USING TEMPORAL NETWORKS TO FIND THE INFLUENCER NODE OF THE BUGGY SITES IN THE CODE COMMUNITIES

by

Kanwardeep Singh Walia

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science

Department of Computer and Information Technology

West Lafayette, Indiana

May 2022

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. John A. Springer, Chair

Department of Computer and Information Technology

Dr. Romila Pradhan

Department of Computer and Information Technology

Dr. Jin Wei-Kocsis

Department of Computer and Information Technology

Approved by:

Dr. John A. Springer

ACKNOWLEDGMENTS

First of all, I would like to thank Dr. John Springer for advising and supporting me throughout my time at Purdue University. Your feedback from my first day at Purdue University helped me to critically think about my research. Your assistance in narrowing down the research questions and coming up with a data set has been very helpful. I really appreciate all the encouragement and mentoring about my career prospects. This degree won't have been possible without your support.

My thesis committee has been very helpful and I thank them for their insightful feedback. The feedback was very helpful in changing the thesis proposal into a final product. Using the "centrality algorithm" in the research was very useful and I am thankful Dr. Romila Pradhan and the thesis committee recommended it during my thesis proposal.

My family and friends have been a constant support throughout my time at Purdue University. I would like to thank all the people who stayed connected with me and kept me motivated.

I would like to thank the Computer and Information Technology at Purdue University for providing me with Graduate Teaching Assistantship. It helped me gain so much experience and make me ready for the industry positions. Thank you, Dr. Springer, for putting your trust and recommending me for the position.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

NIST	National Institute of Standards and Technology
bof	Buffer Overflow
CWE	Common Weakness Enumeration
BF	Bugs Framework

GLOSSARY

National Institute of Standards and Technology SAMATE – "Development of metrics for the effectiveness of software security assessment (SSA) tools. Assess current SSA methods and tools in order to identify deficiencies which can lead to software product failures and vulnerabilities." [1]

Buggy Site – "A buggy site is one that has an instance of the weakness. That is, there is some input that will cause a violation. A non-buggy site is one that does not have an instance of the weakness." [2]

Influencer node – A node (variable) in the source code which acts as the root cause for the buggy site. Once the influencer node is fixed (such as changing data type), the buggy site vulnerability will be fixed automatically.

K-tool – The developed framework for this research to find the influencer node of the buggy site.

ABSTRACT

The cyber-attacks have increased, and with everything going digital, data theft has become a significant issue. This raises an alarm on the security of the source code. Sometimes, to release products early, the security of the code is compromised. Static analysis tools can help in finding possible security issues. Identifying and fixing the security issues may overwhelm the software developers. This process of "fixing" the errors or securing the code may take a lot of time, and the product may be released before all the errors are fixed. But these vulnerabilities in the source code may cost millions of dollars in case of a data breach. It is important to fix the security issues in the source code before releasing the product. This leads to the question of how to fix errors quickly so products can be rolled out with fewer security issues? A possible solution is to use temporal networks to find the influencer nodes in the source code. If these influencer variables are fixed, the connected security issues depending on the influencer in the community (functions) will also get fixed. The research question of the study: Can we identify the influencer node of the buggy site in the source code using temporal networks (K-tool) if the buggy sites present in the source code are identified using static analysis? The study also aims to know if it is faster to find the influencer node using the K-tool than the betweenness centrality algorithm. This research is an "Applied research" and will target the code written in C programming language. Possible vulnerabilities that can be fixed include "Integer Overflow", "Out of bounds", and "Buffer overflow." In the future, we plan to extend to other errors such as "Improper input validation." In this research, we will discuss how we can find the influencer node of the vulnerability (buggy site) in the source code after running the static analysis. Fixing this influencer node will fix the remaining errors pointed out by the static analysis. This will help in reducing the number of fixes to be done in the source code so that the product can be rolled out faster with less security issues.

1. INTRODUCTION

This chapter provides an overview of the research study. It introduces the research by presenting a background of the problem area and research questions. This study also discusses the significance, scope, limitations, and the delimitations of our the research.

1.1 Background

Static Analysis tools work by "scanning and parsing the source text of a program to look for a fixed set of patterns in the code" [3]. The authors describe the process as control, data, interface, path, and information flow analysis. The output of the tools indicates the possibility of an error and may not mean an actual fault. The paper describes that static analyzers helped in finding and fixing faults before they are reported by customers or identified during inspections. These tools look for violations of development standards, non-compliance of coding standards, redundancy in code, memory leaks, division by zero, and many other issues. This shows the need for static analyzers since they can find issues before the software is released saving huge costs. Static analyzers have been successful in finding vulnerabilities such as SQL injections, HTTP splitting attacks, etc. [4]. The paper reports that security breach occurs in nearly half of all databases and the average episode of security breach results in losses of about $4 million. Buffer overflows, string vulnerabilities and other unsafe nature of C have focused projects to prevent vulnerabilities. There are static analysis tools such as Frama-C and Astree which can be used on C source code to find vulnerabilities [1]. Google has developed its static analysis tool and shared the process [5].

1.2 Problem Statement

Even though there is a huge need for the use of static analysis on the software before releasing the product, a survey revealed that the software developers are not using static analysis tools to find bugs in the code [6]. According to the survey, developers gave reasons such as "so many warnings" or "information they provide is not very useful." The paper

reports that false positives and work overload are the reasons for not using the static analysis tools.

This shows the need for coming up with tools that can assist in quickly finding the issues in the source code so that the developers don't have to go through all the static analysis warnings. The developers can focus on the influencer node (source of error) which if fixed will dissolve many other related errors reported by the static analysis tool. Let's dive into the problem by considering an example source code containing a buggy site (shown in Listing 1.1).

Listing 1.1. Example code with an influencer

```
1 long RandomVariable = 40000;              // Line 1
2 short TestVariable = 0;                   // Line 2
3 TestVariable = RandomVariable * 150       // Line 3
```

Running the static analysis on the source code shown in Listing 1.1 will indicate that there is an integer overflow on line 3 with some additional information. However, it is not indicated where is the influencer or the cause of the problem.

The cause of the issue is located on the declaration of the data type of line 2 in Listing 1.1. "TestVariable" is declared as short data type whereas the "RandomVariable" is declared as a long data type. The value of "RandomVariable" could be really large which will cause an integer overflow. Therefore, the influencer is located on *line 2* which should be fixed. This research aims to find these influencer sources of errors in the code communities.

1.3 Research Questions

This research contributes answers to the following questions:

1. Can we identify the influencer node of the buggy site in the source code using temporal networks (K-tool) if the buggy sites present in the source code are identified using static analysis?

2. Is it faster to find the influencer node using the K-tool than the betweenness centrality algorithm?

1.4 Significance

The background section shows that the static analysis tools have been successful in finding vulnerabilities in the source code. Moreover, it is discussed how security breaches can result in losses. Therefore, it is critical to fix the vulnerabilities in the source code. As discussed in the problem statement, the developers have difficulty finding how to fix the vulnerability after it's indicated by the static analyzer tool. This research aims to find a way to cover the bridge between the information provided by the static analyzer tool and *how to fix* the vulnerability. I came up with methods to find the influencer node of the buggy site in the source code. Once the node which causes the issue is identified along with its line number, the vulnerability in the source code can be fixed. This can be achieved by modifying the data type of the influencer node (variable) or trying other options such as adding an if-statement. The if-statement can check if the variable value exceeds more than what the variable data type can handle. This approach will help software developers to easily identify what to fix without looking at too much information which may complicate the bug fixing process.

1.5 Scope

The scope of the research is to find the influencer nodes in the C source code files. It should be able to cover specific classes listed below [7]:

- CWE121 Stack Based Buffer Overflow
- CWE126 Buffer Overread
- CWE190 Integer Overflow

The test cases from the above mentioned classes will be used to generate a data set for this research. However, not all the test cases from the above listed classes will be used due to complexity of various test case categories. For example, the implementation of the "postinc" test cases is different from the implementation of the "multiply" test cases.

1.6 Assumptions

This study makes the following assumptions:

- The static analysis tool used gives the correct location (line number) of the buggy site. Moreover, the buggy site should be a *real* buggy site, and not a false positive.
- The data set generated for this research should be similar to the real-world software or a small operating system. The data set contains Juliet test cases to increase the number of buggy sites.
- The "bad_function" in the Juliet test cases contains a buggy site.
- Line number of the source code file will be used as an *event* for the temporal network.
- K-tool assumes that the "=" sign in the C source code file sets up a relationship between the affected node and the incoming nodes.
- The results for performance should be retrieved for different methods using the same environment (same operating system and same device).

1.7 Limitations

The following are the limitations of this study:

- K-tool results don't recommend how to fix the influencer of the buggy site.
- K-tool currently doesn't find the influencer node of the buggy sites in *all* the source code files.
- K-tool is capable of finding the influencer of the buggy site in the same source code file (same community) but doesn't track the influencer across different source code files (across different communities).
- Source code files with data types such as "typedef", "double", "float" are not supported by the K-tool.

- K-tool may find the influencer node of the buggy site in the source code files containing "wchar" but it is not guaranteed.
- The static analysis may halt on some generated data set source code files which may result in less number of buggy sites. This can result in less number of findings for the influencer nodes of buggy sites using the K-tool.

1.8 Delimitations

This study has the following delimitations:

- The data set generated for this research only contains "void" functions instead of functions with different return types such as "int" and "char."
- K-tool only utilizes "Source code file name" and the "Buggy site line number" to find the influencer node of the buggy site once the temporal network graph for the source code is generated.
- Betweenness centrality algorithm framework doesn't utilize the static analysis information to accurately find the influencer node of the buggy site. Instead, this method finds all the influencer nodes in the source code.
- This study doesn't confirm if all the influencer nodes indicated by the betweenness centrality algorithm are the influencer nodes of the buggy sites in the source code.

1.9 Summary

This chapter mentioned the problem statement, research questions, significance, and the scope of the research. Moreover, we discussed the assumptions, limitations, and the delimitations of this study.

2. REVIEW OF LITERATURE

This chapter provides a review of the literature relevant for network analysis, network graphs, the use of driver nodes, temporal networks, call graphs, and the centrality algorithm.

2.1 Temporal Network Graphs

In the temporal network, the number of edges can be difficult to manage due to the occurrence of patterns at different times [8]. This is why there is a need for efficient algorithms to analyze the data. Moreover, the authors mentioned how pairs of nodes stay connected forever and may not capture the full rich data of temporal networks. This brings an interesting point regarding how to organize the graphs so that every event information is captured while making sure that the change in node connections is captured properly. For example, what if the node is not connected to the other node after a certain event or time? In our research, the events are not time dependent. However, an event occurs when moving to the next line of the source code. A variable may be connected to another variable at one time (line) but may lose that connection in some other line of the source code. The authors of the paper also provided a graph tracking mechanism for the time-based events.

2.1.1 Driver Nodes

Driver node is the concept of finding a node that is driving or influencing the network. A complex network is defined as a system of interlinked nodes [9]. The authors mentioned that over time, a change in the state of node can occur. Moreover, the paper mentioned how the driver nodes are not unique and the minimum number of nodes present in a complex network are fixed. This is related directly to our research since we want to come up with a criteria for classifying if the node is the *driver node* in the community. For example, a node affecting many other nodes could be classified as a driving node. The authors of the paper proposed an algorithm to select a driver node using which they achieved an increase in Region of Acceptance by 95%.

To identify the minimum number of driver nodes, the graph reachability approach can be used in a directed network [10]. The authors proposed an algorithm to find a node with the highest graph reachability index and assign it as a driver node.

When link weight is present on the edge of the network, the structure controllability is not satisfied [11]. The authors proposed a method to find the minimum number of driver nodes that are capable of stronger control. The method uses searching for an optimal connection between nodes and control signals. This could be beneficial in our research to identify the driver nodes in the temporal network. The structural controllability is important since our source code graph networks may deal with a large number of nodes and links.

Structural controllability has been discussed as an important feature of a directed network because the network can get to the desired states with only a few nodes [12]. The paper proposed that initially all the nodes of the system should be considered and categorized as driver nodes. Then, nonzero elements in the column should have at least one node which should be controlled.

2.1.2 Analysis of Networks

Network analysis is a growing field used in many different fields such as biology, ecology, etc. [13]. In our research, the methods used in different fields can be highly applicable. For example, we can learn about the algorithms and methods that can be used to handle the complex network system with millions of edges. Moreover, how the communities and the driver nodes can be identified? Coming back to the paper, the authors mentioned how the static methods have been utilized to analyze dynamic networks. This has been done by capturing the state of the network at a specific time to answer the question (fast analysis). The authors briefly discussed methods for creating time-ordered networks by using arrows between the start and stop times of interaction. Moreover, it is discussed how bidirectional arrows can be used for undirected interactions. It can be connected with our research which doesn't contain the time interactions but instead has line-based event interactions (going to the next line of source code). However, it should be noted that the bidirectional arrows may

not be used in our research because the direction of the edge is related to the source and the target of the line event.

Temporal networks have been used in other fields such as identifying air traffic or driving nodes in the rail systems. The systems which have been studied are highly complex due to the change in links between the nodes over time. There has been discussion about using a dynamic community structure [14]. The authors have developed a method to identify the dynamic patterns of the network and determine timescales. The paper described how the static network structures have been used instead of the dynamic nature of the networks. This becomes important in our research where every source code line can bring a new change in the temporal graph network making it dynamic. The authors used Markov chain methods and proposed that the method used in their research will conclude the lack of data if the dynamics of the network are random.

2.2 Identifying security issues and using Static Analysis

This section discusses the code security issues that exist, what static analysis is, and which static analyzer tools can be used.

Firstly, it is important to understand what static analysis tools are? They are used to examine the source code text without executing it [15]. Software developers can make mistakes during software coding which could be a security issue and can become expensive to fix if left dormant for a longer period. The static analysis tools are used to examine if there are any problems in the source code before it can be released. The paper mentions how manually auditing the source code can be difficult and a time-consuming process. Static analysis tools can only fix certain problems which are written in a set of rules or patterns. Static analysis process should be performed right after the code is written and before the test results are checked. The author discussed different tools such as *BOON* which can be used to check if the array is out of bounds in C. *CQual* can be used for a taint analysis and Splint tools can be used for checking unannounced modifications to the "global variables", "use of variable before initialization", "array bound access", etc.

Static analysis tools can also be used to identify security vulnerabilities such as SQL injections, format string, cross-site scripting, etc [16]. *FindBugs* is one of the examples of static analyzer which started as a project after the observation that some problems in Java code could be detected with trivial analysis. It was observed that the production quality software also contained mistakes. The paper discussed Google's experience with the FindBugs and the process to run it. The process included automating the FindBugs tool to run on the source code. The generating warning showing the buggy site was stored. This process can also be used in our research since the static analysis should be automated to run over the data set to generate a warning (find the buggy site). The paper mentions the use of a web interface that could inform about the false positives. However, in the research, it wasn't clear which warnings belonged to which file. This shows the importance of a static analyzer tool that can point out the buggy site file name, line number, and the error type.

