clarity to a fragmented theoretical construct," *Science Education*, vol. 103, no. 1, pp. 187–205, 2019. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sce.21452

9. P. Pirolli and S. Card, "The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis," 01 2005.

10. M. K. Tennor, "Reverse Engineering Cognition," Tech. Rep., 2015. [Online]. Available: https://apps.dtic.mil/sti/citations/AD1107807

11. G. Canfora and M. Di Penta, "New frontiers of reverse engineering," 06 2007, pp. 326 – 341.

12. D. Votipka, S. M. Rabin, K. Micinski, J. S. Foster, and M. M. Mazurek, "An observational investigation of reverse engineers' processes," *Proceedings of the 29th USENIX Security Symposium*, pp. 1875–1892, 2020.

13. M. Sikorski and A. Honig, *Practical malware analysis: A hands-on guide to dissecting malicious software by Michael Sikorski and Andrew Honig.* No Starch Press Inc, 2012.

14. "Ida pro," https://hex-rays.com/ida-pro/, accessed: 2022-05-24.

15. "Binary ninja," https://binary.ninja/, accessed: 2022-05-24.

16. "Ghidra," https://ghidra-sre.org/, accessed: 2022-05-24.

17. "x64dbg," https://x64dbg.com, accessed: 2023-01-13.

18. "Ollydbg," https://www.ollydbg.de, accessed: 2023-01-13.

19. A. R. Bryant, R. F. Mills, G. L. Peterson, and M. R. Grimaila, "Eliciting a Sensemaking Process from Verbal Protocols of Reverse Engineers," vol. 2, 2011.

20. A. Bryant, "Understanding How Reverse Engineers Make Sense of Programs from Assembly Language Representations," *ProQuest Dissertations and Theses*, p. 265, 2012. [Online]. Available: http://gradworks.umi.com/34/98/3498579.html

21. "Lighthouse - a coverage explorer for reverse engineers," https://github.com/gaasedelen/lighthouse, accessed 2022-05-26.

22. "Sourcery pane for binary ninja," https://github.com/mechanicalnull/sourcery_pane, accessed 2022-05-26.

23. "bncov - scriptable binary ninja plugin for coverage analysis and visualization," https://github.com/ForAllSecure/bncov, accessed 2022-05-26.

24. "Katai," https://github.com/Vector35/kaitai/blob/master/README.md, accessed 2022-05-26.

25. "Bncallgraph," https://github.com/borzacchiello/bncallgraph, accessed 2022-05-26.

26. C. Treude, F. Figueira Filho, M.-A. Storey, and M. Salois, "An exploratory study of software reverse engineering in a security context," 10 2011, pp. 184–188.

27. J. Baldwin, A. Teh, E. Baniassad, D. Rooy, and Y. Coady, "Requirements for tools for comprehending highly specialized assembly language code and how to elicit these requirements," *Requirements Engineering*, vol. 21, 10 2014.

28. J. Maletic and A. Marcus, "Cfb: A call for benchmarks - for software visualization," 01 2003.

29. A. I. Globus and S. Uselton, "Evaluation of Visualization Software," no. May, pp. 41–44, 1995.

30. A. Gazis and E. Katsiri, *Web Frameworks Metrics and Benchmarks for Data Handling and Visualization (PHP-Python)*, 04 2019, pp. 137–151.

31. H. Kienle and H. Müller, "Requirements of software visualization tools: A literature survey," 07 2007, pp. 2 – 9.

32. ——, "Chapter 5 - the tools perspective on software reverse engineering: Requirements, construction, and evaluation," *Advances in Computers*, vol. 79, pp. 189–290, 12 2010.

33. "Process explorer v17.02," https://learn.microsoft.com/en-us/sysinternals/downloads/process-explorer, accessed: 2023-02-01.

34. "Windows performance monitor overview," https://techcommunity.microsoft.com/t5/ask-the-performance-team/windows-performance-monitor-overview/ba-p/375481, accessed: 2023-02-01.

35. "In-depth .net memory profiling," https://memprofiler.com, accessed: 2023-02-01.

36. J. Freire, D. Koop, E. Santos, and C. T. Silva, "Provenance for computational tasks: A survey," *Computing in Science Engineering*, vol. 10, no. 3, pp. 11–21, 2008.

37. L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, and P. Groth, "The Open Provenance Model," no. August 2007, 2010.

38. E. D. Ragan, A. Endert, J. Sanyal, and J. Chen, "Characterizing Provenance in Visualization and Data Analysis : An Organizational Framework of Provenance Types and Purposes," 2015.

39. "Pyside6," https://pypi.org/project/PySide6/, accessed 2022-10-31.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
23–03–2023	Master's Thesis	Sept 2021 — Mar 2023

4. TITLE AND SUBTITLE

Improving Accessibility and Efficiency of Analytic Provenance Tools for Software Reverse Engineering through Native Visualization Support

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

2nd Lt Caleb W. Richardson

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-MS-23-M-054

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

Reverse engineering is a vital technique for identifying and mitigating cyber threats. Yet, despite its importance, reverse engineering is a time-consuming process. Provenance tools help to improve the workflow of reverse engineers by providing an accessible method of viewing their flow through a binary. The current state-of-the-art provenance tool for reverse engineering software called SensorRE, leverages an external server, web browser, and a large array of javascript libraries. This thesis presents Provenance Ninja, a software reverse engineering tool developed in Python that runs directly within Binary Ninja. Provenance Ninja captures reverse engineers' provenance data and provides an interactive graph within the reverse engineering environment. The performance of Provenance Ninja is evaluated against SensorRE by measuring functionality and efficiency. This research demonstrates that it is possible to design a provenance tool to run natively in the reverse engineering software that passes all functionality tests when compared to SensorRE and shows statistically significant efficiency improvements at a 95% confidence level in memory utilization and runtime from this approach. The results of this study contribute to the field of software reverse engineering and have the potential to enhance the effectiveness of cyber threat mitigation efforts.

15. SUBJECT TERMS

reverse engineering, provenance, Binary Ninja,

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 C. Henry, AFIT/ENG
U	U	U	UU	95	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636, ext 7243; wayne.henry@afit.edu