# Visualizing Information Flow

# Graph-Based Approach to Tracing Data Dependencies for Binary Analysis

by

Bailey Capuano

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2022 by the
Graduate Supervisory Committee:

Yan Shoshitaishvili, Co-Chair
Ruoyu Wang, Co-Chair
Adam Doupé

ARIZONA STATE UNIVERSITY

May 2022

## ABSTRACT

Binary analysis and software debugging are critical tools in the modern software security ecosystem. With the security arms race between attackers discovering and exploiting vulnerabilities and the development teams patching bugs ever-tightening, there is an immense need for more tooling to streamline the binary analysis and debugging processes. Whether attempting to find the root cause for a buffer overflow or a segmentation fault, the analysis process often involves manually tracing the movement of data throughout a program's life cycle. Up until this point, there has not been a viable solution to the human limitation of maintaining a cohesive mental image of the intricacies of a program's data flow.

This thesis proposes a novel data dependency graph (DDG) analysis as an addition to `angr`'s analyses suite. This new analysis ingests a symbolic execution trace in order to generate a directed acyclic graph of the program's data dependencies. In addition to the development of the backend logic needed to generate this graph, an `angr management` view to visualize the DDG was implemented. This user interface provides functionality for ancestor and descendant dependency tracing and sub-graph creation. To evaluate the analysis, a user study was conducted to measure the view's efficacy in regards to binary analysis and software debugging. The study consisted of a control group and experimental group attempting to solve a series of 3 challenges and subsequently providing feedback concerning perceived functionality and comprehensibility pertaining to the view.

The results show that the view had a positive trend in relation to challenge-solving accuracy in its target domain, as participants solved 32% more challenges 21% faster when using the analysis than when using vanilla `angr management`.

## DEDICATION

*Thank you to my girlfriend McKenna for her unwavering support and unique ability to ground me when the tasks and deadlines seemed insurmountable. Thank you for being my rock. I am so incredibly lucky to have you by my side, Forvie.*

## ACKNOWLEDGEMENTS

*Thank you to Adam for his invaluable support throughout this process and for sparking within me an initial passion for cybersecurity. His undergraduate information assurance course was my first real introduction to the field and, from the first assignment that had us finding ways to crash other students' programs, I was absolutely hooked.*

*I would like to thank Fish for his guidance and for putting up with my incessant Discord and Slack pings asking for programming advice as I attempted to navigate angr's codebase. No matter the question, he would somehow always have the answer.*

*Lastly, I would like to thank Yan for taking me under his guidance for this thesis and for providing me a wealth of opportunities to further hone my passion for binary analysis through his fantastic courses in computer systems security and applied vulnerability research. I can confidently say that I have never learned as much in such a short period of time as I did in his courses.*

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

# Chapter 1

# INTRODUCTION

Despite decades of advancements in the fields of software security and binary hardening, the problem of exploitation is not one of antiquity [14, 31]. The gravity of this situation can be appreciated after considering the omnipresence of low-level binary code with limited security mitigations in applications such as compilers, operating systems, and embedded systems that are driving forces in modern life [36]. Where there are binaries produced by low-level languages, there are vulnerabilities. The exposure of CVE-2021-3156, in which a heap-based buffer overflow could be exploited in the `sudo` program to gain privilege escalation to the root user, serves as the perfect reminder of the real, persistent threat of binary exploitation [15]. The threats posed by the exploitation of vulnerabilities have consequences that transcend the realm of technology. In an analysis of the impact of security breach disclosures on market value, affected firms saw an average loss of $1.65 billion in market capitalization [11]. Since low-level code saturates every level of modern life and is susceptible to vulnerabilities, the necessity for effective detection and mitigation techniques cannot be overstated.

With the ever-increasing complexity of software, the need for tools that aid in the software debugging process is becoming more and more apparent. In fact, it is common for the software testing and debugging phase to consume anywhere from half to three quarters of the total development cost [6]. Thus, the creation of debugging tools that aid in resolving the root cause of software faults would serve to reduce programmer man hours and development costs.

In an attempt to contribute to the detection and mitigation of vulnerabilities, and thus ease the burden inherent in debugging software, a novel `angr` analysis for data dependency generation that enables users with the ability to explore and interact with a visualized data dependency graph for a given binary and execution trace is presented. This analysis ingests a binary to analyze and an accompanying execution trace that is then parsed to create a representative data dependency graph containing nodes for all of the relevant memory locations and edges representing their inter-dependencies.

While researchers have applied data dependency graphs to binary analysis [35, 43], the literature has yet to see their application to visualizing dependency flows for user-driven binary analysis. This paper seeks to contribute to the scientific discourse by outlining the design of a tool that addresses this gap. As the proposed tool is focused on the end-user, whether they be a software debugger or binary analyst, I sought to explore the impact of the presented analysis on the speed and accuracy of binary analysis through a user study.

Participants were solicited from various online cybersecurity communities and tasked with working through a series of software debugging and CTF-like challenges, answering challenge questions, and providing general survey feedback in order to quantitatively and qualitatively assess the effectiveness of the proposed analysis.

After analyzing the user study results, it was concluded that the proposed data dependency analysis had a positive effect on both challenge-solving accuracy and time. The analysis had an educationally significant effect upon overall challenge-solving accuracy and time, with practical significance for challenges that required tracing data dependencies. The participant pool solved the challenges 171 seconds faster with 32% more accuracy.

# Chapter 2

# BACKGROUND

In order to contextualize and understand the analysis proposed in this thesis, some background terminology must first be defined. As the proposed analysis is not a standalone tool, it relies upon an existing framework and leverages various concepts it provides. The terminology defined in this section are all imperative to the design of the proposed analysis. The relevance of each term to this thesis will be explicitly defined.

## 2.1 angr

Shoshitaishvili *et al.* [36] described `angr` as "a binary analysis framework that integrates many of the state-of-the-art binary analysis techniques in the literature". The framework has found great success in the cybersecurity community, with it being utilized to solve a variety of problems plaguing binary analysts [17, 37–41]. Within the framework are a multitude of static and dynamic analyses. Static techniques are those that derive meaning from a program without execution [32, 33, 44], while dynamic techniques derive meaning from an actual or emulated execution of the program [10, 21, 23, 36]. Both paradigms of analyses are incredibly invaluable to the field of binary analysis as they each provide unique insights into the structure, flow, and potential vulnerabilities of a binary. The proposed data dependency graph generation analysis is an example of a dynamic analysis, as it is reliant upon an ingested symbolic execution trace.

In addition to `angr`, a graphical user interface (GUI) was designed to facilitate easier interaction with the backend and to provide comparable features to leading binary analysis framework interfaces [1, 3, 4]. This GUI is referred to as `angr management`. The analysis proposed in this paper is implemented as an angr analysis and visualized using a view integrated into `angr management`.

## 2.2 Symbolic Execution

Symbolic execution is concerned with the emulation of a program [8, 25, 36]. Rather than performing an actual execution of the program, symbolic execution is concerned with systematically exploring all of the path permutations that an execution could take. Whenever a branching decision is encountered, symbolic constraints are placed on the values in the register file and system memory involved in the control flow decision. While the symbolic execution engine must provide a wealth of functionality, such as a boolean expression representing the conditions satisfied along a given execution path, this analysis will only leverage one aspect tracked by the engine [8]. For the sake of data dependency, the fact that the emulation tracks the state of the register file and memory throughout the program's lifetime is crucial, as the concretized values in the memory store are used as the source of information for building out a data dependency graph [36].

## 2.3 Data Dependency

Data dependency maps the dependency relations between memory locations, in either the register file or within system memory, based on the data each location contains. Although, the memory location in question cannot be the sole consideration in determining a data dependency. As elucidated in the simple x86 code snippet (Figure 2.1), it would be incorrect to treat the `eax` in the fifth and seventh instructions as the same data point when determining data dependencies. While their data regions are the same, both concerning the lower 32 bits of the `rax` register, they exist in

```
1 endbr64
2 push   rbp
3 mov    rbp, rsp
4 mov    dword ptr [rbp-0x4], 0x1d6
5 mov    eax, dword ptr [rbp-0x4]
6 mov    dword ptr [rbp-0x8], eax
7 mov    eax, dword ptr [rbp-0x8]
8 pop    rbp
9 ret
```

*Figure 2.1.* Motivating example of data dependency. For example, the value of `[rbp - 0x8]` at instruction 6 is dependent upon the value of `eax` in instruction 6.

different contexts during the program's life cycle. Therefore, the point at which an instruction occurs must also be considered when determining which data points should be tracked. In this example, these memory locations can easily be differentiated by their instruction addresses.

With an understanding of the factors that must be considered in determining the data points established, the relationships between them can now be explored. A data point $D_1$ is considered dependent upon $D_2$ if the value of $D_1$ is derived from $D_2$. Primarily, this derivation is the result of a read from $D_2$ followed by a write of its value to $D_1$ (perhaps with some intermediary calculations). In the case of the motivating example (Figure 2.1), the value of `[rbp - 0x8]` at instruction 6 is dependent upon the value of `eax` in instruction 6.