2.2.1 Existing types of errors

It is important to understand the existing security issues in the software source code and how they can be categorized. Static analysis tools assist in identifying security issues. However, which security issues belong to which class needs to be mapped. SAMATE Team at NIST took an approach to create the Bugs Framework (BF) which categorizes the different CWEs (Common Weakness Enumeration) under a single class [17]. For example, categorizing write outside of buffer on the stack (CWE-121) and write outside of a buffer in heap (CWE-122) as one class called Buffer Overflow. CWE contains software weaknesses of more than 600 types. The number is too large to focus on one specific security issue. Therefore, our research should look into a single class (or a few classes) of security weakness that could cover different types of CWEs. The authors of the paper categorized 9 CWEs as Buffer Overflow. The list includes CWE 119, 120, 121, 122, 123, 124, 125, 126, 127 which are related to stack overflow, heap overflow, overread, underread, etc. The attributes of BOF are Read and Write. The paper described the causes of the buffer overflow including input not checked properly, array too small, too much data, no NULL termination, wrong Index/Pointer Out of Range, incorrect conversion, integer overflow-wrap around, integer

underflow, etc. The consequences include incorrect results, program crash, system crash, arbitrary code execution, server access/host takeover, Denial of Service, resource exhaustion, altered control flow, etc. This paper helps in understanding and deciding the specific class our research could focus on. Buffer Overflow seems to have a broad number of issues that can have huge consequences. This class could be a suitable part of our research. However, it is important that the static analysis tools should be able to identify these CWE errors (vulnerabilities) in the source code. The paper further discussed other CWEs and categorizes the Injection (INJ) class as parsing an assembled class string into an invalid construct. The causes of this class could be :

1. Input Not Checked properly: "Incomplete blacklist, permissive whitelist, white/black list not Checked Properly" [17].

2. Input Not Sanitized Properly: "Failure to reject Input Altogether, Failure to Remove Offending Characters, Failure to 'Escape' Offending Characters" [17].

The Injection Class consequences include the addition of a command, incorrect results, altered control flow, denial of service, information exposure/change/loss, arbitrary code execution, account access, credentials compromise, etc. The paper further discussed the Control of Interaction Frequency Class (CIF) class which limits the number of repeated interactions or events per user.

1. Number of Interactions Not Checked properly: "Failure to recognize repeated interactions, Failure to Limit, Failure to Properly Limit" [17].

2. Frequency of Interactions Not Sanitized Properly: "Failure to Properly Limit, Failure to Limit" [17].

The attributes of CIF Class include interaction such as authentication attempts, vote, book, checkout, register, and initiate. The unit of attempt prevention could be time, user, or an event. The actor includes the user, automated process, etc. The consequences of this class are Program Logic compromise, incorrect results, wrong physical actions, system compromise, resource exhaustion, call-number exhaustion, information exposure, credentials

compromise, admin server access, etc. This paper gives a lot of information for the ground base of our research and helps in deciding "what class" or "vulnerability" type to pursue. For example, BOF, INJ, and CIF are various classes that could be selected for our research. Moreover, the paper helps in the selection of the static analysis tools. The static analysis tool should be able to identify the issues in source code related to the class. For example, if the static analysis tool doesn't support the Injection (INJ) class, then the INJ class shouldn't be picked for research due to the lack of information regarding the INJ buggy sites.

The Bugs Framework (BF) developed by SAMATE Team at NIST categorized classes such as Buffer Overflow (BOF), Injection (INJ), Verification Bugs (VRF), etc. [18]. BOF is described as access to a memory location outside of that array boundaries. The other BF classes are related to language special elements, conversion of plaintext into ciphertext, checking if data is altered, range violations, etc. The authors proposed the True-Random Number Class for the Bugs Framework which is related to the space of possible outputs. It also mentioned how insufficient random space may lead to the Denial of Service (DOS) attacks. The possible causes could be the using improper external algorithms, incorrect entropy assessment, etc. which could have other consequences such as program crashes, small space, etc. This literature review helps us to better understand the software security issues that could occur. Moreover, our research should include identifying the static analysis tools that can identify the above listed source code vulnerabilities. The overall goal is to identify the influencer nodes (root cause node) of the buggy site (vulnerability) once the buggy sites in the source code are identified using the static analysis tools.

Memory Related Bugs have been defined in 4 different classes [19]. Memory Addressing Bugs (MAD) are caused by Improper operations which are missing, and mismatched. Other causes are Improper Pointers which could be a NULL pointer, Over bounds, Hardcoded address, etc. The consequences could be an improper pointer for the next operation. The Memory Allocation Bugs (MAL) class could lead to not having enough memory allocated for the next operation, memory overflow, memory leak, object corruption, etc. The Memory Use Bugs (MUS) class could lead to memory errors such as uninitialized object, not cleared object, NULL Pointer Dereference, Object Corruption, type confusion, Buffer Overflow, Buffer Underflow, and Uninitialized Pointer Dereference. The memory Deallocation Bugs

(MDL) class could lead to an Improper Pointer/Object for the next operation, memory leak. The paper further described the various memory errors and risks. Moreover, the paper provides information regarding the possible security issues in the source code and how it could be hard to visualize these issues as a buggy site.

The need for having artificial source code instead of natural source code has been discussed [20]. The artificial data set could be used to test the performance of different static analysis tools and a standard list could be used to track all the bugs or flaws in the artificial source code. Moreover, complex data flows can be generated in the artificial code which may not be available in the natural code. The paper discussed the difficulty involved in finding if software vulnerability (bug) is a "real" vulnerability. The limitation of using the artificial source code is that the flaws indicated by the static analysis tool may not occur as frequently in the natural source code.

2.2.2 Selecting the static analysis tool

SAMATE team at NIST discussed the results of running the static analysis tool on the selected test cases during the SATE event [1]. The paper mentions the Ockham Sound Analysis Criteria to recognize the "static analyzers whose analysis is logically sound" [1]. The tools can identify all buggy sites in the source code for at least one weakness class. For our research, the static analysis tool should be able to identify the buggy sites in the source code. Using the buggy site information, the influencer nodes can be determined. It should be noted that if the buggy sites identified are not correct, the influencer node of the buggy site won't be correct. The report also discussed how both Astree and Frama-C passed the SATE VI Ockham Sound Analysis Criteria. The criteria included that the tool claimed to be sound, produced findings for at least one weakness class for a certain threshold of appropriate sites, and doesn't report incorrect findings. For this study, either of the tools can be selected for finding buggy sites in the data set. The report also mentioned that both Astree and Frama-C were run on CWE 121, 122, 123, 124, 126, 127 which are a part of the Buffer Overflow Class [17]. The paper also described how CWE 190 (Integer Overflow) and CWE 191 (Integer Underflow) were tested.

2.3 Call graphs

Creating call graphs for source code has been a useful approach in many cases. It has been used to create features such as source code completion [21]. The authors discussed how state-of-the-art technologies can be used to fill information such as initialization or calling methods. Moreover, the code completion methods based on call graphs have been used to help developers in calling API elements using the current line context.

Removing duplicated source code or code clones has been addressed [22]. It includes creating procedures that can be called instead of the code clones to fix the issue of maintaining multiple copies of the same code. The authors of the paper discussed the approach of control sequence and data dependence of the source code. The lines in the source code such as a "while" statements are considered as a node. Moreover, the paper mentions slicing nodes into various nodes, and pairing similar sliced nodes into a group. This approach is similar to our research where K-tool (developed) takes a line as input and slices it down until all the variable nodes of the source code are identified.

There is research about understanding complex networks such as Linux Kernel. The call graphs have been created to understand the evolution of the Linux kernel source code [23]. The research discussed that the functions are considered as a node and the relationships are set up between the called function and the function calling that function. The research concludes that Linux is stable because the new node added in the newer Linux versions is usually attached to only a few nodes. Connecting this with our research, there should be fewer influencer nodes in the source code to prevent the influence of the buggy sites. This way the damage done by the buggy site can be decreased. Moreover, another paper by the authors goes on to investigate the in-bound and the out-bound nodes in the Linux kernel [24]. The paper identified more than 12,000 nodes in the Linux kernel where the nodes are functional units, and the relationship between the functional units and call events created a "call graph."

Java source code centrality has also been explored by the research community [25]. The paper focused on creating a call graph for the source code to find the arcs and the nodes. The authors analyzed more than 700 nodes and more than 1000 arcs. It is discussed how

the PageRank, closeness, eccentricity, HITS-Authority, and betweenness centrality methods are used to find the top 10 nodes in the Java source code. The results from all the different centrality methods had only some common nodes in the list of top 10 nodes.

2.4 Influencer node detection using the centrality algorithm

This section discusses the implementation of the centrality algorithm for the detection of influencer or important nodes in the source code communities. The overall goal of this research is to find the influencer nodes of the buggy sites in the source code. The centrality algorithm can assist in finding the influencer nodes in the source code which can later be used to find the influencer nodes of the buggy sites.

Centrality can be used to find the important nodes in a network [26]. Moreover, the author discussed the concept of betweenness centrality which can be used to identify nodes that play an important role in paths to other nodes in the network. The paper also described how the flow of information can be controlled by these nodes. Moreover, a node can have few links while maintaining higher betweenness centrality. This shows that betweenness centrality is different than just finding nodes with the highest degree. It helps in finding the important nodes in the system. There are other centrality methods such as degree centrality, closeness centrality, eigenvector centrality, etc. which are discussed by the authors.

The centrality concept has been used in various domains such as finding plagiarism in the source code [27]. The paper discussed how lexical changes in the source code can be detected. This includes identifying comments, changes to names, formatting modifications, etc. The authors mentioned that finding the number of variables and words per line could be a possible way to detect plagiarism. This methodology can also be used to create call graphs of the source code to find the important variables. The paper also discussed the different classifications based on centrality. For example, *mix* classification will have high betweenness centrality while containing low degree centrality. The other classification is the *star* classification where nodes can have high betweenness centrality and high degree centrality.

Nodes with high inbound degree centrality are described as "sinkhole" points in the network and may lead to a dependency risk [28]. The authors explain how nodes with a higher betweenness centrality become important in contributing to the overall risk due to their interaction with multiple risk paths. However, it doesn't mean that these nodes are on the path of high risk. In our research, influencer nodes identified using the highest betweenness centrality could be the important nodes in the source code. It should be noted that the identified nodes using betweenness centrality may not be the source of the buggy sites. Moreover, we speculate that in our research, the influencer nodes of the buggy sites found using the K-tool will most likely be a subset of the influencer nodes found by the betweenness centrality algorithm.

2.5 Summary

This chapter provided a review of the literature relevant to the temporal networks containing driver nodes (influencers) in the communities. Different static analysis tools, code security issues, and call graphs are also discussed. The chapter also mentioned how the influencer nodes can be detected using the betweenness centrality algorithm.

3. METHODOLOGY AND FRAMEWORK

This chapter discusses the methodology of this research. This includes a discussion of needing a framework, how the framework should be designed to find the influencer nodes, and the expected output from the framework.

3.1 Need for a Framework

The major objective of the framework is to find the influencer node (variable) which is the source of error of the buggy site inside the communities of the C source code files. The influencer node of the buggy site is the variable that "when fixed" will also fix the dependent errors inside the community. The "influencer bad line" of source code is the line in the source code where the influencer node is located. When the static analysis tool runs on the source code, it indicates the lines which may contain an error in the source code. However,the research aims to find what causes the issue (vulnerability). This framework assists in filling this information gap so that the software developers know exactly where to fix the vulnerability rather than manually going through the source code to find the cause of vulnerability.

Running Static Analysis tools on source code in Listing 3.1 (also discussed in Section 1.2) indicates *line 3* has an error since the short data type can't handle value equal to 40,000. However, information regarding what caused the problem is not available which is located on *line 2*. The developed framework which we call "K-tool" will provide information that the variable "TestVariable" on *line 2* is the influencer nodes of the buggy site.

Listing 3.1. Example code with a buggy site

```
1 long RandomVariable = 40000;              // Line 1
2 short TestVariable = 0;                   // Line 2
3 TestVariable = RandomVariable * 150       // Line 3
```

Let's imagine there are more lines of source code as shown in Listing 3.2. Running the static analysis tool on the source code should indicate that lines 3, 4, 5, and 7 contain a buggy site (vulnerability). These buggy sites depend on the data type of "TestVariable" (declared on line 2) and fixing the data type of the "TestVariable" will resolve all the vul-

nerabilities inside the function (community). This is the reason to consider "TestVariable" as an *influencer node* of the buggy site in the community.

Listing 3.2. Example code with influencer node of the buggy site

```
1 long RandomVariable = 40000;              // Line 1
2 short TestVariable = 0;                   // Line 2
3 TestVariable = RandomVariable * 150;      // Line 3
4 TestVariable = 32000 + 1500;              // Line 4
5 TestVariable = TestVariable + 1500;       // Line 5
6 TestVariable = 0;                         // Line 6: Not a buggy site
7 TestVariable = RandomVariable - 5000;     // Line 7
```

The focus of this study is to find the influencer nodes of the buggy sites and aims to answer the research question: Can we identify the influencer node of the buggy site in the source code using temporal networks (K-tool) if the buggy sites present in the source code are identified using static analysis? Secondly, we aim to know if it is faster to find the influencer node using the K-tool than the betweenness centrality algorithm?

3.2 Research Type

This research is a mix of Technological Applied Research and Quantitative Research. The focus of this research is to come up with a way to use temporal network graphs and the static analysis results to find the influencer nodes of the buggy sites. Moreover, it is quantitative research because the research aims to investigate if the developed K-tool framework indeed helps in accurately locating the influencer node (source of error) in the source code communities. The results will indicate how many influencer nodes of buggy sites are present in various code communities. Moreover, we aim to know how different the number of influencer nodes of buggy sites indicated by the K-tool is from the number of influencer nodes indicated by the betweenness centrality algorithm.

3.3 Motivation for using Temporal Networks

Why should we use a temporal network? From Listing 3.3, it can be observed that whenever the line of code is run, it may depend on previous information in the code. It can be said that the nodes can influence the outcome of running specific line(s) of code. It is

dynamic because the outcome of running a line of code depends on the information provided to it by the previous link. There is a chain of links that occurs in different lines of the source code. The outcome of every link of code could be different based on the information provided by some other node (variable). Therefore, even in the static analysis of the source code, we can see the dynamic nature due to the links based on the line numbers.

3.3.1 Understanding the problem

The Listing 3.3 shows a complex example source code where multiple links occur. Moreover, the result of running the program shown in Table 3.1 is not what a user expects. This is the reason we consider the example as motivational source code example which demonstrates that the buggy sites can change the output of the source code. Line 14 in the source code acts an influencer node of the buggy site where as the buggy site is located on line 22. This is because "RandomVariable" is declared as a short data type and can't handle value more than "32,767" (tries to insert the value equal to 40,000).

Let us discuss about the significance of the links from an influencer node. If the influencer node (variable) can be found in the source code and if we can confirm that there are no buggy sites, then there is a higher chance that the source code is safe. We can also be sure that wherever the influencer node is connected, it provides the correct information. However, if the influencer has information that contains an error, then we know there is a higher chance that its links are delivering wrong information. For example, the influencer node in the Listing 3.3 is "RandomVariable" which contains a wrong value due to the incorrect data type. Therefore, the information delivered to the "TestVariable" on line 35 is also incorrect. As a result, all these connected nodes contain an error. If we only fix the data type for the "RandomVariable" (influencer node) from *short* to *long*, then the vulnerability errors in the source code will be fixed.

3.3.2 Observing Temporal Networks from a Higher Level

Temporal network graphs can be used observe how different variables and source code files interact in the codebase. If the variables are declared as global (or in another source

code file), we should be able to distinguish them from the variables of the function. This way we can keep track of how much effect the variable can make. For example, if the variable is in the function scope, then it will be the dominating factor. If changes are made to a global

Listing 3.3. Example code showing effects of Source of Error

```
#include <stdio.h>

long RandomVariable = 68000;

int main()
{
    printf("\nRandomVariable(global) = %d", RandomVariable);

    printf("\nHello World");

    long ExtraVariable = 0;

    //NEXT LINE SHOULD BE FIXED (INFLUENCER)
    short RandomVariable = 10;   //Removing this line
    //will make code work correctly
    printf("\nRandomVariable = %d", RandomVariable);

    RandomVariable = 32770;
    printf("\nRandomVariable = %d", RandomVariable);


    RandomVariable = 40000;
    printf("\nRandomVariable = %d", RandomVariable);

    // Do something with RandomVariable
    if (RandomVariable > 30000)
    {
        ExtraVariable = 50000;

        printf("\nOUR CODE IS DOING IMPORTANT WORK");   // Consequence
    }

    // Now declare new Variable
    long TestVariable = 0;
    TestVariable = RandomVariable * 150;   // TestVariable is influenced

    printf("\nTestVariable = %d", TestVariable);

    // TestVariable became the new influencer in our function

    if(ExtraVariable > 0 && TestVariable > 0)
    {
        printf("\nANOTHER IMPORTANT WORK"); // Consequence
    }

    return 0;
}
```

Table 3.1. Output from Listing 3.3

Output of running the example code showing effects of Source of Error
RandomVariable(global) = 68000
Hello World
RandomVariable = 10
RandomVariable = -32766
RandomVariable = -25536
TestVariable = -3830400

variable, it should be tracked as well. If changes are made to other file variables or data structures, those should be covered in addition.

Listing 3.4 shows the scope of the variables and Table 3.2 shows the output of running the source code. This example code shows how a global variable differs from the local variable when both the variables have the same name. In the main function, we use the global variable "user_input_1." In the "second_scenario" function, we declare and initialize "user_input_1" again. However, it is important to note that both the variables have different data types. The main function variable has a *long* data type whereas the function variable declaration has a *short* data type. It can be concluded that there could a bug or vulnerability if the variable with a short data type is accessed and given a value more than it can handle. Let's discuss about the output of the source code shown in Table 3.2. Since the influencer global variable "user_input_1" is declared as a short data type, it causes issues in the "main" function of the code. Therefore, the work/task is never performed. Whereas, in the "second_scenario" function, we neglect the global variable influencer and created a new variable of the same name "user_input_1." Since it is declared as a long data type, it doesn't give any issues and gets the task done. Figure 3.1 shows the links that develop in the source code when the events occur.

Listing 3.4. Source of Error in complex code communities

```c
#include <stdio.h>

short user_input_1 = 68000;

int main()
{
    printf("\n  Running Main function\n\n");
    printf("user_input_1 (global) = %d\n", user_input_1);    //Influencer

    long resulting_user_input_1 = 0;
    //INFLUENCED, GETS incorrect value from the global variable
    resulting_user_input_1 = user_input_1 * 2;

    printf("resulting_user_input_1 (long_local) = %d\n",
    resulting_user_input_1);

    // THIS NEVER RUNS because the  global variable isn't of correct data type
    // Here, the influencer is global user_input_1

    if (resulting_user_input_1 > 30000) //INFLUENCED
    {
        printf("\nOUR CODE IS DOING IMPORTANT WORK_1\n");
        // Consequence of infleuncer (short user_input_1) is that
        // this will never run.
        // This event is influenced by the influencer.
    }

    /******** Run Second Scenario function ********/
    second_scenario();
}

int second_scenario()
{
    printf("\n  Running Second Scenario function\n\n");

    long user_input_1 = 68000; //This is our local influencer.
    //It doesn't depend on the global influencer.

    printf("user_input_1 (long_local) = %d\n", user_input_1);
    long resulting_user_input_1 = 0;

    resulting_user_input_1 = user_input_1 * 2;

    printf("resulting_user_input_1 (long_local) = %d\n",
    resulting_user_input_1);

    // This will run because it's not influenced by the global influencer.
    // Since the local influencer is correct, it works.

    if (resulting_user_input_1 > 30000)
    {
        printf("\nOUR CODE IS DOING IMPORTANT WORK_2\n"); // Consequence
        //of influencer (short user_input_1) is that this will never run.
        //This event is influenced by the influencer.
    }
}
```

Table 3.2. Output from Listing 3.4

Output from different functions of the source code
Running Main function: user_input_1 (global) = 2464, resulting_user_input_1 (long_local) = 4928
Running Second Scenario function: user_input_1 (long_local) = 68000, resulting_user_input_1 (long_local) = 136000

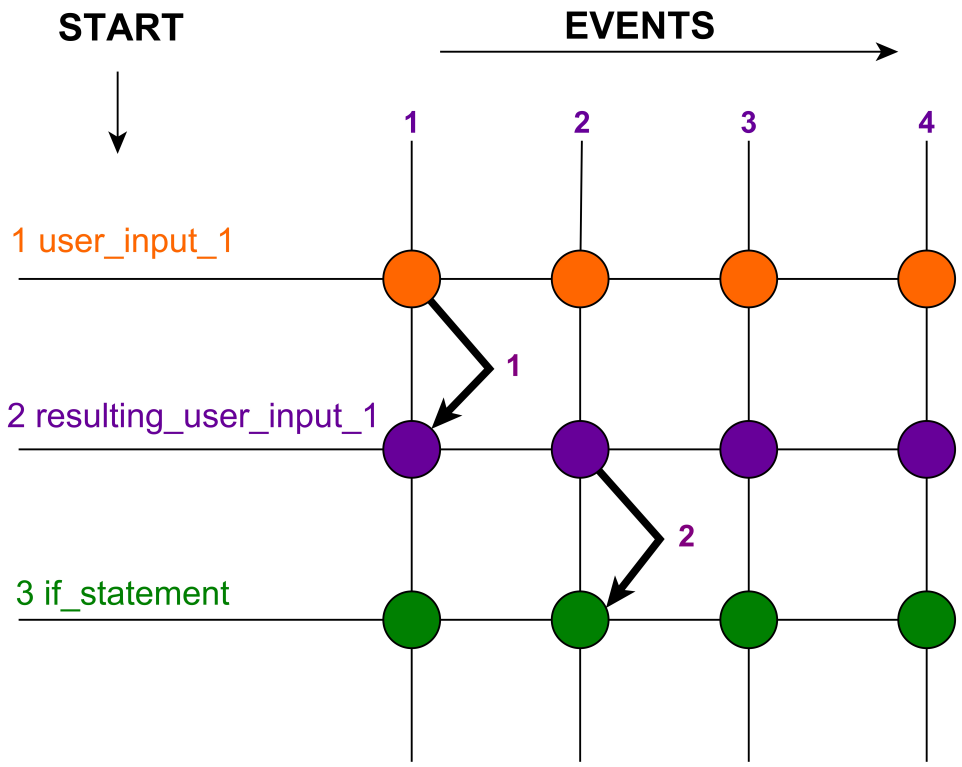

Figure 3.1. Interaction of nodes when events occur

In Listing 3.3 and Listing 3.4, we saw how temporal networks can exist inside the source code. Now, let's take a look at the global variables which can be major influencers (drivers) in the programs. The graph of temporal network framework developed should be able to distinguish between global and local variables with the same names. This way, it will be able to predict *which node* (global or local variable) is the real influencer. Moreover, the temporal network framework should be able to maintain information for both the scenarios.

When we talk about the global variables as an influencer, we also consider any variables that are declared as global or accessible from other source code files.

3.4 Methodology of the temporal network framework

Let's assume we have a C source code file which contains include files, some global variables, and functions. Within functions, we can have local variables, data saving, if statements, curly ("{}") brackets, and other possibilities. Considering this criteria, there are three possibilities in the global and function scope of the source code:

1. There is a global variable(s) and no local variable(s) inside any functions with the same name. In this case, the global declaration and the data type will dominate in the source code.

2. The second case involves having global variable(s) and local variable(s) inside the functions with the same name. If the variable is declared inside the function or curly brackets ("{}"), the local declaration of the variable will dominate.

3. There are no global variable(s) and only local variable(s) exist with the same name. Here, the local variable declaration will dominate.

The above scenarios can also occur when there are multiple source code files in the codebase. However, for our research, we consider one file at a time for the temporal network graph. Ideally, global variables may be located in other source code files so we should consider them as well. The global variables may be updated in other source files so the framework should track the information changed in the global variables by the other files. To narrow down the scope of this study, we don't consider interactions of global variables between different source code files.

3.4.1 Considering types of communities in the source code

Once the framework generates a temporal network graph for a source code file, the generated graph can e further divided into other small temporal network graphs (concept

of smaller communities inside the source code community). Dividing can be done based on curly brackets ("{}"), global variables scope, and functions (main and others). The two possible scenarios are:

1. If the function doesn't use any global variables and only uses the local variables, then it can be considered an independent community.

   (a) This network may have one input link (or more call links) because function parameters may accept a variable. However, if the rest of the variables used are local, then it is an independent community.

   (b) Another possibility is that the function doesn't accept any input to the parameter. It is still an independent community if the variables used are local. It should be noted that the community (function) can have more communities inside it if it contains curly ("{}") brackets.

   (c) Figure 3.2 shows an example of Independent communities. This figure shows a basic temporal network for a C source code file. It includes global variables, the "main" function, and other functions. Also, all the communities are independent because they call other communities using a single link.

2. If the functions use global variables, it won't be an independent community anymore. It can be called as a semi-dependent community because it interacts with the global variables/scope. There can be several links from one semi-dependent community (function_1) to another (global_scope) as shown in Figure 3.3.

3.5 Data tracking mechanism to find the Influencer node of the buggy site

This section discusses the type of communities that could occur in the C source code files and how the framework would be able to find the influencer node of the buggy site. The data tracking graph information used for the framework contains "Network ID" , "Incoming node number", "Link number (Event or line number)", "Affected node number", "Incoming node name", and the "Affected node name."

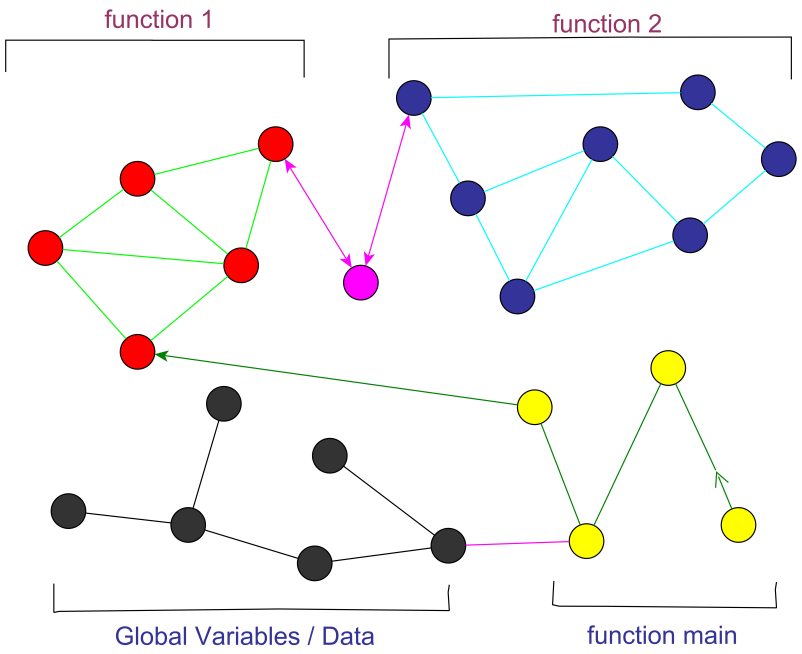

Figure 3.2. Graph showing interaction between different independent communities

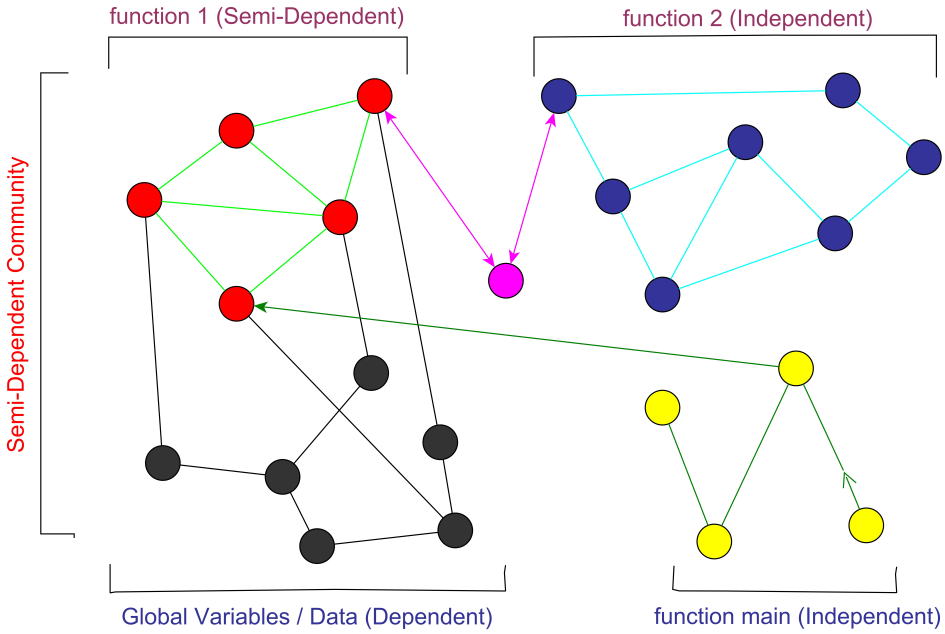

Figure 3.3. Graph showing interaction between different semi-dependent communities

3.5.1 Types of Communities

Section 3.4.1 discussed two types of communities which are "independent communities" and the "semi-dependent communities." Let's focus on independent communities and come up with a way to find an influencer. The nodes in the source code are the variables that are declared or modified. In the code shown in Listing 3.5, "RandomVariable" and "TestVariable" variables are the nodes. Edges are the connections between the nodes. These edges or links are formed when one variable is accessed, modified, or interacts with another variable.

Listing 3.5. Example code with influencer node of the buggy site

```
1 long RandomVariable = 40000;              // Line 1
2 short TestVariable = 0;                   // Line 2
3 TestVariable = RandomVariable * 150       // Line 3
```

Using the information about the nodes and edges, we need to come up with a way to find an influencer in the source code. More specifically, we aim to find the influencer node of the buggy site in the source code. This is where the information regarding the buggy site from static analyzer can be used. Static analyzer will be able to recognize the faulty (buggy) lines of code. In example code (shown in Listing 3.5), we expect the static analyzer to point out that the line 3 is the faulty (buggy) line. This is because "TestVariable" (short data type) can't handle a big value more than 32,767. Therefore, line 3 contains a buggy site. However, *what* should be focused on is the influencer node (or line) in this case. In this case, line 2 should be the influencer node because that's where "TestVariable" is declared as a *short* data type. Fixing line 2 by changing the data type to *long* data type will resolve the buggy site. Therefore, line 2 is the influencer node of the buggy site. "RandomVariable" is an involved node that affects the "TestVariable" (line 3). Moreover, "TestVariable" in line 2 is an involved node. The framework considers the variables on the *right* of the "=" sign as the "incoming nodes." Whereas the variable on the *left* of the "=" sign is the "affected node."