## 2.4 Data Dependency Graph (DDG)

A data dependency graph is a directed acyclic graph (DAG) that tracks all the data dependencies in a given execution trace. The data points, identified by their

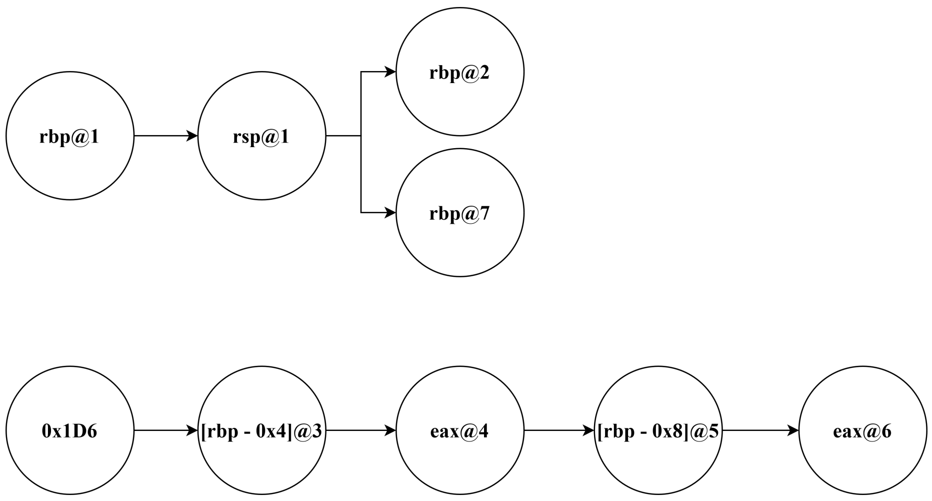


*Figure 2.2.* DDG of motivating example, node names in form of location@address. Dependencies can be traced by following a node's edges backwards to view ancestors and forwards to view descendants.

memory location and point of execution, serve as the graph's nodes. The graph's edges originate from a source node and terminate at a dependent node. From any given node, one can follow the directed edges backwards to find the origin of a node's value. By contrast, the edges leaving a node can be followed to find all dependent descendants in the data dependency graph. In the full dependency graph (Figure 2.2) of the code snippet provided in Figure 2.1, the return value can be observed to have derived from a memory address ([rbp - 0x8]) which yielded its value from another address ([rbp - 0x4]) that had the constant written into it initially (with eax@4 being used to facilitate a memory to memory move in a RISC instruction set). While a toy example, it is evident that having a graphical representation of the flow of data within a more complicated program can prove invaluable to the binary analyst or software debugger.

## 2.5 Intermediate Representation

In compiler design, a source language can be translated into an arbitrary number of instruction sets. Thus, rather than developing optimizations for every unique target architecture, compilers utilize a middle-level, architecture-agnostic data structure instead [7, 19, 27]. This data structure is referred to as the intermediate representation (IR). The source code is translated into IR before optimizations are applied and later converted into the target architecture instruction set. Just as compilers require an intermediary language to apply universal optimizations to source code, so too do decompilers such as `angr`'s. Although, in the case of decompilers, the process entails converting an arbitrary number of instruction sets back into a higher-level universal language. The intermediate representation used by `angr` is borrowed from `Valgrind` and is called `VEX` [5]. Rather than operating on machine code, which would require a tremendous amount of architecture-aware code to handle the many cross-architecture idiosyncrasies, the proposed analysis operates on `VEX` IR.

Chapter 3

# RELATED WORK

Research has seen the application of data dependency graphs spanning the field of computer science, namely code refactoring. The application of DDGs to the problem of code sequence function extraction has proven incredibly effective due to the correlation between strong data dependencies and independent functionality [24]. As source code was readily available in the problem space, the researchers were able to generate data dependency graphs utilizing source code semantics.

DDGs have even been applied to problems faced by computer architects. By applying transformations to directed, acyclic data dependency graphs, Heffernan *et al.* [20] was able to minimize data dependencies in instruction streams. The use of these transformations resulted in a significant improvement in branch prediction performance.

DDGs have proved effective in various strata of computer science. For this reason, this paper seeks to gauge the efficacy of data dependency graph generation and visualization in the realm of binary analysis. More work needs to be done to provide the level of detail required for successful binary analysis. For example, Kanemitsu *et al.* [24] visualized data dependency graphs with the code line number as the node name. While this proved effective for code extraction, this would be of little to no value to the binary analyst as source code access is rare and one line of source code can be compiled into multiple lines of assembly.

Code slicers can also be utilized to generate program dependency graphs (PDGs) to visualize the dependencies between instructions [6, 16, 18, 22]. Although, these face similar limitations to [24] in that they are source-code dependent and operate at too high of a level to be usable by the binary analyst. The use of data dependency graphs as used by compilers most closely envisions the level of detail the analysis proposed in this paper sought to produce [2, 26]. While they still map dependencies between instructions, the nodes represent the individual IR instructions as opposed to a line of source code.

Limited research has seen the application of data dependency graphs to binary analysis [35, 43]. DepGraph, an IDA plugin developed by Serpilliere [35], is capable of generating the constraints that a variable could take based off a data dependency and control dependency graph. Although, this plugin only operates on a single argument at a time and does not render the data dependency graph for the user to view. Thus, this plugin serves more as a dependency-aware constraints solver than a true data dependency graph visualization.

With a lack of existing tooling focused on generating data dependency graphs for the purpose of user-driven binary analysis, I seek to contribute to the scientific discourse through proposing a solution that learns from the many applications of DDG generation in order to deliver an effective analysis.

# Chapter 4

# DESIGN

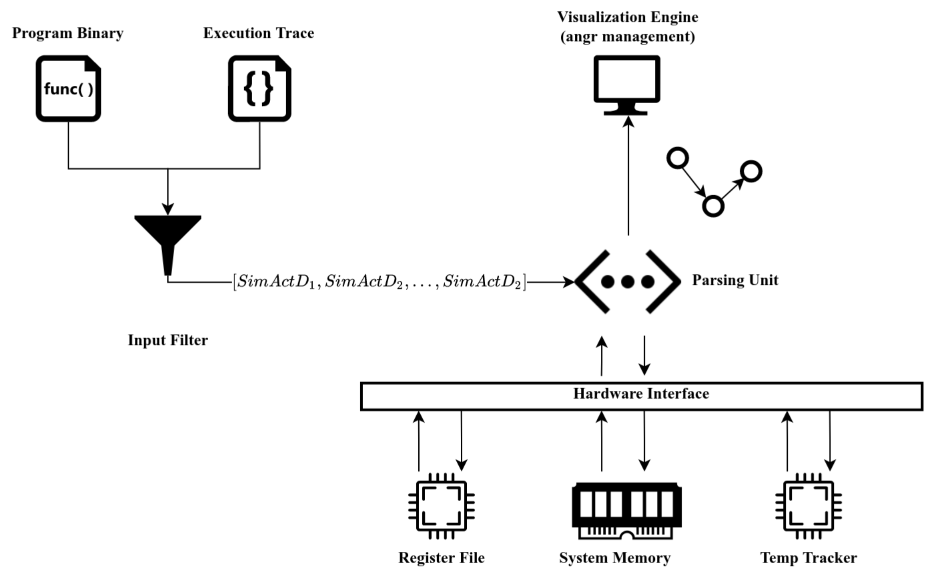


*Figure 4.1.* Design diagram of proposed analysis.

The design goal of the proposed analysis is to ingest an arbitrary binary and execution trace and efficiently parse a representative data dependency graph for visualization. In order to develop this functionality, the design was split into the independent modules seen in Figure 4.1.

## 4.1 Input Filter

This module is responsible for ingesting two critical files for data dependency graph generation: the program binary and the execution trace. The binary can be any executable file that the user would like to analyze while the execution trace must correspond to that binary. While execution trace generation was outside the scope of this thesis, the user could perform symbolic execution in `angr` upon the binary to generate it. This would typically involve creating the initial state at the first line of code that the user is interested in graphing and having the symbolic execution stop upon reaching the final line of code of interest. The resulting `SimState` must then be JSON encoded.

The input filter would then be responsible for decoding the execution trace to recover the underlying `SimState`. The only property of concern to this analysis within the `SimState` is its history, which contains an array of the read and write actions undertaken during symbolic execution. This array contains a series of `SimAction` instances, which may not all necessarily be concerned with the flow of data throughout the program's execution. The instances that belong to the `SimActionData` class are the only elements that concern data movement. Thus, this module will filter out all of the irrelevant elements from the history and output an array of the program's `SimActionDatas` in ascending order of the time in which they were taken.

## 4.2 Parsing Unit

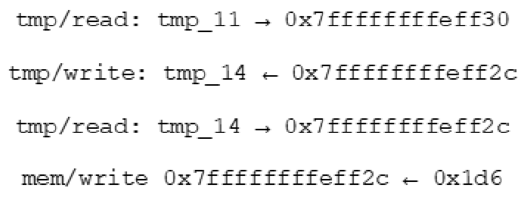
$$
\begin{gathered}
\mathrm{V} \rightarrow \{BLOCKS, INSTRUCTIONS, INSTRUCTION, STATEMENT\} \\
\Sigma \rightarrow \{WriteToVar, WriteToMem, ReadFromVar, ReadFromMem\} \\
S \rightarrow BLOCKS \\
BLOCKS \rightarrow BLOCK\ BLOCKS \mid BLOCK \\
BLOCK \rightarrow INSTRUCTION\ BLOCK \mid INSTRUCTION \\
INSTRUCTION \rightarrow STATEMENT\ INSTRUCTION \mid STATEMENT \\
STATEMENT \rightarrow WriteToVar \mid WriteToMem \mid ReadFromVar \mid ReadFromMem
\end{gathered}
$$

*Figure 4.2.* Context free grammar of the analysis' language. Designed to tightly mirror the aspects of `VEX IR` relevant to this analysis.

The parsing unit is the main module of the proposed analysis. I designed a context free grammar (Figure 4.2) to capture the structure of the linear sequence of reads and writes ingested by the analysis. A recursive descent parser processes this sequence. This grammar follows the same structure of describing CPU actions that are taken by `VEX`. This design decision was driven by the fact that the grammar in question is relatively concise, and the code required to implement a recursive descent parser tightly mirrors the grammar [34]. Furthermore, the code that drives this analysis will need continued maintenance beyond the time frame of this thesis. Moreover, keeping the parser simpler and more readable, will make future contributions and improvements easier.

The terminals in the proposed context-free grammar are used to represent the different categories of `SimActionData` that the analysis processes. As can be seen in the naming scheme of the terminals, a `SimActionData` is categorized based upon its action and type. The supported actions are WRITE and READ while the supported types are TMP, MEM, and REG. Although the MEM and REG types are straightforward, representing any operations that involve a memory address or register, the TMP type is not as readily apparent. TMP, short for temporary, operands can be viewed as variables as they are used to hold the result of intermediary steps in an operation. As an example, the AMD64 instruction `mov dword ptr [rbp { 0x4], 0x1d6` which moves the decimal value 470 into a memory offset from the base pointer, could be represented by the `SimActionDatas` presented in 4.3. In this example, two temporary variables are used to hold the value of the base pointer register and the calculated offset from rbp to facilitate the memory write.