3.5.2 Discussion about the nodes in the source code

In the source code, possible scenarios where variables have interactions are declarations of the variable, initializing a value to the variable, interactions with an if-statement or a while-statement, for-loops, or changing the value of the variable. Moreover, other cases include declaring a structure or accessing the variable of a structure. In the source code temporal network, we need to figure out what is the event number. The possible cases include either making the event number equal to the affected node number or keeping the event number separate from the affected node number.

Why we didn't select the "event number" equal to the "affected node number?" From Listing 3.5, "TestVariable" in line 3 is the affected node. Moreover, "TestVariable" is also the affected node in line 2. Since we know that line 3 contains a buggy site which is indicated by static analysis, we can trace back in our source code temporal network graph and find the first declaration of the "TestVariable." This way, we will arrive to line 2 where "TestVariable" is declared and check its data type. Using this approach, we can successfully locate the influencer node of the buggy site. However, this method is complicated in terms of tracking which affected node occurs at which event. This brings us to the next approach discussed in the next paragraph.

In our another approach, the event number could be different than the affected node number. This is the selected method for our framework. Let's assume the event number is not equal to the affected node number, then what is the affected node number? It can be the number we allot to the variable when it is declared. For example, in line 2 of the Listing 3.5, we can allot affected node number equal to 2 (TestVariable). Now, when the K-tool (created framework) reaches line 3, the affected node number will still be equal to 2 because that is the number given to the node during declaration. To sum it up, the framework gives variable a *number* and whenever the affected variable gets a new value in the source code, the affected node number gets accessed. The benefit of this approach is that the framework will know the first time the affected node was declared which is located on line 2. Modifying the data type of this variable will fix the buggy site located on line 3. However, how to analytically figure out that line 2 needs to be fixed? Line 3 will indicate that the affected

node number is 2. Therefore, the framework looks at all the node numbers in our source code graph and finds the first time *node 2* is declared. This is the influencer node in the program and the framework can also find the corresponding line number where the affected node was first declared. This is also discussed in Section 3.5.4 using a source code example.

3.5.3 Naming convention of the nodes

Incoming (source) node are the nodes (variables) which go to the affected (target) nodes when the event (line) occurs. If multiple incoming nodes are going to the same affected node, all will have the same link number. This is because they all occur on the same line. The methodology considers event number equal to the line number. In the temporal network graph, the link number can be used to find all involved incoming nodes. Moreover, the affected node can be tracked using the incoming node number and link number. The framework should handle the events (line numbers) that don't create or affect any nodes. This is achieved by ignoring the source code line because no useful information is extracted. For example, if a line is empty, the K-tool shouldn't mention it in the temporal network graph. Here are some possible cases considered in the temporal network graph:

1. Incoming (source) node name: These are the nodes which are located on the right of the "=" sign in the source code line. The framework should be able to handle the different types of variables such as int, double, and float. Let's consider an example, if a source code line contains "int Test1 = 100." The framework should create a node for the incoming node "100" which is after the "=" sign as a "val_int_100" node. If some other variable becomes equal to 100 (in some other line), there is no need to create a new "val_int_100" node since it already exists. These nodes will have no link number. Moreover, the incoming and affected node numbers in this case will also be the same. If a line is detected which has an integer/decimal/float data type, the framework creates a node for that number and then processes the line (event). The following discusses how to handle various data types in the source code:

   (a) For char, we create a node for the variable with the convention "char_."

(b) There can be variables declared as a double data type. For example, a line says "double Test1 = 100.93." The K-tool should create a node for "100.93" which is after the "=" sign in the line. The created node is "val_double_100_93."

(c) If the variable data type is float such as "float Test1 = 100.93", the tool should create a node for "100.93" and call it "val_float_100_93."

2. Affected (target) node Name: This will be the node which receives an input from the incoming node(s) when the event occurs. This will have the same link number (line number) as the incoming nodes. This includes all the nodes (variables) that are affected or changed. In most cases, the variable that appears on the left of the "=" in the line is the affected node.

3. Incoming node Number: Incoming node number is the number given to the node when a node (variable) is declared.

4. Affected node Number: It is the number given to the node which gets affected by any incoming node.

5. Link Number: It is same as the line number (event number) in the source code file. The reason behind this decision is that the temporal network graph will be able to track which events (lines) cause what interactions in the code.

6. Declared variable: To deal with a line that just declares a variable, the framework should create a "val_int_0" node. The declared variable has no value so the framework assigns a temporary incoming node to the variable. Moreover, the declared variable is considered as an affected node.

3.5.4 Finding influencer node using temporal network graph

Let's consider the source code shown in Listing 3.6. This code contains buggy sites and the aim is to find the influencer node of the buggy site using the temporal network graph. The static analyzer finds the buggy site (source of fault) line 3 (network id 6). Figure 3.4 shows the temporal network graph generated for the source code file till line number 3. On

network id 6 (where the buggy site is located), the affected node number is 4. Now, using the graph, the first occurrence of "affected node 4" can be tracked. This gives information such as the line number where the variable is declared. This means that changing the data type of the variable on line 4 will resolve the buggy site. Therefore, affected node 4 on line 4 is the influencer node (source of error) of the buggy site. Figure 3.5 shows the interactions between various nodes in the source code when the events occur.

Listing 3.6. Example code with influencer node of the buggy site

```
1    long RandomVariable = 40000;              // Line 1
2    short TestVariable = 0;                   // Line 2
3    TestVariable = RandomVariable * 150;      // Line 3
4    TestVariable = 32000 + 1500;              // Line 4
5    TestVariable = TestVariable + 1500;       // Line 5
6    TestVariable = 0;                         // Line 6: Not a buggy site
7    TestVariable = RandomVariable - 5000;     // Line 7
```

Network id	Incoming (depending) Node Number	Link (event/line) Number	Affected (Final) Node Number	Incoming (depending) Node Name	Affected (Final) Node Name
1	1	-	1	Val_int_40000	Val_int_40000
2	1	1	2	Val_int_40000	RandomVariable
3	3	-	3	Val_int_0	Val_int_0
4	3	2	4	Val_int_0	TestVariable
5	5	3	4	Val_int_150	TestVariable
6	2	3	4	RandomVariable	TestVariable

Figure 3.4. Using Framework to find influencer

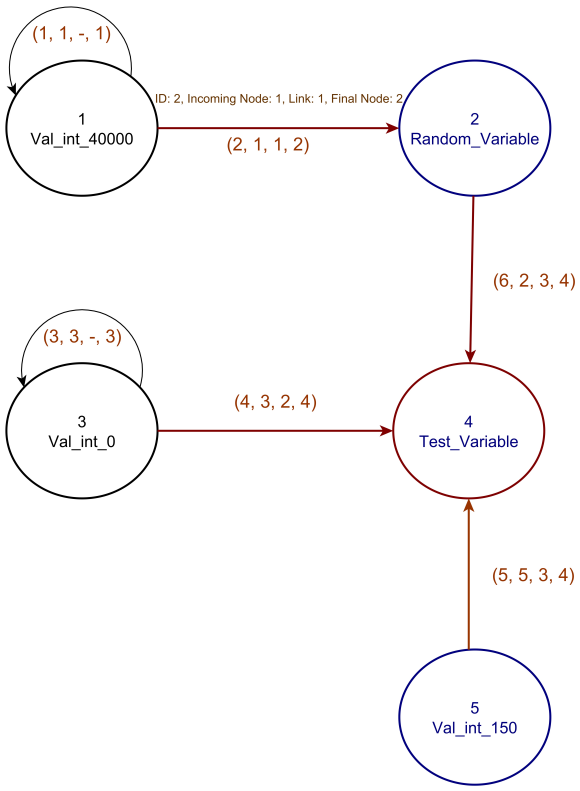

Figure 3.5. Creating the graph for example source code

3.6 Criteria to determine strong influencer vs weak influencer in the source code

Section 3.5.4 shows how the temporal network graph can be used to find the influencer node of the buggy site once the static analysis information is available. However, the effect of the influencer whether it is a strong influencer or a weak influencer should be figured out. Not every influencer node should be considered as a strong influencer. Otherwise, every function may have multiple strong influencers. We come up with a criteria to look at the impact of the influencer node in the community (function). For example, if the influencer node is affecting many nodes inside the community(function), then it can be considered a strong influencer. It is important to define "many" nodes in the community. For example, let's assume that the influencer node of the buggy site affects three other nodes (outgoing edges), then it can be considered a strong influencer. Three nodes is just an example number. The methodology uses the percentage method that can inform if the influencer node is the

dominating node in the community (function). A strong influencer node has more than 50% of the total outgoing edges (links) in the community (function/global). However, if there are only two edges in the community and one of the edge is an outgoing edge from the influencer node, 50% rule will not be successful because it will show the node as a strong influencer. To overcome this problem, the framework uses another method along with a 50% threshold rule. There should be at least *six* outgoing edges inside the community (function) along with the 50% rule. If the influencer node passes this criteria, it will be called a strong influencer node in the source code. All other influencer nodes which don't pass this criteria will be classified as a weak influencer node.

3.7 Using centrality as the other method to find the influencer sites

This section talks about the use of the centrality algorithm to find the influencer sites in the source code files. It should be noted that this method doesn't use static analysis information to find the influencer nodes. However, it finds all the important nodes in the source code file. We used NetworkX tool for this method [29]. The source code was adopted which is free to share and adapt under the "CC BY 4.0" license [30]. The modifications to the source code included extracting source code for the betweenness centrality method and importing the K-tool generated source code graph data (shown in the appendix A.1).

For finding the influencer nodes in the source code, we decided to use the "betweenness centrality" algorithm because of its capability to act as a bridge-like connector in the network [31]. Moreover, the paper states that the removal of the node with the highest betweenness centrality may result in a disconnection between many pairs of the graph. Therefore, the nodes with the highest betweenness centrality act as important nodes. This is the reason this study uses "betweenness centrality" as a method to find the influencer or important nodes in the source code. Betweenness centrality identified nodes will be important because they contain important information between different pairs of nodes. If these nodes get affected, the information in the source code may change. Therefore, the important nodes identified by the betweenness centrality method should be protected to prevent any source code vulnerabilities.

It is important to know that the nodes identified using the betweenness centrality may not be the influencer nodes of the buggy sites. Our speculation is that the influencer nodes of the buggy sites identified using the K-tool will be a subset of the nodes identified by the betweenness centrality.

3.8 Data collection

The experiment is designed to use the K-tool (developed framework) and the betweenness centrality algorithm to find the influencer nodes. The flow diagram for the research process is shown in Figure 3.6. Both the experiments must use the same data set. The data set used should be complex to make sure that the research is done on a source code that is similar to a real software source code or a small operating system. To achieve this, we created a codebase with source code files containing many functions that call other different functions from different *runner* files. Runner files are the source code files which call other source code files and test cases. Moreover, the data set should contain buggy sites in the code so we can find the influencer node of the buggy site in the source code. The steps to create the data set are listed below:

1. Extract the bad and good functions from the Juliet test cases and store them in different files.

2. Create runner files that will randomly call different bad test cases and functions from the other runner files.

3. Run static analysis on the created runner files to extract the buggy sites.

3.8.1 Extraction of bad and good test cases from Juliet Data set

Our study referred to NIST's Juliet data set which contains C/C++ test cases [7]. Initially, there were 41,629 test cases for different classes. For the scope of this research, we decided to only include "CWE121 Stack Based Buffer Overflow", "CWE126 Buffer Overread", and "CWE190 Integer Overflow" test cases. The total number of files for these

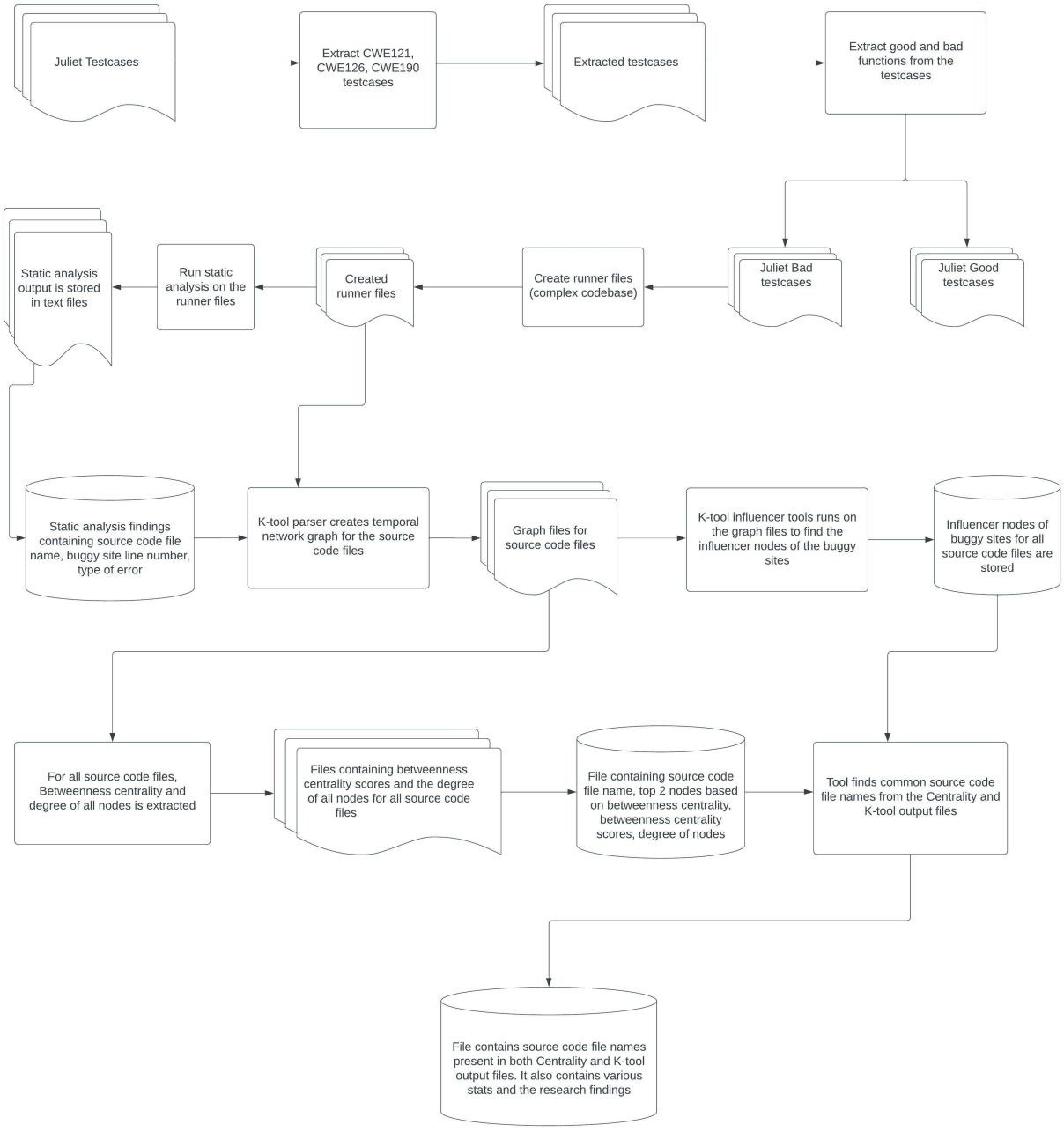

Figure 3.6. Workflow diagram showing the experimental process

categories was 10,366. We decided to remove the C++ source code files and files that contained "typedef" since they were out of the scope of this study. Therefore, the resulting test case files for the experiment were 9,557.