The largest unit that the analysis is concerned with is the block, with a program being a sequence of blocks. In `VEX`, a block is a collection of instructions with a single entrance and an unbounded number of exits. The end of a block is simply delineated by the `SimActionData` currently being parsed having a different block address from the next.



```
tmp/read: tmp_11 → 0x7fffffffeff30

tmp/write: tmp_14 ← 0x7fffffffeff2c

tmp/read: tmp_14 → 0x7fffffffeff2c

mem/write 0x7fffffffeff2c ← 0x1d6
```

*Figure 4.3.* Example of temporary operands in `angr`.

An instruction, while not being part of the `VEX` language, is incorporated into the analysis' parser simply for readability. Rather than parsing the many statements that belong to a single block at a time, the statements belonging to a given instruction address are logically grouped. The end of an instruction is depicted by the `SimActionData` currently being parsed having a different instruction address from its successor.

A statement is the smallest and most important unit of analysis, as `VEX` statements are what change state in the symbolic execution. A statement is parsed according to its type and action. It is during this portion of parsing where nodes in the data dependency graph are created and linked together.

The recursive descent parser is used to convert the provided sequence of `SimActionData` elements into a data dependency graph. The data dependency graph is then forwarded to the visualization engine for rendering a visualization to the screen.

## 4.3 Hardware Interface

This module serves as a supplementary module to the parsing unit, providing it with a memory of the read and write actions parsed thus far. This is imperative to have as, without tracking the reads and writes, it isn't possible to create edges between the nodes generated by the parser. The hardware interface facilitates the interactions between the parsing unit and the simulated register file, simulated system memory, and temporary tracker. The parsing unit can request to read from or write to a given register, variable, or memory address through the functionality it exposes.

In order to track the current state of all registers that the program moves data into and out of, an emulated register file is utilized to associate the register's current value with its associated node in the graph. On a write to a register, the value and associated node are updated. On a read from a register, the current associated node is used as the value's source for linking purposes.

In order to track the current state of all memory addresses that the program reads and writes to, an emulated memory is also maintained. This works in the same manner as the register file.

The temporary variable tracker is a per-block association of values with the current temporary node. This is reset at the end of a parsed block.

## 4.4 Visualization Engine

This module is responsible for ingesting the generated data dependency graph and visualizing it in an effective user interface for the binary analyst to peruse. This was accomplished through the creation of an `angr management` view.

Chapter 5

# IMPLEMENTATION

Data dependency generation is implemented as an `angr` analysis and is registered under the name 'DataDep'. This allows the `angr` library user to access and utilize DDG functionalities through `project.analyses.DataDep()`. To maintain interoperability between DataDep and other `angr` analyses, the analysis outputs an instance of a NetworkX Digraph representing the generated DDG [28]. The generated graph can then be further operated upon by the user or visualized using standard NetworkX visualization techniques [29]

In order to generate a DDG, the analysis must be provided a symbolic execution state which acts as a source of data moves performed by the program. This state is referred to as an "end-state", as it provides a history containing all the reads and writes taken by the program up until a given endpoint. In `angr`, these simulated read and write actions are encapsulated in *SimActionDatas*. As far as `angr` is concerned, an action can be one of the types seen in Table 5.1.

As naively ingesting an end state that captures all the reads and writes performed by a large program would result in a tremendous and unwieldy number of nodes for the

Table 5.1

*Supported Action Types.*

| **Type of Action** | **AMD64 Example** |
|---|---|
| A read from a 'variable' | `mov rdx, rdi` (in terms of `rdi`) |
| A write to a 'variable' | `mov rdx, rdi` (in terms of `rdx`) |
| A read from a memory address | `mov r9, [rax]` |
| A write to a memory address | `mov dword ptr [rax], 0xdeadbeef`) |

user to parse through, the analysis supports a finer-grained approach by tailoring the graph to a portion of code of interest. This is accomplished by providing a symbolic execution state that begins its execution at the earliest point in time of concern and ends its execution on the instruction one beyond the latest point in time of concern. This "end-state" serves as the execution trace that must be provided along with the binary in question. The user may also optionally specify the block instructions or range of instruction addresses to include in the graph.

## 5.1 Dependency Nodes

The analysis operates on four distinct classes of nodes: memory, register, temporary, and constant. The first three types of nodes are used to represent a MEM, REG, and TMP `SimActionData`, respectively. The final nodal type, constant, is used to represent any untracked, literal value written or read from a `SimActionData`. As constants cannot, by definition, be dependent upon any other node. In other words, one cannot write to the decimal value 470. Thus, constants will always appear as the highest-level ancestors in a DDG. Each of these node types is represented by a corresponding class of nodes used by the analysis. The inheritance relationships between these classes, as seen in Figure 5.1 is explained in greater detail below.

### *5.1.1 Base Dependency Nodes*

The base dependency node class is abstract and serves as a template for what attributes its descendant classes must have. This class defines that, at a minimum, a node possesses a class type, instruction address, statement index, and action identifier. While the instruction address and statement index is straightforward, being the instruction of the address in which the dependency node resides and the index of the statement in that address respectively, the action warrants explanation. The action

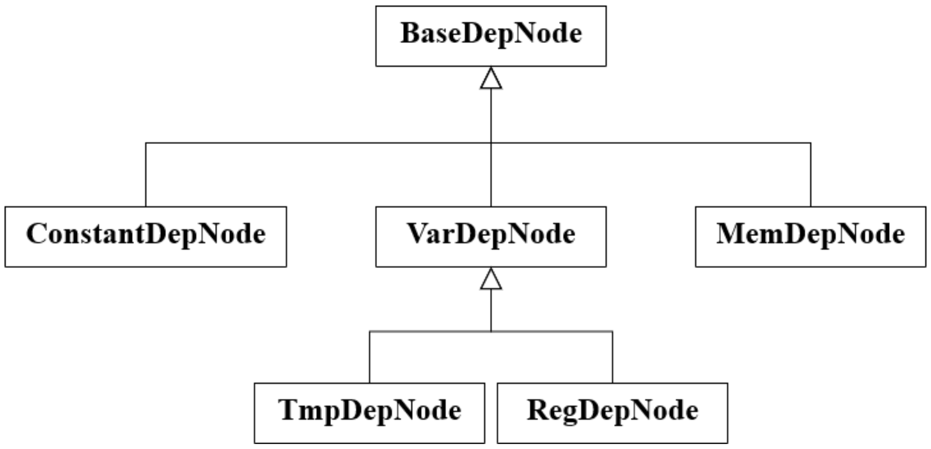


*Figure 5.1.* UML diagram of dependency node class inheritance. All nodes in a generated data dependency graph are represented by the leaf classes in this figure, with all other classes being abstract.

identifier is sourced directly from the node's corresponding `SimActionData` and is a unique counter marking the occurrence of the given action in the entire program's execution.

### *5.1.2 Constant Dependency Nodes*

The constant dependency nodes class is concrete and represented in a generated DDG. It is solely identified by its value.

### *5.1.3 Memory Dependency Nodes*

The memory dependence nodes class is also concrete and represented in a generated DDG. In addition to the attributes that identify a base dependency node, a memory dependency node is further identified by its memory address.

#### *5.1.4 Variable Dependency Nodes*

The variable dependency nodes class is abstract, serving as a common ancestor to temporary and register dependency nodes. While temporary and register nodes are different types of `SimActions`, both are identified by a register number and can be parsed without needing to be differentiated.

#### *5.1.5 Temporary Dependency Nodes*

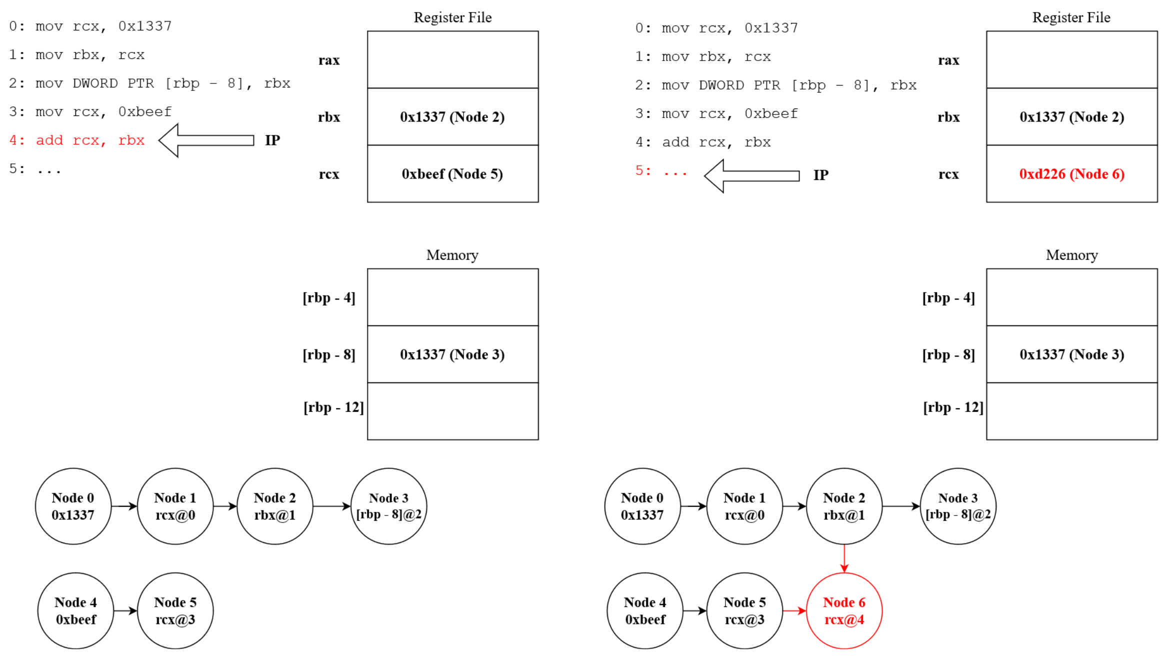
The temprorary dependency nodes class is concrete and represented in a generated DDG. While it adds no additional functionality to its base class, it is included in order to differentiate between register and temporary nodes for ease of filtering the graph for display. As the number of temporary nodes will be high for a given program, it is often beneficial to hide them in a generated DDG.

#### *5.1.6 Register Dependency Nodes*

The register dependency nodes class is concrete and represented in a generated DDG. It exists to differentiate between register and temporary nodes for the reasons proposed in Section 5.1.5

### 5.2 Generating and Tracking Nodes

Based on the type and action of the `SimActionData` currently being parsed, different fields are pulled out of the action to create the respective dependency node. Upon creating a node for an action, the action is popped from the parsing queue and the node is added to the graph. The node is originally unconnected, as it has yet to be linked to the DDG.


```
0: mov rcx, 0x1337
1: mov rbx, rcx
2: mov DWORD PTR [rbp - 8], rbx
3: mov rcx, 0xbeef
4: add rcx, rbx
5: ...
```

```
0: mov rcx, 0x1337
1: mov rbx, rcx
2: mov DWORD PTR [rbp - 8], rbx
3: mov rcx, 0xbeef
4: add rcx, rbx
5: ...
```

*Figure 5.2.* Register file and graph state before and after executing instruction 4. After executing the addition instruction, a new node for the destination operand is created and linked to the source operands. The register file for the destination operand is also updated to point to this new node.

## 5.3 Linking Nodes

The program in Figure 5.2 will be used as a motivating example to explain the linking process. The leftmost diagram gives an overview of the state of the register file, memory, and graph prior to the execution of instruction 4 and will serve as the state that is used to determine dependencies. As an add requires the CPU to read the value from the source operand and target operand, nodes 2 and 5, which are currently associated with `rbx` and `rcx`, respectively, will be located in the register file and tracked as data sources. Once the addition is calculated by the ALU, the value is then written back to the target operand (`rcx`). To facilitate this in the analysis, node 6 is created for `rcx` at this state and is linked to its two tracked source nodes as a

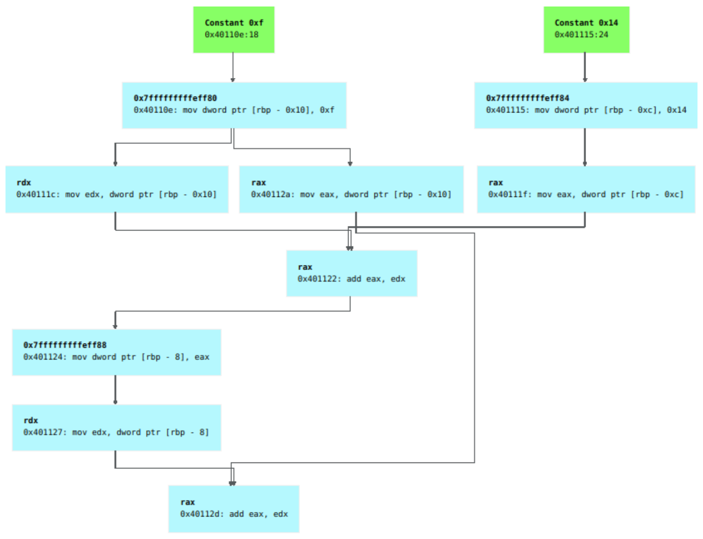

Constant 0xf
0x40110e:18
Constant 0x14
0x401115:24
0x7fffffffffeff80
0x40110e: mov dword ptr [rbp - 0x10], 0xf
0x7fffffffffeff84
0x401115: mov dword ptr [rbp - 0xc], 0x14
rdx
0x40111c: mov edx, dword ptr [rbp - 0x10]
rax
0x40112a: mov eax, dword ptr [rbp - 0x10]
rax
0x40111f: mov eax, dword ptr [rbp - 0xc]
rax
0x401122: add eax, edx
0x7fffffffffeff88
0x401124: mov dword ptr [rbp - 8], eax
rdx
0x401127: mov edx, dword ptr [rbp - 8]
rax
0x40112d: add eax, edx

*Figure 5.3.* Visualization of a simple data dependency graph. This screenshot is of a subgraph that traces two subsequent additions.

dependency. As the next operation that utilizes `rcx` as an operand should be reliant upon the value of `rcx` post-addition, the register file for `rcx` is updated to point to node 6.

## 5.4 Visualizing Nodes

In addition to implementing the backend analysis for generating a data-dependency graph, its visualization was also within the scope of this thesis. This was accomplished through contribution to `angr management`, the official frontend for `angr`. An example visualization of a simple data dependency graph is depicted in Figure 5.3.

To improve the user experience, various features were implemented to aid the user in more quickly resolving the dependency of instructions with which they are concerned. For example, search functionality is available to jump through all nodes that match a provided name, instruction address, and/or value. Any of these fields can be omitted in the search, and thus will not be considered in node filtering. In addition to this feature, temporary nodes can be toggled on and off to prevent the screen from being cluttered with useless information when the temporary variables are not relevant to the analysis More importantly, however, the most relevant control is the generation of subgraphs.

### *5.4.1 Subgraphs*

A subgraph can be generated from any given node in the data dependency graph. That is, the user can specify to trace the dependency of node X "forwards" and view all nodes that are ancestors of X or trace "backwards" and view all descendants of X. This feature is especially helpful in decreasing screen clutter, as the majority of nodes on a graph will not be relevant to a given dependency trace.

Chapter 6

# EVALUATION

The goal of the user study is to determine the effectiveness of the data dependency graph analysis and its `angr management` visualization with respect to software debugging and binary analysis. In order to quantitatively and qualitatively determine its effectiveness, a user study was designed in which participants were tasked with solving a series of software debugging and binary analysis challenges. The experiment was split into three phases: introduction, challenge-solving, and survey. During the introduction phase, participants were given an overview of `angr management` and the views they were allowed to utilize during the experiment. The challenge-solving phase entailed the participants working through the challenges and answering a series of questions concerning each challenge. After all challenges were completed, the participant would then enter the survey phase where a series of survey questions were administered that captured the participant's relevant background and overall perceptions of the experiment.

## 6.1 Control vs Experimental Groups

Participants were randomly divided into a control and experimental group as determined by a random session key that was used to de-identify user data. A control group was introduced as a means of measuring baseline performance in the challenge-solving phase in comparison to the results of the experiment group in order to gauge effectiveness. Those participants who were assigned to the control group were only provided an overview of `angr management`, while those who were assigned to the experimental group were provided an additional overview of the functionality of the data dependency graph view. During the challenge-solving phase, the experiment group was allowed to utilize the data dependency view as an aid in analyzing the challenge binary. This feature was disabled for the control group.

## 6.2 Challenges

In regards to experiment duration, three challenges were designed for this experiment: two focused on software debugging and one similar to traditional CTF challenges which focused on binary analysis more generally. In the subsequent sections the challenges are described in further detail.

- *Median*: This software-debugging challenge finds the median of nine numbers by means of four successive calls to a median function. The first three calls were responsible for finding the median of the trisected sub-arrays, with the final call finding the median of the previous three results. Although, a logical comparison error in the median implementation causes the second call to the function to return an incorrect median, cascading to an incorrect result from the final call as well. The user was tasked with determining which call to median resulted in the bug and the nature of the bug.

- *Follow*: This challenge was designed to emulate a traditional CTF challenge, with the correct input printing a flag. In order to determine the correct input, the participant would have to follow a data dependency maze, following the value 0x1337 as it moves from register `rbx` to register `rax` through a complicated series of register shuffling with many dead-ends. If the user-specified path was correct, the flag would be printed to standard output. The user was asked to provide details about what they attempted during the solving time, if they were able to solve the challenge, and what their inputs to the program were.

- *Notes*: This software-debugging challenge asked the user to resolve a mock user bug report in a note-taking application. This binary allows the user to create, read, edit, and delete notes. However, a failure to check for the existence of a note before dereferencing it in the read functionality would cause a null-dereference and segmentation fault. The user was asked to identify the source of the bug and detail its nature.

## 6.3 Framework

In order to ease the burden inherent in installing `angr` and recording their own data, participants were provided a cloud-based system which enables interaction with a pre-configured virtual machine. After being provided a session key, the user would be able to navigate to the experiment's domain and begin the experiment at the time of their choice. The framework would walk the participant through a sequence of pages that served as a guide for the user's session. After getting consent from the user, he or she would be presented with a general introduction to the study. After navigating to the next page, a pre-configured virtual machine accessible via RDP would be cloned and powered on for the user to solve the challenges on. This framework provided a reliable means of recording the user's progress on the virtual machine and precise timestamps of how long each challenge took each user. While the system pre-dates the experiment, various customizations were made in order to protect user data and introduce randomization.

Randomness was key to eliminating bias and uncontrolled variables in this experiment. First, participants were randomly assigned to the control group, unable to use the DDG, or experiment group. The former enabled the experiment to have a measurable baseline. Secondly, the order of challenges was randomized per participant to eliminate the impact of progressive learning and fatigue on the later challenges.

Table 6.1

*Permitted Views.*

| **Control** | **Experimental** |
|---|---|
| Functions | Functions |
| Disassembly | Disassembly |
| Hex | Hex |
| Strings | Strings |
| Interaction | Interaction |
| Console | Console |
| Log | Log |
|  | Data Dependency |

To achieve this randomness, the user's unique session key was used as the seed to a random number generator that decided these factors. Once the sequence was shuffled, the survey pages associated with each challenge were dynamically shuffled to match the new order. While the participation website was easily randomized, the challenge of syncing `angr management` on the virtual machine to utilize the same randomness proved more complex.

## 6.4 Custom `Angr` Wrapper

The design of the experiment required modifications to `angr management`, namely support for restricting access to views and loading binaries in a pre-determined sequence. For the sake of this experiment, the views in Table 6.1 were permitted. Noticeable omissions include decompilation and symbolic execution. These were removed to address the inherent challenge of designing challenges for a ten-minute time frame. They must be simple enough to be solved within the time frame but complicated enough to warrant analysis. These challenges would be trivial with access to decompilation and symbolic execution. If included, the experiment would be invalidated. Rather, pitting data dependency against disassembly view, which it seeks to complement, makes more sense for the scope of this project.