After running the Frama-C's static analysis tool using the "eva" plugin on the bad test case files, we noticed that the static analysis may stop running on a source code file due to a "non-terminating" error if a buggy site is found. We think Frama-C could handle these cases but we skipped exploring how to fix this issue since it wasn't a major concern. Therefore, we decided to remove all the test case files with "non-terminating" error or if no "eva-alarm" was found. The total number of Juliet bad test case files was 915. Listing 3.7 shows an example Juliet test case file that was created after extracting the bad function from the test case source code file.

Listing 3.7. Modified Juliet test case file to extract the bad function

```
// Filename: CWE190_Integer_Overflow__int_fgets_postinc_08.c

#include "std_testcase.h"

#define CHAR_ARRAY_SIZE (3 * sizeof(data) + 2)

void CWE190_Integer_Overflow__int_fgets_postinc_08_bad()
{
    int data;
    /* Initialize data */
    data = 0;
    if(staticReturnsTrue())
    {
        {
            char inputBuffer[CHAR_ARRAY_SIZE] = "";
            /* POTENTIAL FLAW: Read data from the console using fgets() */
            if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL)
            {
                /* Convert to int */
                data = atoi(inputBuffer);
            }
            else
            {
                printLine("fgets() failed.");
            }
        }
    }
    if(staticReturnsTrue())
    {
        {
            /* POTENTIAL FLAW: Incrementing data could cause an overflow */
            data++;
            int result = data;
            printIntLine(result);
        }
    }
}
```

3.8.2 Creating runner files to create a complex data set

To make a complex codebase, we decided to create our test cases called "runner_files" that randomly call functions from other runner files and functions from bad Juliet test cases. The runner file also calls functions from other associated runner files. For example, a function from "runner_1.c" file calls a function located in "runner_5.c" file. Moreover, we grouped 12 runner files together which interact with each other. All these grouped files can be considered as a service and run from a single "runner_for_*.c" file. For example, "runner_for_1_12.c" containing the main function calls all the functions of all associated runner files which are located in "runner_1.c" till "runner_12.c". The idea behind this to consider the data set as a real software or a small operating system containing services such as graphics, sound, etc. The "runner_for_1_12.c" file can be considered as a service file which ends up calling other associated files to accomplish a service task. Listing 3.8 shows the example source code file which calls different functions in other source code files including the Juliet test cases. Listing 3.9 shows the example source code runner file which contains a "main" function and calls functions of all the associated runner files.

The drawback of the created data set is that it is not too complex. In the future, we plan to create more complex runner files by modifying the Juliet test case functions to return an output. After modifying the test cases, we can declare variables in the runner file and store the output of the test case functions in the declared variables. These declared variables in the runner files can be used as input to other functions present in the runner files. This will result in a more complex data set and will contain more influencer nodes of the buggy sites in the data set.

For this study, we created 444 runner files and every 12 files can be considered a service which are called from the "runner_for_*.c" file. This resulted in 37 service (runner_for_*.c) files which contain the "main" function and these are the files on which we will run the static analysis. Why were 12 files grouped as a service? We wanted at least 10 files to be covered as a service. However, there were 915 bad Juliet test cases to be called. To achieve this, we tried creating a group of 10, 11, 12, 13, 14, and 15 files. In this experiment, 12 files were able to maximize the number of Juliet test cases called. Therefore, we decided to group

of 12 files for each service. The number of functions in a every runner file were randomly selected with minimum number of functions equal 3 and the maximum functions up to 5. Also, the reason to create 444 runner files is that 444 is divisible by 12 so all runner files can be associated with a service group runner file. It should be noted that every runner file can call any number of Juliet test cases (selected randomly). The data set creation tool makes sure that no C source code "include file" is included *again* in any other runner file to prevent the infinite looping of included files. When creating 450 runner files that call approximately 915 Juliet test case files, we got the error that there are no more bad test case files to be called. This is because the runner files creation tool was out of bad Juliet test case files to be called. Therefore, we decided to keep the number of runner files less than 450.

Listing 3.8. Generated source code runner_*.c file which calls different functions including test cases

```
1
2  // Filename: runner_277.c
3
4  #include "../testcases/testcases_bad/CWE190_Integer_Overflow...bad.c
5  #include "std_testcase.h"
6
7  void global_some_function_277_1()
8  {
9      global_some_function_283_4();
10 }
11
12 void global_some_function_277_2()
13 {
14     global_some_function_285_2();
15 }
16
17 void global_some_function_277_3()
18 {
19     CWE190_Integer_Overflow__unsigned_int_max_postinc_08_bad();
20 }
21
22
23 int main_runner_277()
24 {
25     global_some_function_277_1();
26     global_some_function_277_2();
27     global_some_function_277_3();
28     return 0;
29 }
```

Listing 3.9. Generated source code for runner_for_*.c file

```
1
2  #include "runner_277.c"
3  #include "runner_278.c"
4  #include "runner_279.c"
5  #include "runner_280.c"
6  #include "runner_281.c"
7  #include "runner_282.c"
8  #include "runner_283.c"
9  #include "runner_284.c"
10 #include "runner_285.c"
11 #include "runner_286.c"
12 #include "runner_287.c"
13 #include "runner_288.c"
14
15
16 int main()
17 {
18     main_runner_277();
19     main_runner_278();
20     main_runner_279();
21     main_runner_280();
22     main_runner_281();
23     main_runner_282();
24     main_runner_283();
25     main_runner_284();
26     main_runner_285();
27     main_runner_286();
28     main_runner_287();
29     main_runner_288();
30     return 0;
31 }
```

3.8.3 Running static analysis on the created runner files

The approach to run the static analysis was adopted from NIST [1]. The static analysis tool was run on 37 "runner_for_*.c" files. The output of the static analysis was stored in a text file. K-tool extracted the important information from all the produced static analysis results files. The information retrieved is shown in table 3.3 The output file containing information of buggy sites for all the source code files was stored in "static_analysis_results.csv" file. This file contained 466 lines (buggy sites). Moreover, the results file had 395 unique buggy sites which means that the source code file names with duplicate line numbers were removed. Table 3.3 shows some of the output of the results file.

Table 3.3. Static Analysis Results file

Source code file name	Buggy site line number	Error type
CWE190_Integer_Overflow___int_rand_square_12_bad	15	unsigned overflow
CWE121_Stack_Based_Buffer_Overflow___CWE129_rand_02_bad	13	unsigned overflow
CWE190_Integer_Overflow___short_rand_multiply_32_bad	14	signed downcast
CWE190_Integer_Overflow___int_fgets_postinc_08_bad	32	signed overflow
CWE126_Buffer_Overread___CWE129_rand_32_bad	26	out of bounds index

3.9 Experiment Design to extract the influencer nodes using the K-Tool

This section discusses the developed framework *K-tool* along with the results. The tool is separated into two sections (discussed below):

1. Creating a parser to extract the incoming (source) nodes, affected (target) nodes from the data set to create a temporal network graph.

2. Using the source code temporal network graph to find the influencer buggy sites from the runner files and the Juliet test cases (data set).

Before finding the influencer node of the buggy site, we extract all the nodes that are present in the source code files using the K-tool parser. The source code files include all the runner files and the bad test case files. Moreover, the static analysis results generated in the Section 3.8.3 are combined in the graph. It should be noted that different test case files may have different data types and could be complex. Due to this, some test case files may not be supported. In the future, we aim to identify the type of test cases not currently supported and modify the parser to accept them. Once one test case file of certain type is supported, all the remaining test case files of that type should become supported.

K-tool parser generates a temporal graph for a source code as shown in Table 4.1 and discussed in Section 4.1. The temporal graphs are used an input to find the influencer nodes of the buggy sites (shown in Table 4.2). Later, the experiment involves finding influencer nodes in the source code files using the betweenness centrality algorithm (shown in Table 4.4). Finally, this study will analyze various stats generated after comparing the influencer nodes of the buggy sites shown by K-tool and the influencer nodes indicated by the betweenness centrality algorithm (Table 4.5).

3.10 Variable

In this study, the expected output is the list of influencers nodes present in the code communities. This list should be independently generated by the K-tool (created framework) and the betweenness centrality algorithm. Our assumption is that the number of influencer nodes indicated by the K-tool will be less than the number of influencers indicated by the betweenness centrality algorithm for the same source code file.
Independent Variables:

1. Number of source code files: This is the number of source code files given as an input to generate a graph.

2. Number of links present in the source code file: This represents the total number of edges in the temporal network graph.

Dependent Variables:

1. Number of Influencers: Comparing the numbers of influencers indicated by the K-tool with the number of influencers indicated by the betweenness algorithm.

2. Influencer nodes list generated by the K-tool is a subset of the list of influencers indicated by the betweenness centrality algorithm for the same source code files.

3.11 Summary

This chapter described the methodology of this study which includes how the K-tool (created framework) will find the influencer nodes of the buggy site. Moreover, the chapter we described the experimental setup to generate the test cases for this research. We also discussed how the K-tool and betweenness centrality algorithm can be used implemented to find the influencer nodes in the source code files.

4. EXPERIMENT RESULTS AND DATA ANALYSIS

This chapter discusses the data collected from the different methods such as K-tool and the betweenness centrality algorithm. It also discusses the insights, analyzes the data, and discusses the performance. Moreover, this chapter talks about the results of this study by comparing the results from the K-tool with the results of the betweenness centrality algorithm.

4.1 Resulting temporal network graph for the variable nodes of a test case

K-tool extracts all the necessary information in a temporal network graph for Listing 3.7 as shown in Table 4.1. It shows that the static analysis tool indicates a buggy site on line 32. However, the influencer of the buggy site is located on *line 9* where "data" is initialized. Therefore, "data" is the influencer node of the buggy site in the source code. The "data" could cause an integer overflow on line 33 where data is an input to the result variable. We also visualized the created graph for the source code file shown in Table 4.1 using Gephi software [32]. The visualized graph is shown in Figure 4.1. It can be noticed that "data" node has the highest interaction in the graph which indicates its potential to be an influencer node.

Table 4.1. Graph Map for a source code file

Network ID	Incoming node number	File line number	Affected node number	Incoming node name	Affected node name	file name	Line has buggy site
1	2	nan	2	val_int_0	val_int_0	CWE190...c	✗
2	4	nan	4	val_int_1	val_int_1	CWE190...c	✗
3	2	9	1	val_int_0	data	CWE190...c	✗
4	2	11	1	val_int_0	data	CWE190...c	✗
5	2	15	3	val_int_0	inputBuffer	CWE190...c	✗
6	2	20	1	val_int_0	data	CWE190...c	✗
7	1	32	1	data	data	CWE190...c	✓
8	4	32	1	val_int_1	data	CWE190...c	✓
9	1	33	5	data	result	CWE190...c	✗

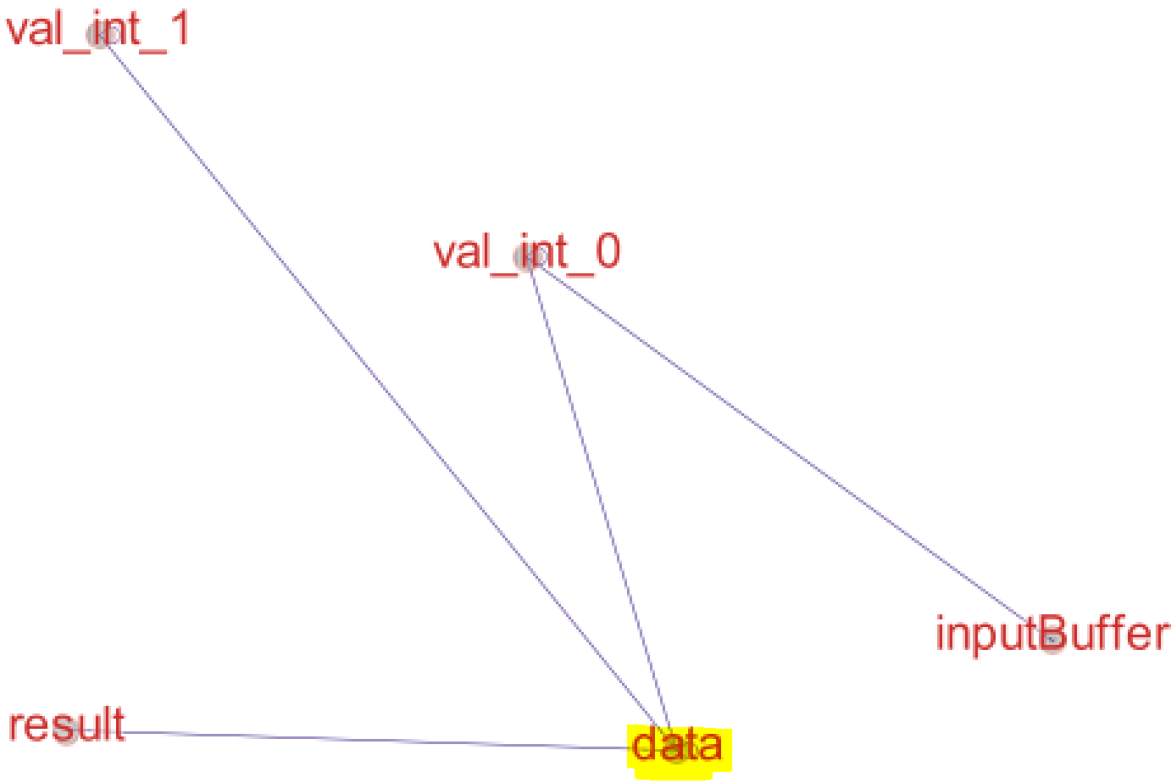

Figure 4.1. Graph created using Gephi shows the source and target nodes

4.2 Finding the influencer node of the buggy site using K-tool

To find the influencer nodes, K-tool accepts temporal network graph files (shown in Section 4.1) as the input files. The methodology developed in Section 3.5.2 and Section 3.5.3 is used to find the influencer node of the buggy sites in the C source code files. The list of influencer nodes of the buggy sites detected by the K-tool is saved in the "all_influencer_info.csv" file. The file contains information such as the "Name of the file with the influencer node of the buggy site", the "line number where the influencer node of the buggy site is detected", and "the name of the variable (or node) which is the influencer node of the buggy site in the source code." There were 241 influencer nodes and 183 unique files in which influencer nodes were located. Some findings are also shown in Table 4.2.

Table 4.2. K-tool results file showing influencer nodes of buggy sites

Source code file name	Line number of the influencer node of the buggy site	Error Variable/Node name
CWE121_Stack_Based_Buffer_Overflow__CWE129_fgets_03_bad	32	buffer
CWE121_Stack_Based_Buffer_Overflow__CWE129_rand_01_bad	7	data
CWE121_Stack_Based_Buffer_Overflow__CWE129_rand_01_bad	14	buffer
CWE126_Buffer_Overread__CWE129_fgets_08_bad	31	buffer
CWE126_Buffer_Overread__CWE129_rand_01_bad	7	data
CWE126_Buffer_Overread__CWE129_rand_01_bad	13	buffer
CWE190_Integer_Overflow__char_fscanf_add_31_bad	16	result
CWE190_Integer_Overflow__char_rand_add_01_bad	7	data
CWE190_Integer_Overflow__char_rand_add_01_bad	13	result
CWE190_Integer_Overflow__int_fgets_postinc_08_bad	9	data
CWE190_Integer_Overflow__int_fscanf_add_03_bad	19	result
CWE190_Integer_Overflow__int_rand_multiply_01_bad	7	data
CWE190_Integer_Overflow__int_rand_multiply_01_bad	15	result

4.3 Influencer Nodes in source code files found using the betweenness centrality

The graphs created by the K-tool parser were used by this algorithm to prevent redundant code and utilizing the existing framework for node extraction. The incoming nodes and affected nodes were identified by the K-tool which can be used as an input for the betweenness centrality algorithm. We identified the betweenness centrality and degrees centrality of all the nodes in the source code file. The results for every source code file were saved in the output files directory. Table 4.3 shows various nodes in a source code file.

Table 4.3. betweenness centrality score for a source code file using NetworkX

Node name	betweenness centrality score	Degree of the node
data	0.833	5
val_int_0	0.5	4
inputBuffer	0.0	1
val_int_1	0.0	3
result	0.0	1

Table 4.3 also shows the betweenness centrality score and the degree of the nodes. From these generated files, we removed the placeholder nodes that were created by the K-tool.

The placeholder nodes include converting "0' to "val_int_0" where "0" was the initialized value of the variable. This resulted in getting only the nodes (or variables) that existed in the source code file.

After observing the results shown in Table 4.3, we decided to keep only the top two nodes based on the betweenness centrality score. However, there is a possibility the other nodes (third highest centrality and more) could be relevant. The option to retrieve other insignificant nodes was added to the tool if needed for future analysis. Moreover, we decided not to include the results for a file if the betweenness centrality for the top node was "0". If the second highest betweenness centrality for the node was "0", it wasn't included. The results for all the source code files were stored in "extracted_combined_centrality_file.csv" file. Some of the output of the results file is shown in Table 4.4.

Table 4.4. Nodes with highest and second highest betweenness centrality score for various source code files

Source code file name	First highest node name	First node's betweenness centrality Score	First node's degree	Second highest node name	Second node's betweenness centrality Score	Second node's degree
CWE121...fgets_01_bad	data	0.2381	2.0	buffer	0.2381	2.0
CWE121...fgets_06_bad	data	0.1786	2.0	buffer	0.1786	2.0
CWE121...fgets_31_bad	data	0.3929	3.0	buffer	0.2143	2.0
CWE121...fgets_32_bad	data	0.5	4.0	buffer	0.1944	2.0
CWE121...fscanf_12_bad	data	0.25	3.0	buffer	0.1389	2.0
CWE121...fscanf_31_bad	data	0.4286	3.0	buffer	0.2381	2.0
CWE121...fscanf_32_bad	data	0.5357	4.0	buffer	0.2143	2.0
CWE121...cpy_12_bad	data	0.05	3.0	-	-	-
CWE121...loop_12_bad	data	0.2321	4.0	source	0.1429	2.0
CWE121...memcpy_12_bad	data	0.0833	3.0	-	-	-
CWE121...32_bad	data	0.6	4.0	-	-	-
CWE126...loop_12_bad	data	0.1	4.0	-	-	-
CWE126...01_bad	data	0.6667	2.0	-	-	-
CWE126...fscanf_12_bad	data	0.3333	3.0	-	-	-
CWE190...31_bad	data	0.8333	3.0	result	0.5	2.0
CWE190...postinc_01_bad	data	1.0	5.0	-	-	-
CWE190...square_04_bad	data	0.3333	2.0	-	-	-

The results (example shown in Table 4.4) indicated at least one node which could be an influencer node with a maximum of two influencer nodes in every source code file. There

were 655 source code files containing the influencer nodes using the betweenness centrality score.

4.3.1 Graphing the influencer nodes indicated by the centrality algorithm

Figure 4.2 shows the graph for the "data" node which the betweenness centrality algorithm indicates as the major influencer node in many Juliet test case files. The "data" node has a minimum degree 2 with up to 7 as the maximum degree. This figure also shows how variable names can be an important indication to find how closely variables interact with the other variables.

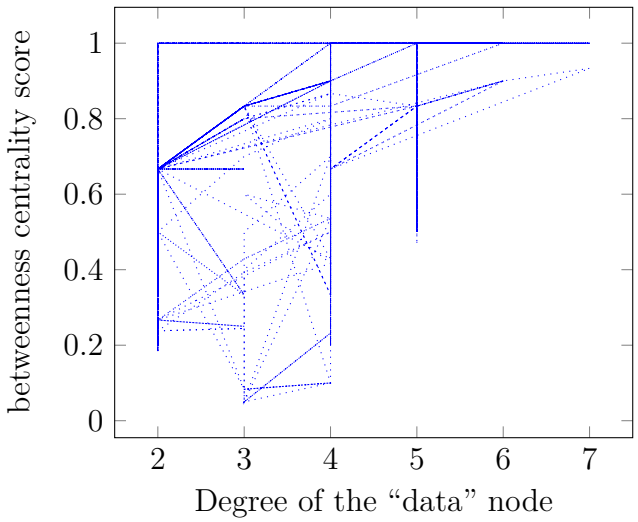

Figure 4.2. Degree vs the betweenness centrality score of the "data" node

We can see from Figure 4.3 that the second-highest influencer node indicated by the betweenness centrality algorithm is "result" . The degree of this node is 2 which means that the "result" node (variable) doesn't interact a lot with the other nodes. Also, *Some* occurrences have *low* betweenness centrality score and a *high* degree of node.

Figure 4.4 shows the betweenness centrality vs the node degree of the "buffer" node in the source code files. *Most* occurrences have *low* betweenness centrality score and a *high* degree of node. The relationship shows degree is 2 with a variable betweenness centrality up to 0.34.

4.3.2 Extracting results from the K-tool and the betweenness centrality

After running the experiment, the results from the two methods were combined into a single results file for comparison. The common source code files were extracted from the "all_influencer_info.csv" file (influencer nodes of buggy sites indicated by the K-tool) and the "extracted_combined_centrality_file.csv" file (influencer nodes indicated by the betweenness centrality algorithm). The results were saved in the "combined_results_analy-

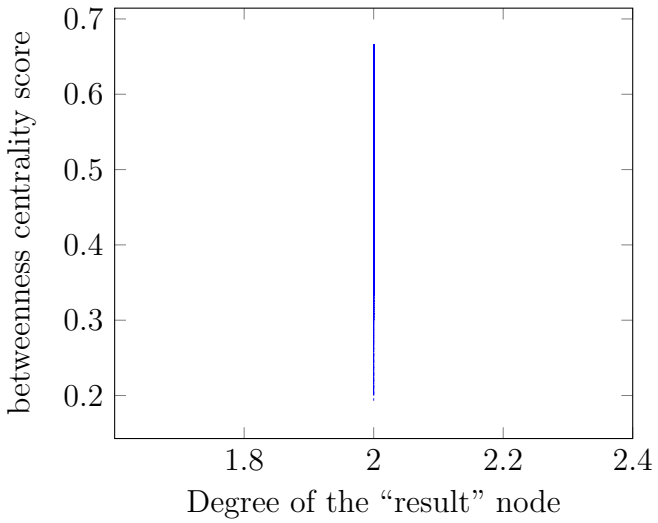

Figure 4.3. Degree vs the betweenness centrality score of the "result" node

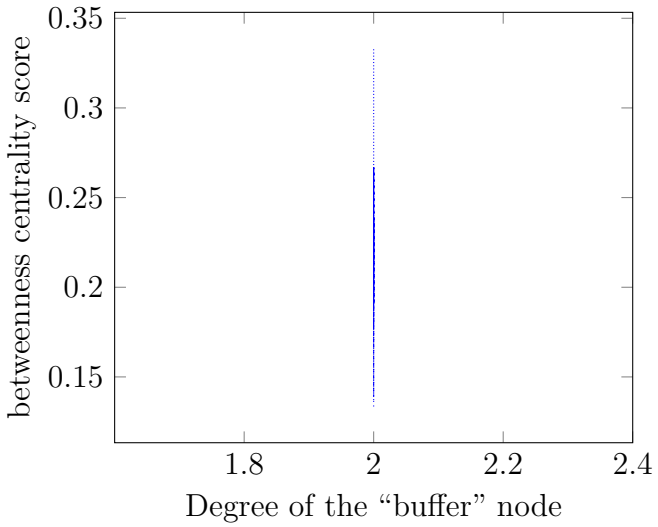

Figure 4.4. Degree vs the betweenness centrality score of the "buffer" node

sis_file.csv" file. There were 183 source code files in which both K-tool and the betweenness centrality algorithm identified influencer nodes. Some of the output is shown in Table 4.5.

Table 4.5. Comparing influencer nodes for K-tool and centrality

Source code file name	K-tool influencer nodes of the buggy sites	centrality influencer nodes	Are all the K-tool nodes present in the nodes indicated by centrality	Are all the centrality nodes present in the nodes indicated by K-tool sites	Are all the K-tool indicated nodes exactly the same as the nodes indicated by centrality
CWE121..02_bad	buffer	data, buffer	✓	×	×
CWE121..12_bad	data	data, source	✓	×	×
CWE126..12_bad	dest	data	×	×	×
CWE126..17_bad	buffer	data	×	×	×
CWE190..12_bad	result	data	×	×	×
CWE190..31_bad	result	data, result	✓	×	×
CWE190..31_bad	data, result	data, result	✓	✓	✓
CWE190..bad	result	data, result	✓	×	×
CWE190..11_bad	data, result	data, result	✓	✓	✓
CWE190..03_bad	data, result	data	×	✓	×

4.4 Analyzing data by comparing the results from the K-tool and the betweenness centrality

Analyzing the Table 4.6 for the common 183 source code files indicated by both the methods, we noticed that the influencer nodes of the buggy sites indicated by the K-tool are present in the influencer nodes indicated by the betweenness centrality algorithm in 72.67% of the source code files. However, the influencer nodes indicated by the betweenness centrality algorithm are present in the influencer nodes indicated by the K-tool in only 63.93% of the source code files. This shows that K-tool can identify the same influencer nodes as the betweenness centrality algorithm in more cases as compared to vice-versa. Therefore, many findings support that K-tool indicated influencer nodes are a subset of the influencer nodes indicated by the betweenness centrality algorithm.

Only 50.81% of the source code files have the same influencer nodes indicated by both K-tool and betweenness centrality. Assuming K-tool indicates all the influencer nodes of the buggy sites in the source code file, the betweenness centrality algorithm indicated all influencer sites as an influencer node of the buggy site in about 51% of the source code files.

Table 4.6. Stats for comparing common influencer nodes between K-tool and centrality

For 183 source code files	Are all the K-tool nodes present in the nodes indicated by centrality	Are all the centrality nodes present in the nodes indicated by the K-tool	Are all the K-tool indicated nodes exactly the same as the nodes indicated by centrality algorithm
	×	×	×
Total present	133	117	93
Percentage	72.67 %	63.93 %	50.81 %

4.5 Comparing influencer nodes results for different classes of Juliet test cases

We decided to analyze the influencer node results from the K-tool and betweenness centrality algorithm based on various classes of the Buffer Framework [17]. Table 4.7 shows that all the influencer nodes indicated by the K-tool are present in betweenness centrality indicated influencer nodes for the Stack-Based Buffer Overflow test cases. This means that we can be confident about finding the influencer nodes using both K-tool and the betweenness centrality algorithm for the Stack-Based Overflow class. However, only 14 influencer nodes for the buggy sites are found by the K-tool which means that "many" Stack-based Buffer Overflow test cases are still not supported.

Table 4.7. Stats for various Juliet classes based on all K-tool nodes in centrality

Class	Are all the K-tool nodes present in the nodes indicated by centrality	Number of testcases present	Total test cases in this Class	Percentage
CWE121 Stack-Based Buffer Overflow	✓	14	14	100 %
CWE126 Buffer Overread	×	0	19	0 %
CWE190 Integer Overflow	×	119	150	79.33 %
Overall	×	**133**	**183**	**72.67 %**

For the Buffer Overread class test cases (Table 4.7), there are no common influencer nodes between the two methods which indicates that the results for Buffer Overread from the betweenness centrality algorithm may not be used as trusted information of influencer nodes of buggy sites. In 79.33% of Integer Overflow-based test cases, K-tool influencer nodes are present in the nodes indicated by betweenness centrality. This shows that the K-tool

influencer nodes of buggy sites are important nodes in the source code file. We make this claim because betweenness centrality algorithm informs about all the important nodes in the graph.

Table 4.8 shows that only 6 (42.86 %) Stack-Based Buffer Overflow source code files have all betweenness centrality indicated nodes in the K-tool indicated nodes for the same files. It is interesting that 8 source codes for Buffer Overread class have betweenness centrality indicated nodes in the K-tool nodes set whereas this value was 0 for K-tool influencer nodes in the betweenness centrality nodes. Therefore, we can infer that K-tool finds more influencer nodes than the betweenness centrality algorithm for the Buffer Overread test cases. For the Integer Overflow test cases, we can see that less number (68.66 %) of influencer nodes indicated by the betweenness centrality are present in the K-tool indicated influencer nodes as compared to the K-tool influencer nodes (79.33 % as shown in Table 4.7) present in the betweenness centrality influencer nodes for the same class.

Table 4.8. Stats for various Juliet classes based on all centrality nodes in K-tool

Class	**Are all the centrality nodes present in the nodes indicated by K-tool**	**Number of testcases present**	**Total test cases in this Class**	**Percentage**
CWE121 Stack-Based Buffer Overflow	✗	6	14	42.86 %
CWE126 Buffer Overread	✗	8	19	42.11 %
CWE190 Integer Overflow	✗	103	150	68.66 %
Overall	✗	**117**	**183**	**63.93 %**

Only 87 source code files (out of 150) which is 58 % of of the source code files of the Integer Overflow class have exactly the same influencer nodes for both the K-tool and the betweenness centrality algorithm. This is shown in Table 4.9.

From the Tables 4.7, 4.8, 4.9, we can conclude that K-tool currently works best for Integer Overflow based source code files. The betweenness centrality algorithm and the K-tool both indicate highest number of common influencer nodes CWE190 Integer Overflow class.

Table 4.9. Stats for various Juliet classes based on all K-tool nodes exactly the same as centrality

Class	Are all the K-tool indicated nodes exactly the same as the nodes indicated by centrality	Number of testcases present	Total test cases in this Class	Percentage
CWE121 Stack-Based Buffer Overflow	✓	6	14	42.85 %
CWE126 Buffer Overread	×	0	19	0 %
CWE190 Integer Overflow	×	87	150	58 %
Overall	×	**93**	**183**	**50.81 %**

4.5.1 Comparing Performance of the K-tool vs the betweenness centrality

K-tool consists of two parts: 1) a parser to generate the temporal network graphs and 2) a tool to find the influencer node of buggy site using the temporal network graphs and the static analysis information. Table 4.10 shows the execution time to run static analysis tool on the created data set service runner files. Running the static analyzer on the service runner file also runs the static analysis on the associated Juliet test cases.