The greater challenge occurred when syncing the order of loaded binaries and trace files with the web server. As some form of communication of the randomly generated group membership and challenge order needed to be transmitted from the server to the virtual machine, a binary-encoded JSON file was written to the virtual machine upon creation utilizing the `VirtualBox` command execution functionality. When the customized `angr` wrapper was launched by the participant, it would first check for this file and reorient the order in which it loads binaries to sync with the server.

## 6.5 Survey

After completion of each challenge and its associated questionnaire, the user was asked to provide more general feedback by means of a final survey. The participants were asked questions pertaining to their experience in software debugging and vulnerability analysis, their comfort with `angr management`, and their overall understanding of the challenges provided. While these questions were asked of all participants, further questions were asked dependent upon group membership. For those users in the control group, an example data dependency graph screenshot was provided. Questions about the perceived usefulness and clarity of the 'hypothetical' view accompanied the screenshot. Contrarily, users in the experiment group were asked to rate data dependency graph view in terms of its usefulness and clarity. Additionally, a free-form further feedback field was provided to ascertain the user's thoughts regarding future improvements that could be made to the view.

## 6.6 Collected Data

In addition to the user's responses to the challenge questionnaires and survey, the time a user took to complete each challenge was recorded. This data will be used to quantify a performance delta between control and experiment groups in terms of challenge-solving speed. Lastly, the user's virtual machine session was video recorded as a means of verifying user's participation and to resolve any possible bugs or crashes should they have occurred. No personally identifiable information was tracked or stored in the server's database. Instead of utilizing a name or email, data is tethered to a session by the random, unique session key provided to each user.

# Chapter 7

# RESULTS

## 7.1 Participants

Participants were sourced primarily from a variety of cybersecurity-oriented Discord servers, as members in these groups would have a higher chance of having the requisite skills to complete the experiment. Initially, members were primarily recruited from Discord servers created for various undergraduate and graduate cybersecurity courses offered at Arizona State University. Once this pool was depleted, recruitment messages were sent to various reverse-engineering oriented Discord servers with members from around the globe. These servers provided a sufficient pool of enthusiastic participants.

A total of 78 session keys were sent out to individuals who had expressed interest in participation, whether through email or by responding to the Discord recruitment messages. To incentivize participation, participants were awarded $50 for completion of the entire study. As a result, there was a turnout of 42 participants who fully completed the study. An additional 13 participants had their data thrown out, as they either partially completed the experiment or were unable to participate due to RDP latency or other technical difficulties.

Of the participants who fully completed the experiment, 19 had less than 2 years of experience in software debugging and 23 had 2+ years. As for vulnerability analysis experience, 30 had less than 2 years of experience and 12 had 2+ years. Participants with less than 2 years of experience in a subject will be referred to as being ‘inexperienced’ while those with 2+ years will be referred to as ‘experienced’.

## 7.2 Challenge Correctness

Table 7.1

*Aggregated Challenge Results.*

| | **Control** | | **Experimental** | |
|---|---|---|---|---|
| Challenge | Pass | Fail | Pass | Fail |
| Median | 7 | 14 | 5 | 16 |
| Follow | 7 | 14 | 14 | 7 |
| Notes | 11 | 14 | 14 | 7 |

Table 7.2

*Significance of Challenge Correctness.*

| Challenge | P-value | Cohen's $d$ | Effect Size[1] |
|---|---|---|---|
| Median | 0.753 | -0.207 | Small Negative & Somewhat Educationally Significant |
| Follow | 0.015 | 0.69 | Moderate & Practically Significant |
| Notes | 0.173 | 0.287 | Small & Educationally significant |

[1] Using effect size descriptors by Cohen [13] & Wolf [42].

Each participant's challenge questionnaire was scored on a binary scoring system, with 1 point being awarded for the correct answer and 0 for the incorrect answer. As the same grading criteria was applied to control and experimental submissions, the opportunity for biased grading was eliminated. Table 7.1 summarizes the results of the challenge-solving portion of the experiment. The experimental group saw better performance when solving the follow and notes challenges. In fact, a 100% improvement was seen in solve percentage between the control and experiment group in the follow challenge. This was determined to be a statistically significant difference, showing that data dependency view is a contributor to improved performance in regards to this challenge. However, there were not enough samples or variance to determine statistical significance for the notes and median challenges. Due to the marginal differences between notes and median solves, the P-value of correctness between the groups is 0.076. If the standard 95% confidence interval is used, this is just barely outside the range required to reject the null hypothesis. Within this specific participant pool, the control group was more likely to solve the median challenge. This unexpected result could be due to the wording of the median questionnaire, as it proved confusing to participants. Thus, this marginal difference could be explained by a poorly designed challenge with poor questions. The fact that follow, a challenge based on making sense of a complicated chain of data dependencies, saw a significantly higher solve-rate amongst the experiment group while notes and median saw marginal improvements and degressions speaks to the reality that data dependency graph analysis is a specialized tool. While it is provably effective in the specific domain of resolving data dependencies, improvement gains are less dramatic in other realms of binary analysis. Due to both the ten minute attempt suggestion given to challenges and the need to keep challenges feasibly solvable within that time frame, it is difficult to capture an element of data dependency in every challenge.

Unsurprisingly, users with more experience in software debugging performed better on the software debugging challenges: median and notes. Of the 5 solves for median in the experimental group, 4 came from experienced software debuggers. As for notes, 8 of the 14 solves came from participants with a strong debugging background. This shows that data dependency graph analysis is an effective software debugging tool that can be judiciously applied by the experienced software debugger. To further support this notion, follow, a challenge without elements of software debugging, saw the same performance among inexperienced and experienced software debuggers. Similarly, experience with vulnerability analysis had no discernible impact on challenge correctness. This shows that data dependency graph should not be advertised as a vulnerability analysis tool.

## 7.3 Completion Time

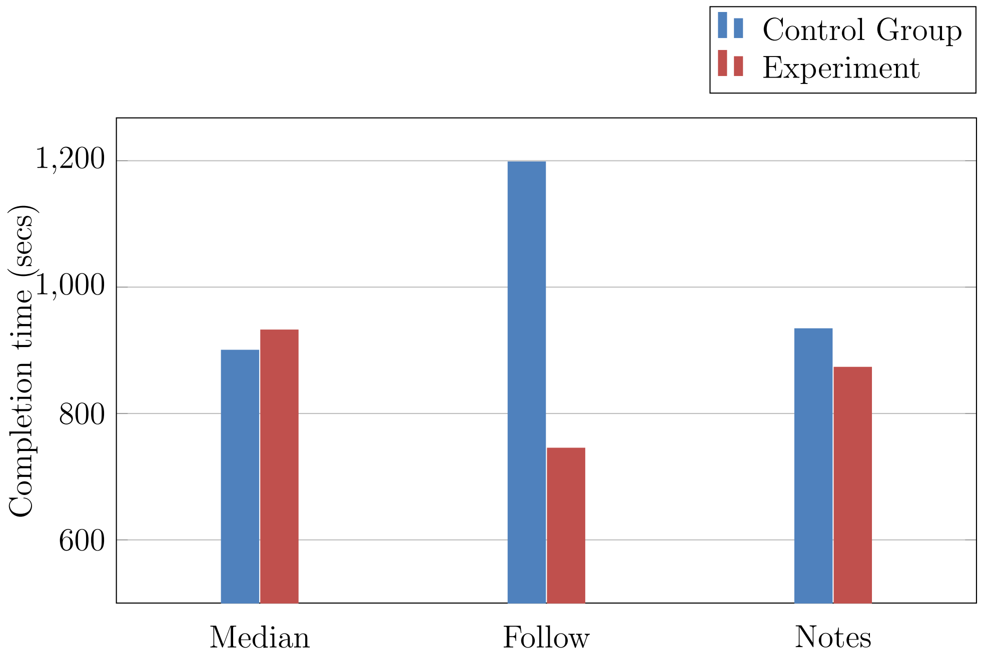


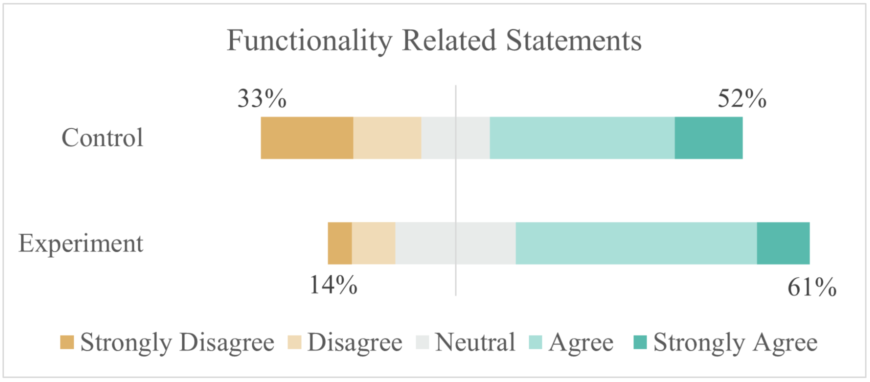
*Figure 7.1.* Completion time results.

The average completion time of each challenge was computed for the experiment and control groups. Only participants who were able to get the correct answer were incorporated into these calculations, as the speed it takes a participant to derive an incorrect answer is irrelevant. As many participants were unable to get the correct answer, fewer data points were able to be considered. The only conclusive statistical result was, again, in the follow challenge. When using a 95% confidence interval, the null hypothesis stating that data dependency view resulted in faster solves on average, could not be rejected with a P-value of P=0.132. Although, with a Cohen's *d* of 0.309, data dependency view had a small and educationally significant impact on solve times. Furthermore, the results within the participant pool show a trend for solving challenges faster using data dependency graph. The only challenge that saw poorer performance with data dependency graph was median, which could be attributed to the extremely small sample size of participants who were able to solve the challenge. Despite the experiment group solving challenges 171 seconds faster on average, no statistical significance can be determined with the data. This could possibly be attributed to the user's being provided a ten minute timer for challenge-solving, which eliminates the majority of the possible variance.