Table 4.10. Static Analysis Execution time

Number of runner files processed	Number of runner files with buggy sites (indicated by static analysis)	Execution time (in seconds)
37	11	46

In our experiment, the temporal network graph generated by the K-tool parser is used by both K-tool and the betweenness centrality algorithm. Therefore, temporal network graphs play an important role in finding the influencer nodes in the source code files. Figure 4.5 shows the time taken by the K-tool and betweenness centrality algorithm to find the influencer nodes in the Juliet test cases. K-tool performs much faster and takes 1/5 of the time taken by the betweenness centrality algorithm to find the influencer. However, it should be noted that Figure 4.5 doesn't consider the amount of time taken to run static analysis on the test case files. To find influencer nodes, betweenness centrality algorithm requires the

affected (target) nodes and incoming (source) nodes information whereas the K-tool requires affected nodes, incoming nodes, static analysis buggy site line number, and source code line number. Since K-tool requires static analysis information, an additional process is added which may increase the time to get the influencer nodes information. It should be noted that static analysis tool execution time may differ based on the static analysis tool used.

K-tool and betweenness centrality algorithm methods both used the same temporal network graph to find the influencer nodes. Therefore, it is fair to call K-tool faster than the betweenness centrality method to find the influencer node in the source code files.

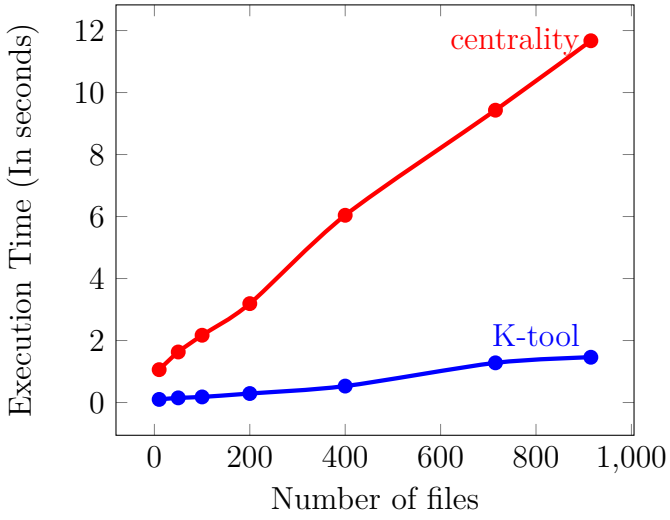

Figure 4.5. Processed files vs. Execution Time

Based on the results we have seen in Section 3.9 and Section 4.4, we can answer Research Question 1 by concluding that it is possible to find the influencer node of the buggy site in the source code using temporal networks (K-tool) if the buggy sites are identified using static analysis. Moreover, Section 4.5.1 answers Research Question 2 and concludes that it is faster to find the influencer node of the buggy site using the K-tool than the betweenness centrality algorithm .

4.6 Summary

In this chapter, we showed the temporal network graphs created for the source code files and how K-tool uses them to find the influencer nodes of the buggy sites. Later, the chapter

mentioned the results of betweenness centrality method to find the influencer nodes. This chapter also analyzed the influencer nodes results from both K-tool and the betweenness centrality methods. In the end of the chapter, the performance of the two different methods (K-tool and betweenness centrality) to find influencer nodes is presented.

5. DISCUSSION AND FUTURE WORK

This chapter addresses the experiment process and discusses the important findings. Moreover, we compare the findings from the K-tool and the betweenness centrality algorithm. The chapter later talks about the unanticipated findings, limitations, weaknesses of the research and the future research recommendations.

5.1 Addressing Experiment Environment

1. The experiments have been performed on Ubuntu (20.04.3 LTS) WSL. The Frama-C version used is "21.1 Scandium." The performance could vary on the different operating systems.

2. The K-tool requires static analysis results to accurately find the influencer node of the buggy site in the source code. The time taken to run static analysis on one service runner (runner_for_*.c) file is less than the time taken to run static analysis tool on all the source code files individually. This approach was adopted to reduce the overall time taken to find an influencer node of buggy site using the K-tool.

5.2 Research problem and the findings

The research focuses on tackling the problem of finding *what* is important to fix so that the buggy sites present in the source code can be resolved. As shown in Section 4.2, K-tool is able to find the influencer nodes of the buggy sites in the source code. To accurately find the influencer node, K-tool utilizes buggy site information (line number) from the static analysis tool. It is important to mention that the static analysis results had 395 buggy sites (as shown in Section 3.8.3) whereas the K-tool indicated 241 influencer nodes for the buggy sites (Section 4.2). Therefore, it can be concluded that 154 influencer nodes of the buggy sites may not have been identified by the K-tool. Our speculation is that some types of Juliet test case source code files may not be currently supported by the K-tool and should be addressed in the future work of this research.

5.3 Interpreting Findings and their Importance

The influencer nodes of the buggy sites indicated by the K-tool are important due to their accuracy. The have the ability to inform the software developers how to easily fix the vulnerability by knowing where the influencer node (variable) of the buggy site is declared. On the other hand, the betweenness centrality algorithm identifies all the influencer nodes but they all may not be the cause of the buggy site. Betweenness centrality algorithm indicated influencer nodes tell developers which are the important nodes in the source code and these nodes should be handled carefully. For example, if the variable is an influencer node, is it of the correct data type? Developers could check if the minimum and maximum value the variable can handle is checked using an if-statement before assigning any value to the variable? It should be noted that number of nodes indicated by the betweenness centrality algorithm information may be a lot more than what K-tool indicates.

Since the centrality algorithm involves the betweenness centrality and the degree of the node, I decided to interpret if there is any relationship between the betweenness centrality score of the node and the degree of the node. This was done by plotting the graphs for some of the important influencer nodes. Figure 4.2 shows that the "data" node reaches the maximum (1 out of 1) betweenness centrality score in many some cases. This indicates that this node is be a strong influencer node in the source code files. Moreover, it can be inferred from the graph that a node with a maximum centrality score may not have high degree. For example, nodes with degrees 2 till degree 7 all achieved the maximum possible betweenness centrality score. The "star" and "mix" classification for the centrality are discussed in [27]. Many occurrences of the "data" node fall into *star* classification because of high degree and high betweenness centrality score.

Our speculation regarding Figure 4.3 is that the "result" node may not be a very strong influencer node in the source code file due to less interaction with the other nodes (degree of the node is 2). The maximum betweenness centrality for this node is less than 0.7 which is a slightly lower as compared to the betweenness centrality score of the "data" node. Regarding the classification, some occurrences of the "result" node will fall in the *mix* classification because some occurrences have low betweenness centrality score and high degree. From Fig-

ure 4.4, it can be seen that "buffer" node occurs as a weak influencer node as compared to the "data" and "result" nodes. We speculate this because of the dimmed dotted line after betweenness centrality score reaches the value of 0.275. The degree of this node is 2 which is same as the degree of "result" node. However, the major difference lies between the maximum betweenness centrality score of the "result" and the "buffer" node. The maximum betweenness centrality of the "buffer" node is less than 0.35. This value is 1/2 of the maximum betweenness centrality score achieved by the "result" variable node. Only some of the occurrences of the "buffer" node will fall into *mixed* classification.

The results (Section 4.2) show that this study was able to identify the influencer nodes of the buggy sites for various classes (Section 1.5) of test cases. However, how do the results of this study compare to the natural software source code? We think that K-tool will be able to identify the influencer nodes of the buggy sites in all source code files which are designed similar to the test cases currently supported in this research. There is a possibility that some data types such as "typedef" may not be supported because they were not considered when developing the K-tool. On the other hand, we expect that the betweenness centrality algorithm will notify the user about the influencer nodes in the source code which may or may not be the cause of a buggy site. It should also be noted that K-tool provides the information such as the location of the influencer node (line number) whereas the betweenness centrality method doesn't provide the information regarding the location of the influencer node. Therefore, K-tool makes it easier for the software developers to find the influencer node, and this node can be fixed more quickly using the K-tool as compared to the betweenness centrality method.

5.4 Comparing the Findings

In the literature review, we discussed that there has been research to find the nodes in the Linux kernel source code where functions are the nodes and relationship exists in the source code between the functions and the service calls [24]. However, in our research, nodes are the variables and every link depicts a relationship of variables with the other variables in the source code. There isn't much literature available on these experiments. For our

study's comparison, we compare the results of the K-tool (developed) with the betweenness centrality algorithm. K-tool indicated 183 source code files with influencer nodes of the buggy sites whereas the betweenness centrality algorithm found 655 source code files with influencer nodes (may or may not cause a buggy site). Assuming K-tool influencer nodes are the ground truth, we can conclude that the centrality algorithm found influencer nodes in all the files which were indicated by the K-tool.

The betweenness centrality method found the influencer nodes in more than 472 source code files which were not indicated by the K-tool. We speculate that these 472 source code files may have influencer sites but they may not be a source of a buggy site. The other reason could be that the K-tool currently doesn't support those source code files. Thus, K-tool fails to find any influencer nodes. Another possible reason could be that the static analysis may have halted when it ran on the service runner files. This could have prevented the extraction of the line with the buggy site. This eventually resulted in K-tool not being able to find influencer nodes in those source code files.

5.5 Unanticipated Findings

1. The K-tool indicated "dest" node (variable) as the influencer node of the buggy site whereas the centrality algorithm mentions "data" as the influencer node in the "CWE126_Buffer_Overread___wchar_t_declare_loop_12_bad.c" file. In my opinion, both "data" and "dest" could be the influencer nodes of the buggy sites. The "data" is an incoming node to the "dest" variable. The "dest" node became an influencer node and could be fixed by changing the size of the "dest" variable equal to the size of "data". The other way to fix the issue could be to add an if-statement to check the length of the "data" variable in the source code. Influencer nodes may not be the same from the two methods (K-tool and betweenness centrality) and it could be possible to fix the vulnerabilities using many approaches.

2. Some cases may have the "influencer node of the buggy site" and the "buggy site" on the same line. For example, the static analyzer indicated buggy site on line 16. K-tool also shows influencer node ("result") of the buggy site on line 16 in the "CWE190_In-

teger_Overflow___char_fscanf_add_31_bad.c" file. However, the betweenness centrality algorithm indicates both "data" and "result" as the influencer nodes. The buggy site is located in the line "char result = data + 1;" where "data + 1" could cause an overflow. The "result" node is an influencer node because it will be distributing the incorrect information. This is how "data" variable obtains the incorrect information. However, it can be argued that "data" is an influencer node that gives the incorrect information in the first place and if the "data" provided information was correct, the "result" variable won't have any vulnerability. Therefore, information from the betweenness centrality algorithm is more useful in this case. It also builds up a case for future research that K-tool should be able to track the "incorrect information providing variable" when the buggy site and influencer node of the buggy site are on the same line.

5.6 Limitations and Weaknesses

1. In the test case "CWE190_Integer_Overflow___int_fscanf_postinc_32_bad.c", we observed that the "data" variable was declared multiple times using the curly brackets ("{}"). This means that there can be multiple smaller communities inside the source code community which K-tool doesn't currently support. The "data" declaration line location indicated by the K-tool isn't the real location where "data" is originally declared in the source code.

2. The K-tool influencer nodes are highly dependent on the results of the static analysis tool. In our process, we couldn't identify the buggy sites in all the available test cases due to a "non-terminating" error which we Frama-C could support but we didn't dive into the cause. This is why certain test case types may not be currently supported by the K-tool.Moreover, the "Manifest" file available for the Juliet test cases suite can be used to find the location of the buggy site for all the test cases [1].

3. The data set doesn't have cases where the "return" value is returned. Therefore, the test cases have "void" functions and we don't have any test cases where the value

returned from the function is stored in the variable. This is why there isn't much interaction or flow of data between the different source code files.

4. Due to the above-listed limitation of the data set, the K-tool doesn't currently handle tracking of the influencer nodes that arise from one source code file and may affect some other source code file where the function is called. This is the current weakness of the framework and can be considered as a part of future work.

5. "Typedef", "wchar", "pointers" (*), "float", "double", and other data types in C language may not be currently supported by the K-tool.

5.7 Recommendations for Further Research

1. K-tool should provide information such as the data type of the influencer node of the buggy site.

2. K-tool should have the capability of recommending to the end-user how to fix the buggy site. For example, K-tool should inform the data type of the variable that needs to be fixed and a new recommended data type. It should also provide possible instructions which can assist in fixing the buggy site. This includes recommending the user to add an if-statement to check if the input value to a variable will exceed what the data type of the variable can handle.

3. K-tool doesn't currently support a mechanism to inform the user how bad is the influencer node in the source code. This could be achieved by using the "degree of the node" information. This could help users to identify which nodes to fix in terms of their importance and the overall effect.

4. The future research should modify the K-tool to indicate if the influencer is a "strong" or a "weak" influencer as discussed in Section 3.6. Currently, the betweenness centrality algorithm method implemented just considers the top-two influencer nodes based on their betweenness centrality score. We should consider other influencer nodes in the source code as well after a certain threshold of the betweenness centrality score.

5. The data set should be made more complex by having more flow of information between various source code files.

6. To improve the K-tool, and to add more support for different test cases, the future work should include finding out which test case types which don't have any influencer node indicated by the K-tool. Then, these identified test cases indicated should be investigated and used to make K-tool support them.

6. CONCLUSION

This study starts with understanding what static analysis tools currently output regarding the vulnerabilities. Static analysis tools are an important part to fix the bugs that occur in the source code. This study finds a unique way to cover the bridge between the output of static analysis tool and the information received by the developers to fix the vulnerabilities.

We learned how to run the static analysis tools to find the buggy sites (line numbers) in the source code [1]. This has been a critical part in the development of the K-tool framework. Later, our research dived into how the framework should be developed and how it will be able to deal with the source code communities and still find the influencer node of the buggy site. The part of the methodology was converted into an actual software tool called *K-tool* which created temporal network graphs using the source code files. Moreover, we discussed how this research led to the development of a complex codebase to mimic a software or a small operating system. This developed codebase utilized Juliet testcases [20]. The study also explored the betweenness centrality algorithm to see if it could potentially find influencer nodes in the source code.

This research aimed on finding the influencer nodes of the buggy sites from the source code files using the static analysis information. As discussed in Section 4.2, we were able to find the node (variable) name and the location (line number) of the influencer node of buggy site using a K-tool. Moreover, we noticed that the K-tool is faster than the betweenness centrality algorithm in finding the influencer nodes (if the static analysis information is available). It is important to note that betweenness centrality algorithm found more influencer nodes than the K-tool for the same dataset. We discussed various speculations regarding this outcome. In our research, we also noticed that having static analysis tools buggy site line number (as a prerequisite) can slow down the process of finding the influencer nodes using the K-tool. Chapter 4 discussed how the betweenness centrality algorithm finds the influencer nodes and not the "influencer nodes of the buggy sites." This means that the influencer node may not be the cause of error in the source code but it would be an important node which should be handled carefully and have correct data type to prevent any vulnerabilities. An important aspect about the betweenness centrality is that it doesn't require the static

analysis results to find the influencer node as compared to the K-tool which requires static analysis buggy site line number. This is the reason K-tool may have better accuracy in finding the influencer node of the buggy site but may require more run time than the betweenness centrality algorithm. This means betweenness centrality algorithm could performs faster since it doesn't require the static analysis process.

The research finally concludes by discussing which various Juliet classes have more influencer nodes indicated by the K-tool than the betweenness centrality. The research shows that Integer Overflow is supported more by the K-tool as compared to the other Juliet test case classes. Later, the study compares the performance of the K-tool and the betweenness centrality algorithm in terms of the time to process a specific number of files. Chapter 4 ends by discussing the two research questions of this study.

REFERENCES

[1] P. E. Black, P. E. Black, and K. S. Walia, *SATE VI Ockham Sound Analysis Criteria*. US Department of Commerce, National Institute of Standards and Technology, 2020.

[2] A. M. Delaitre, B. C. Stivalet, P. E. Black, V. Okun, T. S. Cohen, A. Ribeiro, *et al.*, "Sate v report: Ten years of static analysis tool expositions," 2018.

[3] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 240–253, 2006.

[4] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis.," in *USENIX security symposium*, vol. 14, 2005, pp. 18–18.

[5] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Communications of the ACM*, vol. 61, no. 4, pp. 58–66, 2018.

[6] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" In *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 672–681.

[7] P. Black, *Juliet 1.3 test suite: Changes from 1.2*, en, 2018-06-14 2018. DOI: https://doi.org/10.6028/NIST.TN.1995.

[8] A. Paranjape, A. R. Benson, and J. Leskovec, "Motifs in temporal networks," in *Proceedings of the tenth ACM international conference on web search and data mining*, 2017, pp. 601–610.

[9] M. Singh, R. N. Mahia, and D. M. Fulwani, "Towards characterization of driver nodes in complex network with actuator saturation," *Neurocomputing*, vol. 201, pp. 104–111, 2016.

[10] R. Haghighi and H. Namazi, "Algorithm for identifying minimum driver nodes based on structural controllability," *Mathematical Problems in Engineering*, vol. 2015, 2015.

[11] P. Zhang, Z. Ji, and Z. Li, "Minimum driver nodes selection in complex networks," in *2017 36th Chinese Control Conference (CCC)*, IEEE, 2017, pp. 8461–8466.

[12] X. Wang, Y. Xi, W. Huang, and S. Jia, "Deducing complete selection rule set for driver nodes to guarantee network structural controllability," *IEEE/CAA Journal of Automatica Sinica*, vol. 6, no. 5, pp. 1152–1165, 2017.

[13] B. Blonder, T. W. Wey, A. Dornhaus, R. James, and A. Sih, "Temporal dynamics and network analysis," *Methods in Ecology and Evolution*, vol. 3, no. 6, pp. 958–972, 2012.

[14] T. P. Peixoto and M. Rosvall, "Modelling sequences and temporal networks with dynamic community structures," *Nature communications*, vol. 8, no. 1, pp. 1–12, 2017.

[15] B. Chess and G. McGraw, "Static analysis for security," *IEEE security & privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[16] F. Caset, S. Blainey, B. Derudder, K. Boussauw, and F. Witlox, "Integrating node-place and trip end models to explore drivers of rail ridership in flanders, belgium," *Journal of Transport Geography*, vol. 87, p. 102 796, 2020.

[17] I. Bojanova, P. E. Black, Y. Yesha, and Y. Wu, "The bugs framework (bf): A structured approach to express bugs," in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, 2016, pp. 175–182.

[18] I. Bojanova, Y. Yesha, and P. E. Black, "Randomness classes in bugs framework (bf): True-random number bugs (trn) and pseudo-random number bugs (prn)," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, IEEE, vol. 1, 2018, pp. 738–745.

[19] I. Bojanova and C. E. Galhardo, "Classifying memory bugs using bugs framework approach," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, IEEE, 2021, pp. 1157–1164.

[20] F. Meade, "Juliet test suite v1. 2 for c/c++ user guide," *no. December*, 2012.

[21] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion," in *2012 34th International Conference on Software Engineering (ICSE)*, IEEE, 2012, pp. 69–79.

[22] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *International static analysis symposium*, Springer, 2001, pp. 40–56.

[23] L. Wang, Z. Wang, C. Yang, L. Zhang, and Q. Ye, "Linux kernels as complex networks: A novel method to study evolution," in *2009 IEEE International Conference on Software Maintenance*, IEEE, 2009, pp. 41–50.

[24] Y.-F. Wang and D.-W. Ding, "Topology characters of the linux call graph," in *2015 2nd International Conference on Information Science and Control Engineering*, IEEE, 2015, pp. 517–518.

[25] L. Ying and D.-w. Ding, "Topology structure and centrality in a java source code," in *2012 IEEE International Conference on Granular Computing*, IEEE, 2012, pp. 787–789.

[26] S. Gómez, "Centrality in networks: Finding the most important nodes," in *Business and Consumer Analytics: New Ideas*, Springer, 2019, pp. 401–433.

[27] M. J. Mišić, J. Ž. Protić, and M. V. Tomašević, "Improving source code plagiarism detection: Lessons learned," in *2017 25th Telecommunication Forum (TELFOR)*, IEEE, 2017, pp. 1–8.

[28] G. Stergiopoulos, P. Kotzanikolaou, M. Theocharidou, and D. Gritzalis, "Risk mitigation strategies for critical infrastructures based on graph centrality analysis," *International Journal of Critical Infrastructure Protection*, vol. 10, pp. 34–44, 2015.

[29] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11–15.

[30] C. N. W. John R. Ladd Jessica Otis and S. Weingart, *Exploring and analyzing network data with python*, https://doi.org/10.46430/phen0064, [Online], 2017.

[31] L. Lu and M. Zhang, "Edge betweenness centrality," in *Encyclopedia of Systems Biology*, W. Dubitzky, O. Wolkenhauer, K.-H. Cho, and H. Yokota, Eds. New York, NY: Springer New York, 2013, pp. 647–648, ISBN: 978-1-4419-9863-7. DOI: 10.1007/978-1-4419-9863-7_874. [Online]. Available: https://doi.org/10.1007/978-1-4419-9863-7_874.

[32] M. Bastian, S. Heymann, and M. Jacomy, *Gephi: An open source software for exploring and manipulating networks*, 2009. [Online]. Available: http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154.

A. SOURCE CODE

A.1 Source code for the Centrality Algorithm

The following source code was obtained from https://doi.org/10.46430/phen0064 which is available under the CC-BY 4.0 license. We extracted only the source code for the betweenness centrality algorithm and modified it to use our data set [30].

Listing A.1. Betweenness centrality source code

```
#!/usr/bin/env python3
# Source code source : https://doi.org/10.46430/phen0064
# Source code modified by : Kanwardeep Singh Walia
#
import os
import sys
import csv
from operator import itemgetter
import networkx as nx
from networkx.algorithms import community #This part of networkx,
# for community detection, needs to be imported separately.

directory_to_process = sys.argv[1]
directory_to_process = os.path.expanduser(directory_to_process)

save_file_location = sys.argv[2]
save_file_location = os.path.expanduser(save_file_location)

ALL_NODES_FILE_PREFIX = "all_nodes_"
ALL_NODES_FILE_SUFFIX = ".csv"

SOURCE_TARGET_NODES_FILE_PREFIX = "source_target_"
SOURCE_TARGET_NODES_FILE_SUFFIX = ".csv"

output_method_b_centrality_info_file_prefix = "centrality_info_"
output_method_b_centrality_info_file_suffix = ".csv"

output_method_b_additional_info_file_prefix = "additional_info_"
output_method_b_additional_info_file_suffix = ".csv"

output_graph_gexf_file_prefix = "graph_gexf_"
output_graph_gexf_file_suffix = ".gexf"

def get_file_name_without_path(file_name, character_to_remove="/"):
    while file_name.__contains__(character_to_remove):
        slash_index = file_name.index(character_to_remove)
        file_name = file_name[slash_index + 1:]
    return file_name

def find_centrality(this_all_nodes_file_with_path,  \
    this_source_target_file_name_with_path):
    # Read in the nodelist file
    with open(this_all_nodes_file_with_path, 'r') as nodecsv:
        nodereader = csv.reader(nodecsv)
```

```
        nodes = [n for n in nodereader][1:]

    # Read in the edgelist file
    with open(this_source_target_file_name_with_path, 'r') as edgecsv:
        edgereader = csv.reader(edgecsv)
        edges = [tuple(e) for e in edgereader][1:]

    #### TO SAVE ####
    file_name_without_path = get_file_name_without_path( \
    this_all_nodes_file_with_path, character_to_remove="/")

    file_name_without_prefix_suffix =  \
    get_file_name_without_all_nodes_prefix_suffix( \
    file_name_without_path, all_nodes_file_prefix= \
    ALL_NODES_FILE_PREFIX, all_nodes_file_suffix=ALL_NODES_FILE_SUFFIX)

    output_additional_info_csv_file_name =  \
    output_method_b_additional_info_file_prefix +  \
    file_name_without_prefix_suffix +  \
    output_method_b_additional_info_file_suffix

    output_centrality_csv_file_name =  \
    output_method_b_centrality_info_file_prefix +  \
    file_name_without_prefix_suffix +  \
    output_method_b_centrality_info_file_suffix

    output_graph_gexf_file_name = output_graph_gexf_file_prefix +  \
    file_name_without_prefix_suffix + output_graph_gexf_file_suffix

    ##### GIVEN SOURCE CODE (MODIFIED) STARTS ########
    # Get a list of just the node names (the first item in each row)
    node_names = [n[0] for n in nodes]

    # Print the number of nodes and edges in our two lists
    print(len(node_names))
    print(len(edges))

    G = nx.Graph() # Initialize a Graph object
    G.add_nodes_from(node_names) # Add nodes to the Graph
    G.add_edges_from(edges) # Add edges to the Graph

    graph_info = nx.info(G)
    print(graph_info) # Print information about the Graph

    density = nx.density(G)
    print("\n\nNetwork density:", density)

    degree_dict = dict(G.degree(G.nodes()))
    nx.set_node_attributes(G, degree_dict, 'degree')

    sorted_degree = sorted(degree_dict.items(), key=itemgetter(1), \
        reverse=True)

    file_additional_info_open = open(save_file_location + \
    output_additional_info_csv_file_name, "w")

    file_additional_info_open.write(str(graph_info) + "\n" +  \
    "Average Density : " + str(density) + "\n")
```

```
file_additional_info_open.write("\n" + "TOP 20 NODES BY DEGREE"  \
+ "\n")

# TOP 20 Nodes are:

print("\n\n Top 20 nodes by degree: \n")
for d in sorted_degree[:20]:
    print(d)
    file_additional_info_open.write(str(d) + "\n")

############ CENTRALITY
print("\n\n\n           CENTRALITY:\n\n")

print("\n\nABOUT TO PRINT CENTRALITY OF THE FOLLOWING NODE: ")

# betweenness centrality
betweenness_dict = nx.betweenness_centrality(G)

# Assign each to an attribute in your network
nx.set_node_attributes(G, betweenness_dict, 'betweenness')
sorted_betweenness = sorted(betweenness_dict.items(),  \
key=itemgetter(1), reverse=True)

print("\n\n\n   Top 20 nodes by betweenness centrality:\n")
for b in sorted_betweenness[:20]:
    print(b)

file_centrality_info_open = open(save_file_location +  \
output_centrality_csv_file_name, "w")
file_centrality_info_open.write(str("Node," + "Centrality,"  \
+ "Degree" + "\n"))
file_additional_info_open.write("\n" + str("Node, " +  \
"Centrality, " +  "Degree" + "\n"))

#First get the top 20 nodes by betweenness as a list
top_betweenness = sorted_betweenness[:20]

print("\n\nTOP 20 BETWEENNESS CENTRALITY: \n")

#Then find and print their degree
for tb in top_betweenness: # Loop through top_betweenness
    degree = degree_dict[tb[0]] # Use degree_dict to access  \
    # a node's degree, see footnote 2
    print("Name:", tb[0], "| Betweenness Centrality:", tb[1],  \
    "| Degree:", degree)
    file_additional_info_open.write(str(tb[0]) + ", " + str(tb[1]) \
    +  ", " + str(degree) + "\n")
    file_centrality_info_open.write(str(tb[0]) + "," + str(tb[1])  \
    +  "," + str(degree) + "\n")

##### GIVEN SOURCE CODE (MODIFIED) ENDS ########
file_additional_info_open.close()
file_centrality_info_open.close()

############ SAVE DATA GRAPH:
print("\n\n SAVING DATA\n\n")
nx.write_gexf(G, save_file_location + output_graph_gexf_file_name)

################################################
```

```
def get_file_name_without_all_nodes_prefix_suffix(file_name,  \
        all_nodes_file_prefix, all_nodes_file_suffix):
    if file_name.startswith(all_nodes_file_prefix):
        file_name = file_name.replace(all_nodes_file_prefix, '')
        file_name = file_name.replace(all_nodes_file_suffix, '')
    else:
        raise Exception("FILE name doesn't start with prefix suffix  \
        for all_nodes")
    return file_name

def get_file_name_with_source_target_prefix_suffix(file_name,  \
    all_nodes_file_prefix, \
    all_nodes_file_suffix, source_target_file_prefix,  \
    source_target_file_suffix): # output --> CWE_...._bad

    if file_name.startswith(all_nodes_file_prefix):
        file_name = file_name.strip(all_nodes_file_prefix)
        file_name = file_name.strip(all_nodes_file_suffix)
        file_name = str(source_target_file_prefix) + file_name + \
        str(source_target_file_suffix)
    return file_name

def get_all_nodes_source_target_file_names_with_path( \
    directory_to_process, all_nodes_file_prefix,  \
    all_nodes_file_suffix, source_target_file_prefix,  \
    source_target_file_suffix):
    list_dir = os.listdir(directory_to_process)
    all_nodes_source_target_dirt = dict()

    for file_name in list_dir:
        if file_name.startswith(all_nodes_file_prefix) and  \
        file_name.endswith(all_nodes_file_suffix):

            get_source_target_file_name =  \
            get_file_name_with_source_target_prefix_suffix( \
            file_name, all_nodes_file_prefix, all_nodes_file_suffix,  \
            source_target_file_prefix, source_target_file_suffix)

            if os.path.exists(directory_to_process + "/" + \
            get_source_target_file_name):
                this_all_nodes_file_name_with_path = str( \
                directory_to_process + \
                "/" + file_name)
                this_source_target_nodes_file_name_with_path = str( \
                directory_to_process + "/" +  \
                get_source_target_file_name)

                all_nodes_source_target_dirt[ \
                this_all_nodes_file_name_with_path] =  \
                this_source_target_nodes_file_name_with_path

            else:
                raise Exception("COULDN'T FIND CORRESPONDING SOURCE  \
                TARGET FILE : " + \
                str(get_source_target_file_name))
    return all_nodes_source_target_dirt
def get_file_run_command(directory_to_process):
    get_all_nodes_source_target_nodes_files_dict =  \
```

```
    get_all_nodes_source_target_file_names_with_path( \
    directory_to_process,  \
    all_nodes_file_prefix=ALL_NODES_FILE_PREFIX,  \
    all_nodes_file_suffix=ALL_NODES_FILE_SUFFIX,  \
    source_target_file_prefix=SOURCE_TARGET_NODES_FILE_PREFIX, \
    source_target_file_suffix=SOURCE_TARGET_NODES_FILE_SUFFIX)

    print(get_all_nodes_source_target_nodes_files_dict)

    for this_all_nodes_file_with_path in  \
    get_all_nodes_source_target_nodes_files_dict:
        this_source_target_file_name_with_path =  \
        get_all_nodes_source_target_nodes_files_dict[ \
        this_all_nodes_file_with_path]
        print("\n\n\n   PROCESSING FILE : " +  \
        this_all_nodes_file_with_path)
        find_centrality(this_all_nodes_file_with_path,  \
        this_source_target_file_name_with_path)

##########

print("\n\n    RUN \n")
get_file_run_command(directory_to_process)
```

```
# End of file
```