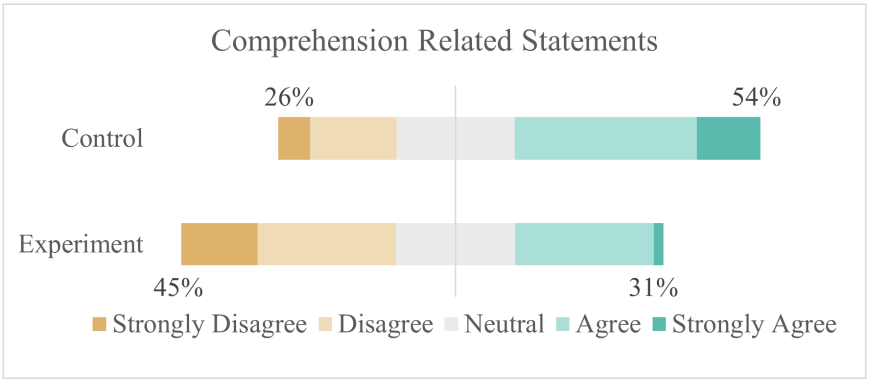
Experienced software debuggers in the experimental group saw over a 200% speedup in completion time for solving median, cementing the importance of software debugging knowledge in solving this challenge. The two groups had a Cohen's $d$ of 0.753 for this challenge, meaning data dependency view had a medium and practically significant effect for this challenge. No correlation between software debugging expertise and completion time was exhibited by the other two challenges. As for experience with vulnerability analysis, those with experience saw a 185% speedup in completion time for solving notes. This may be attributed to the fact that the challenge was modeled after a common CTF challenge format, which was commented on in many participant's feedback for that challenge.

## 7.4 User Perception

In addition to the quantitative data that was gathered, participants were also asked to provide their general feedback at the end of the experiment. The full list of survey questions can be seen in Appendix B. Figures 7.2a and 7.2b provide a summary of the user's perception. Questions were aggregated into two groups: comprehension related (questions 7&8 for control and 7&13-14 for experiment) and functionality related (9 for control and 8-12 for experiment). User feedback communicates that the experimental group generally viewed the functionality of the view favorably, but



*(a)*



*(b)*

*Figure 7.2.* Aggregated participant agreement with (a) functionality and (b) comprehension related statements.

viewed its comprehensibility negatively. This communicates that the analysis itself is a useful addition to the suite of `angr` analyses, but the user interface requires rework to be more in line with user expectations.

In addition to asking the experimental group for feedback on the data dependency view, users in the control group were asked two questions about a "hypothetical" data dependency view in the post-participation survey. The control group was given an example screenshot of a simple data dependency graph, depicted in Figure 2.2, and asked if they thought that the hypothetical view would be helpful. They were then subsequently asked if they thought the view would be confusing and unnecessary. Of the control-group participants who chose to respond to these survey questions, 62% considered the proposed view as helpful and 24% unhelpful. 43% of users thought that the data dependency graph depicted in the photo was clear while 29% found the view confusing. This demonstrates a desire for more tooling in `angr` among the control group and that disassembly view was not sufficient in solving the challenges.

# Chapter 8

# DISCUSSION

## 8.1 Lessons Learned

Throughout the course of this thesis, an abundance of lessons were learned with respect to designing a user interface tailored to the user's experience and a user study able to capture the efficacy of the DDG. The lessons outlined in this section will serve as the motivation for potential future work.

The results of the survey reveal that a sizable portion of the study-base found the user interface confusing or lacking in desired functionality. Largely, the suggested functional additions and improvements promote better interaction between the user and the generated data dependency graph. Asking the user for a trace file and displaying the entire data dependency graph is not in line with the user's expectations or interests. Rather than displaying the entire graph at once, with all the nodes in the provided trace file's history and expecting the user to utilize the search functionality to navigate, it would be better to allow the user to filter down the graph prior to displaying it. The analysis' backend currently supports more fine-grained approaches to DDG generation but are not reflected by endpoints in the user interface. Possible improvements will be described in the future work section.

As for the design of the user study, the use of randomization had its drawbacks. While instrumental in eliminating the effects of fatigue and progressive learning, the randomization of challenge order without respect to challenge difficulty proved problematic. While the effect of progressive learning was eliminated, this proved counter-intuitive when the user was randomly assigned one of the more difficult challenges first. In their feedback, some participants noted that they would have appreciated starting with one of the later challenges first to better learn data dependency view prior to applying it to a more difficult challenge. A possible solution that would still eliminate the effect of progressive learning but also resolve the issue of users facing more difficult challenges first would be to have each challenge scale its difficulty to correlate to its randomized order in order to maintain an easy → medium → hard progression.

In addition to the effects inherent in randomization, whichever challenge the user was initially assigned would have a completion time swayed by factors excluding difficulty. As many participants did not have prior experience with `angr management`, asking them to learn the basics of interacting with the framework, the novel view, and solving a challenge all at once had a definite sway on the time it takes to complete the first task.

After analyzing the results, the importance of wording in questions and objectivity were understood to be paramount. Questions concerning the user's self-perceived skill level was not statistically tied to results, and thus those types of questions proved useless. Furthermore, the wording of the median questions proved confusing to some participants. This required more lenient grading that took into account the user's free-form feedback as opposed to grading their multiple choice answers exclusively.

As evidenced by feedback asking for functionality that was already explained on the introductory page, the medium by which the introductory lesson was delivered was not effective. Research has shown that product manuals are not an effective means of conveying information to users. Most people do not read instruction manuals and will utilize a fraction of a product's features and functionality [9]. Rather than presenting the introduction as a static webpage, a video would have been more engaging and provided users with a clearer demonstration of the features at their disposal. The decision to convey feature implementation had a serious negative effect on user perception of the data dependency view, with users requesting zoom functionality, subgraph functionality, search functionality, and node highlighting. All of these features were explained in the introduction.

The largest lesson learned was the downsides of deploying this experiment on a cloud-based framework. While the use of a cloud-based solution allowed for participants from the wider cybersecurity community as well as more flexibility with the participant's schedules, many users reported latency as a discomfort during the challenge-solving phase. While already being a network-intensive protocol, RDP becomes far more unusable with distance. Many interested participants from Europe and Asia were unable to complete the experiment due to the unbearable latency. The latency inherent to RDP even had a negative impact on user feedback in regards to the views comprehensibility. As evident from the free-form feedback, two participants found the view confusing due to the lag they encountered while using data dependency view. This will have had an effect on the data, despite being an independent variable. Future studies would have to weigh this consideration into account when considering experiment deployment methods. In addition to relying on a server to conduct this experiment, participation was very much at the mercy of the server's uptime and performance. Early in the user study, the server suffered from thrashing and would shut down under high load. Prior to this error being resolved, various participants would either have to have their results thrown out or were discouraged from continuing. Thus, if a cloud-based solution were to be utilized in a future user study, more servers would need to be deployed globally, clustered by participant locality. Having multiple servers would also serve as fault-tolerance, with one malfunctioning server offloading its work onto its peers to prevent interruption of participation.

## 8.2 Future Work

As evidenced by the feedback received and discussed in the lessons learned, work must be done to improve the user interface. Possible improvements and reworks, as aggregated from participant feedback, will now be described in greater detail.

About 1/5 of users reported a desire for a more cohesive experience when using data dependency graph. Rather than loading a trace file for the current binary, these users stated they would prefer the option to right-click the constants or instructions in the disassembly view that they would like to generate a data dependency graph of. The current implementation could easily be pivoted to accommodate this modality of interaction. A symbolic execution state could be initialized at the earliest instruction in the user-specified selection and allowed to run, with a find target of the end instruction. Should there be a path that exists, the state at the end of symbolic execution could be passed to the data dependency analysis.

While the aforementioned solution will work, symbolic execution is an expensive operation. If the user were to only use data dependency on one or two instruction subsets during their analysis, then no further optimizations are necessary. Although, should the user wish to generate many DDGs, the lag between the start of the task and the generation of the view would be an annoyance. A possible solution would be to move the data dependency generation to a background thread that, upon completion of the underlying analysis, emits a Qt signal to inform the main thread that it can now generate the view. While the same amount of time would be required to generate the view, the UI would not be locked until completion and the user would be free to continue their analysis elsewhere.

Another possible optimization could see the generation of a DDG being queued upon the user's first request to see one. Regardless of the instruction subset requested, an initial request could see the data dependency analysis being dispatched in a background thread for the entire program-space. While the first data dependency graph requested would be slower, all subsequent calls could just utilize the subgraph functionality and be near instantaneous. Although, this solution has many trade-offs: increased computation that may go unutilized, more utilization of RAM to store the larger Digraph, and some binaries being far too large to warrant a DDG being generated for the entire program-space.

User feedback suggested that the current user interface provides a jarring experience when attempting to manually follow a dependency in the graph. For one, the scrolling speed resulted in users losing their place in the graph. Additionally, users reported a desire for color-coding certain paths branching out of a node to make manual tracing more feasible. This problem is compounded by the clutter produced by the arrows in the graph which should also be addressed.

The last category of improvements suggested by the users was improvements to searching. The search panel should allow users to specify if they would like to search within the current sub-graph or within the entire data dependency graph, as a few users voiced their desire for this functionality. Users also requested the ability to search for instructions using regex.

Future work on data dependency graph should be focused on bridging the divide between data dependency graph and disassembly view / decompilation view. The ability to more quickly switch between the control flow graph and the data dependency would address the majority of feature requests from the participant pool.

Another promising vein of research for inclusion in data dependency graph is the application of taint analysis in deciding how graphs should be generated [12, 30, 45]. The application of taint analysis could make data dependency graph more relevant in the realm of vulnerability analysis, as a correlation could not be established for the view proposed in this experiment. With the ability to specify sources (user-controllable origins of data) and sinks (potentially vulnerable instructions or functions) a data dependency graph could be generated that shows any existing connections between the source and the sinks. With the ability to trace forwards from the source to a sink or backwards from a sink to the source, the user would be presented with the series of checks and input restrictions required to exploit the vulnerable code in question. Furthermore, breadth-first search could be employed to find the shortest path from a given source to a given sink if multiple dependency paths exist (to make exploitation easier). Presenting the shortest path as opposed to the entire data dependency graph would also help prevent overloading the user with a massive dependency tree.

Chapter 9

# CONCLUSION

Two contributions were made through this paper. First, a data dependency graph analysis was introduced as a new means of analyzing binaries in the `angr` ecosystem. By employing a recursive-descent parser and a simulated register file and dedicated memory, the analysis is able to track dependencies between data regions throughout a program's execution. The subset of instructions analyzed can be altered through fine-grained controls that allow the user to specify a subset of executed instructions for graph generation. This data dependency graph can also be visualized through a custom `angr management` data dependency view.

Secondly, the efficacy of this view was tested through a user study that utilized a customized `angr` wrapper and a cloud-based deployment framework. In the experiment, users were split into a control and experimental group and were asked to solve a series of software-debugging and more generalized binary analysis challenges. The results showed an overall increase in the number of challenges correctly solved amongst the experimental group, with a statistically significant increase involving the challenge concerning data dependencies. As for time to solve, participants in the experimental group were able to solve the challenges much faster than their peers in the control group. No statistical significance could be attributed to this observation. Users perceived the view to be incredibly functional, but found its comprehensibility challenging. Based off of user feedback, a numerous amount of lessons learned were proposed with possible future work detailed.

The implementation of data dependency graph in its current state, while showing promising results in its niche, is just the beginning of the possible impact this view could have on binary analysis speed and accuracy. Should the suggestions posed in the future work be adopted, the view could see much more impressive results in future user studies. It is hoped that the user study design proposed in this paper should serve as a template for any future studies in this vein in order to continue to enrich the suite of tools available to the software debugger and binary analyst of the future.

## REFERENCES

[1] “Binary ninja”, https://binary.ninja/, accessed: 2022-04-18 (2022).

[2] “Dependence graphs in llvm”, https://llvm.org/docs/DependenceGraphs/index.html, accessed: 2022-04-18 (2022).

[3] “Ghidra”, https://ghidra-sre.org/, accessed: 2022-04-18 (2022).

[4] “Ida pro”, https://hex-rays.com/ida-pro/, accessed: 2022-04-18 (2022).

[5] “Intermediate representation”, https://docs.angr.io/advanced-topics/ir, accessed: 2022-04-18 (2022).

[6] “Slicing”, https://www.frama-c.com/fc-plugins/slicing.html, accessed: 2022-04-18 (2022).

[7] Aho, A. V., *Compilers: principles, techniques and tools (for Anna University), 2/e* (Pearson Education India, 2003).

[8] Baldoni, R., E. Coppa, D. C. D’elia, C. Demetrescu and I. Finocchi, “A survey of symbolic execution techniques”, ACM Computing Surveys (CSUR) **51**, 3, 1–39 (2018).

[9] Blackler, A. L., R. Gomez, V. Popovic and M. H. Thompson, “Life is too short to rtfm: how users relate to documentation and excess features in consumer products”, Interacting with Computers **28**, 1, 27–46 (2016).

[10] Buhov, D., R. Thron and S. Schrittwieser, “Catch me if you can! Transparent detection of shellcode”, in “Software Security and Assurance (ICSSA), 2016 International Conference on”, pp. 60–63 (IEEE, 2016).

[11] Cavusoglu, H., B. Mishra and S. Raghunathan, “The effect of internet security breach announcements on market value: Capital market reactions for breached firms and internet security developers”, International Journal of Electronic Commerce **9**, 1, 70–104 (2004).

[12] Clause, J., W. Li and A. Orso, “Dytan: a generic dynamic taint analysis framework”, in “Proceedings of the 2007 international symposium on Software testing and analysis”, pp. 196–206 (2007).

[13] Cohen, J., “Statistical power analysis for the behavioral sciences (revised ed.)”, (1977).

[14] Cowan, C., C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang and H. Hinton, “Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks.”, in “USENIX security symposium”, vol. 98, pp. 63–78 (San Antonio, TX, 1998).

[15] CVE-2021-3156, "CVE-2021-3156.", Available from MITRE CVE database, CVE-2021-3156., URL `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156` (2021).

[16] Ferrante, J., K. J. Ottenstein and J. D. Warren, "The program dependence graph and its use in optimization", ACM Transactions on Programming Languages and Systems (TOPLAS) **9**, 3, 319–349 (1987).

[17] Flack, M., R. Foster and S. Xu, "Graph convolutional network for classifying binaries with control flow graph data", (????).

[18] Gallagher, K. B., *Using program slicing in software maintenance*, Ph.D. thesis, University of Maryland, Baltimore County (1990).

[19] Grune, D., K. Van Reeuwijk, H. E. Bal, C. J. Jacobs and K. Langendoen, *Modern compiler design* (Springer Science & Business Media, 2012).

[20] Heffernan, M., K. Wilken and G. Shobaki, "Data-dependency graph transformations for superblock scheduling", in "2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)", pp. 77–88 (IEEE, 2006).

[21] Hernandez, G., F. Fowze, T. Yavuz, K. R. Butler *et al.*, "Firmusb: Vetting usb device firmware using domain informed symbolic execution", ACM Conference on Computer and Communications Security (2017).

[22] Higo, Y. and S. Kusumoto, "Enhancing quality of code clone detection with program dependency graph", in "2009 16th Working Conference on Reverse Engineering", pp. 315–316 (IEEE, 2009).

[23] Honig, J., "Autonomous exploitation of system binaries using symbolic analysis", (2017).

[24] Kanemitsu, T., Y. Higo and S. Kusumoto, "A visualization method of program dependency graph for identifying extract method opportunity", in "Proceedings of the 4th Workshop on Refactoring Tools", pp. 8–14 (2011).

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

[26] Kuck, D. J., R. H. Kuhn, D. A. Padua, B. Leasure and M. Wolfe, "Dependence graphs and compiler optimizations", in "Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages", pp. 207–218 (1981).

[27] Muchnick, S. *et al.*, *Advanced compiler design implementation* (Morgan kaufmann, 1997).

[28] NetworkX, "NetworkX digraph – directed graphs with self loops", `https://networkx.org/documentation/stable/reference/classes/digraph.html`, accessed: 2022-04-01 (2022).

[29] NetworkX, “NetworkX drawing”, `https://networkx.org/documentation/stable/reference/drawing.html`, accessed: 2022-04-01 (2022).

[30] Newsome, J. and D. X. Song, “Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software.”, in “NDSS”, vol. 5, pp. 3–4 (Citeseer, 2005).

[31] Pappas, V., M. Polychronakis and A. D. Keromytis, “Transparent {ROP} exploit mitigation using indirect branch tracing”, in “22nd USENIX Security Symposium (USENIX Security 13)”, pp. 447–462 (2013).

[32] Parvez, M. R., *Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries*, Master’s thesis, University of Waterloo (2016).

[33] Redini, N., A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel and G. Vigna, “BootStomp: On the security of bootloaders in mobile devices”, (2017).

[34] Redziejowski, R. R., “Parsing expression grammar as a primitive recursive-descent parser with backtracking”, Fundamenta Informaticae **79**, 3-4, 513–524 (2007).

[35] Serpilliere, “Data flow analysis: Depgraph”, `https://miasm.re/blog/2017/02/03/data_flow_analysis_depgraph.html` (2017).

[36] Shoshitaishvili, Y., R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, “Sok:(state of) the art of war: Offensive techniques in binary analysis”, in “2016 IEEE Symposium on Security and Privacy (SP)”, pp. 138–157 (IEEE, 2016).

[37] Stephens, N., J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution.”, in “NDSS”, vol. 16, pp. 1–16 (2016).

[38] Taylor, C. and C. Colberg, “A tool for teaching reverse engineering.”, in “ASE@ USENIX Security Symposium”, (2016).

[39] Wang, F. and Y. Shoshitaishvili, “Angr - the next generation of binary analysis”, in “2017 IEEE Cybersecurity Development (SecDev)”, pp. 8–9 (2017).

[40] Wang, R., Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel and G. Vigna, “Ramblr: Making reassembly great again”, (2017).

[41] Wang, S.-C., C.-L. Liu, Y. Li and W.-Y. Xu, “Semdiff: Finding semtic differences in binary programs based on angr”, in “ITM Web of Conferences”, vol. 12, p. 03029 (EDP Sciences, 2017).

[42] Wolf, F. M. *et al.*, *Meta-analysis: Quantitative methods for research synthesis*, vol. 59 (Sage, 1986).

[43] Zhang, Z., W. You, G. Tao, G. Wei, Y. Kwon and X. Zhang, "Bda: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation", Proceedings of the ACM on Programming Languages **3**, OOPSLA, 1–31 (2019).

[44] Zheng, Y., K. Cheng, Z. Li, S. Pan, H. Zhu and L. Sun, "A lightweight method for accelerating discovery of taint-style vulnerabilities in embedded systems", in "Information and Communications Security", pp. 27–36 (Springer, 2016).

[45] Zhu, E., F. Liu, Z. Wang, A. Liang, Y. Zhang, X. Li and X. Li, "Dytaint: The implementation of a novel lightweight 3-state dynamic taint analysis framework for x86 binary programs", Computers & Security **52**, 51–69 (2015).

APPENDIX A

RECRUITMENT ADVERTISEMENT

In order to recruit participants from the various academic and cybersecurity Discord servers, the following message was sent out: Have you taken ASU's CSE 365, CSE 466, or CSE 545 or do you have a background in reverse engineering, debugging, and exploiting binaries? If you answered "yes" to either of these questions, you are invited to participate in a research study conducted by Arizona State University! You will be asked to solve various challenges using a tool called angr management in order to research the effectiveness of new features on the platform. Prior experience with angr management is not necessary. The research study is approximately 1.75 hours (100 minutes) and can be taken at your time of choice. Your participation will be rewarded with a $50 Amazon gift card upon completion. For more information, contact ¡contact details¿. Participation in this study is voluntary.

## APPENDIX B

## SURVEY QUESTIONS

## B.1 Median

1. Have you seen this challenge before?
    - Yes
    - No
    - I prefer not to answer
2. Briefly describe what you did during this challenge (bullet point explanation is acceptable):
    - Free form response
3. Were you able to identify the source of the bug?
    - Yes
    - No
    - I prefer not to answer
4. After which function call to middle does the error originate?
    - First
    - Second
    - Third
    - Fourth
    - I do not know
    - I prefer not to answer
5. Which of the following best describes the nature of the bug?
    - Memory corruption
    - Signed comparison bug
    - Logical error
    - Integer overflow
    - I do not know
    - I prefer not to answer
6. If possible, briefly describe the cause of the bug.
    - Free form response

## B.2 Follow

1. Have you seen this challenge before?
    - Yes
    - No
    - I prefer not to answer
2. Briefly describe what you did during this challenge (bullet point explanation is acceptable):
    - Free form response
3. Were you able to solve the challenge?
    - Yes
    - No
    - I prefer not to answer
4. What input(s) did you provide to solve this challenge?.
    - Free form response

## B.3 Notes

1. Have you seen this challenge before?
    - Yes
    - No
    - I prefer not to answer
2. Briefly describe what you did during this challenge (bullet point explanation is acceptable):
    - Free form response
3. Were you able to identify the source of the bug?
    - Yes
    - No
    - I prefer not to answer
4. In which function call does the bug emerge?
    - delete_note
    - read_note
    - edit_note

- create_note
- I do not know
- I prefer not to answer

5. Which of the following best describes the nature of the bug?
    - Uninitialized variable
    - Signed comparison bug
    - Syntax error
    - Null dereference
    - I do not know
    - I prefer not to answer
6. If possible, briefly describe the cause of the bug.
    - Free form response

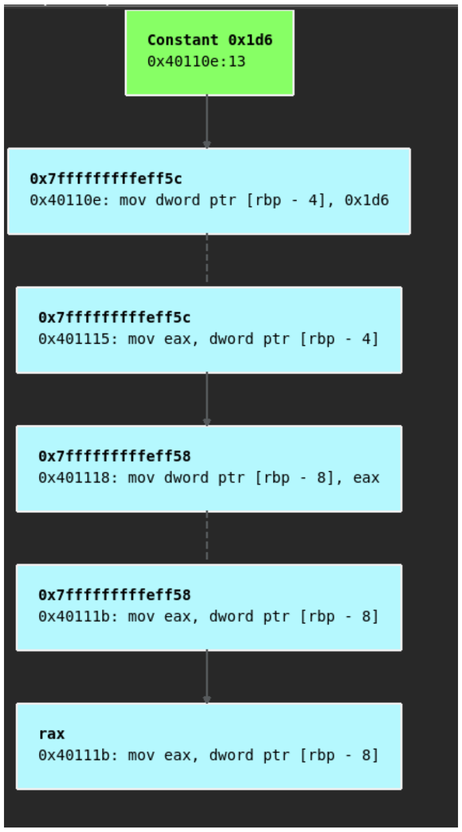
## B.4 End Survey

### *B.4.1 Control*

1. How many years experience do you have in software debugging?
    - None
    - Less than 1 year
    - 1 year
    - 2 years
    - 2+ years
    - I prefer not to answer
2. How many years experience do you have in vulnerability analysis?
    - None
    - Less than 1 year
    - 1 year
    - 2 years
    - 2+ years
    - I prefer not to answer
3. What is your perceived software debugging skill level?
    - Novice
    - Beginner

- Competent
- Proficient
- Expert
- I prefer not to answer

4. What is your perceived vulnerability analysis skill level?
    - Novice
    - Beginner
    - Competent
    - Proficient
    - Expert
    - I prefer not to answer
5. Have you used angr management before your participation today?
    - Yes



```
Constant 0x1d6
0x40110e:13

0x7ffffffffeff5c
0x40110e: mov dword ptr [rbp - 4], 0x1d6

0x7ffffffffeff5c
0x401115: mov eax, dword ptr [rbp - 4]

0x7ffffffffeff58
0x401118: mov dword ptr [rbp - 8], eax

0x7ffffffffeff58
0x40111b: mov eax, dword ptr [rbp - 8]

rax
0x40111b: mov eax, dword ptr [rbp - 8]
```

*Figure B.1.* Control group hypothetical data dependency view screenshot

- No
- I prefer not to answer

6. What is your perceived comfort level with angr management?
   - Novice
   - Beginner
   - Competent
   - Proficient
   - Expert
   - I prefer not to answer
7. I am sure that I correctly understood what the code of each challenge does
   - Strongly Disagree
   - Disagree
   - Neutral
   - Agree
   - Strongly Agree
   - I prefer not to answer
8. I found it difficult to understand these challenges
   - Strongly Disagree
   - Disagree
   - Neutral
   - Agree
   - Strongly Agree
   - I prefer not to answer
9. Disassembly view provided adequate information to solve these challenges
   - Strongly Disagree
   - Disagree
   - Neutral
   - Agree
   - Strongly Agree
   - I prefer not to answer
10. A data dependency graph view like in the image below (Figure B.1) would have been helpful

- Strongly Disagree
- Disagree
- Neutral
- Agree
- Strongly Agree
- I prefer not to answer

11. This view in the image below (Figure B.1) seems confusing and unnecessary
    - Strongly Disagree
    - Disagree
    - Neutral
    - Agree
    - Strongly Agree
    - I prefer not to answer

### *B.4.2 Experiment*

1. How many years experience do you have in software debugging?
    - None
    - Less than 1 year
    - 1 year
    - 2 years
    - 2+ years
    - I prefer not to answer
2. How many years experience do you have in vulnerability analysis?
    - None
    - Less than 1 year
    - 1 year
    - 2 years
    - 2+ years
    - I prefer not to answer
3. What is your perceived software debugging skill level?
    - Novice
    - Beginner
    - Competent

- Proficient
- Expert
- I prefer not to answer

4. What is your perceived vulnerability analysis skill level?
    - Novice
    - Beginner
    - Competent
    - Proficient
    - Expert
    - I prefer not to answer
5. Have you used angr management before your participation today?
    - Yes
    - No
    - I prefer not to answer
6. What is your perceived comfort level with angr management?
    - Novice
    - Beginner
    - Competent
    - Proficient
    - Expert
    - I prefer not to answer
7. I am sure that I correctly understood what the code of each challenge does
    - Strongly Disagree
    - Disagree
    - Neutral
    - Agree
    - Strongly Agree
    - I prefer not to answer
8. I saw no need to use data dependency view to solve these challenges
    - Strongly Disagree
    - Disagree
    - Neutral

- Agree
- Strongly Agree
- I prefer not to answer

9. Data dependency view aided in understanding the challenges
   - Strongly Disagree
   - Disagree
   - Neutral
   - Agree
   - Strongly Agree
   - I prefer not to answer
10. Data dependency view is lacking in valuable information
    - Strongly Disagree
    - Disagree
    - Neutral
    - Agree
    - Strongly Agree
    - I prefer not to answer
11. I find data dependency view useful
    - Strongly Disagree
    - Disagree
    - Neutral
    - Agree
    - Strongly Agree
    - I prefer not to answer
12. Data dependency view is redundant and unnecessary
    - Strongly Disagree
    - Disagree
    - Neutral
    - Agree
    - Strongly Agree
    - I prefer not to answer
13. Data dependency view was clear and easy to understand

- Strongly Disagree
- Disagree
- Neutral
- Agree
- Strongly Agree
- I prefer not to answer

14. Data dependency view confused me
    - Strongly Disagree
    - Disagree
    - Neutral
    - Agree
    - Strongly Agree
    - I prefer not to answer

15. If any, what improvements could be made to make data dependency view more user friendly?
    - Free form response

## APPENDIX C

## IRB APPROVAL

EXEMPTION GRANTED

Adam Doupe
SCAI: Computing and Augmented Intelligence, School of
-
doupe@asu.edu

Dear Adam Doupe:

On 2/21/2022 the ASU IRB reviewed the following protocol:

| | |
|---|---|
| Type of Review: | Initial Study |
| Title: | Expediting Binary Analysis Through Data Dependency Graphs and Proximity Control Flow Graphs |
| Investigator: | Adam Doupe |
| IRB ID: | STUDY00015332 |
| Funding: | Name: DOD: Defense Advanced Research Projects Agency (DARPA), Funding Source ID: FP00017167 |
| Grant Title: | *CHECRS: Cognitive Human Enhancements for Cyber Reasoning Systems* |
| Grant ID: | *FP00017167* |
| Documents Reviewed: | • Consent_Form, Category: Consent Form;<br>• DARPA Proposal, Category: Sponsor Attachment;<br>• Debugging and Vulnerability Challenge Questions, Category: Measures (Survey questions/Interview questions /interview guides/focus group questions);<br>• Debugging_and_Vulnerability_Survey, Category: Measures (Survey questions/Interview questions /interview guides/focus group questions);<br>• Entire Experiment Text Outline, Category: Participant materials (specific directions for them);<br>• Instructions / Greeting, Category: Recruitment Materials;<br>• IRB Form, Category: IRB Protocol;<br>• Recruitment Message, Category: Recruitment Materials;<br>• Reverse Engineering Challenge Questions, |

| | |
|---|---|
| | Category: Measures (Survey questions/Interview questions /interview guides/focus group questions);<br>• Reverse_Engineering_Survey, Category: Measures (Survey questions/Interview questions /interview guides/focus group questions); |

The IRB determined that the protocol is considered exempt pursuant to Federal Regulations 45CFR46 (2) Tests, surveys, interviews, or observation, (3)(i)(A) - benign behavioral interventions on 2/21/2022. As a part of IRB review, scientific merit was considered.

In conducting this protocol you are required to follow the requirements listed in the INVESTIGATOR MANUAL (HRP-103).

If any changes are made to the study, the IRB must be notified at research.integrity@asu.edu to determine if additional reviews/approvals are required. Changes may include but not limited to revisions to data collection, survey and/or interview questions, and vulnerable populations, etc.

*REMINDER - Effective January 12, 2022, in-person interactions with human subjects require adherence to all current policies for ASU faculty, staff, students and visitors. Up-to-date information regarding ASU's COVID-19 Management Strategy can be found here. IRB approval is related to the research activity involving human subjects, all other protocols related to COVID-19 management including face coverings, health checks, facility access, etc. are governed by current ASU policy.*

Sincerely,

IRB Administrator

cc: Sean Smits
Sean Smits
Zeming Yu
Adam Doupe
Ruoyu Wang
Bailey Capuano