8-2023

Characterizing and Detecting Software Attack Surface Components

Sara Moshtari
sm2481@rit.edu

Recommended Citation

Moshtari, Sara, "Characterizing and Detecting Software Attack Surface Components" (2023). Thesis. Rochester Institute of Technology. Accessed from

Characterizing and Detecting Software Attack Surface Components

by

Sara Moshtari

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Computing and Information Sciences

B. Thomas Golisano College of Computing and
Information Sciences

Rochester Institute of Technology
Rochester, New York
[August 2023]

Characterizing and Detecting Software Attack Surface Components

by

Sara Moshtari

Committee Approval:

We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computing and Information Sciences.

Dr. Mehdi Mirakhorli

[Advisor's name] Date
Dissertation Advisor

Dr. Christian Newman

[Committee member's name] Date
Dissertation Committee Member

Dr. Mohamed Wiem Mkaouer

[Committee member's name] Date
Dissertation Committee Member

Dr. Zhe Yu

[Committee member's name] Date
Dissertation Committee Member

Dr. Dan Phillips

[External Chair's name] Date
Dissertation Defense Chairperson

Certified by:

[Ph.D. Program Director name] Date
Ph.D. Program Director, Computing and Information Sciences

Characterizing and Detecting Software Attack Surface Components

by
Sara Moshtari

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences Ph.D. Program in
Computing and Information Sciences
in partial fulfillment of the requirements for the
Doctor of Philosophy Degree
at the Rochester Institute of Technology

Abstract

The notion of *Attack Surface* refers to the critical points on the boundary of a software system which are accessible from outside or contain valuable content for attackers. The ability to identify attack surface components of software system has a significant role in effectiveness of different security analysis approaches such as vulnerability analysis. Most prior works focus on the security analysis approach and use an approximation of attack surfaces. There have not been many attempts to create a comprehensive list of attack surface components. Although limited number of studies have focused on attack surface analysis, they defined attack surface components based on project specific *hypotheses* to evaluate security risk of specific types of software applications. This thesis provides a comprehensive attack surface model and proposes novel approaches for automating detection of attack surface components in source code. By leveraging a qualitative analysis approach, we empirically identify an extensive list of attack surface components. To this end, we conduct a Grounded Theory (GT) analysis on 1444 previously published vulnerability reports and weaknesses. We extract vulnerability information from two publicly available repositories: 1) *Common Vulnerabilities and Exposures (CVE)* and 2) *Common Weakness Enumeration (CWE)*. We ask three key questions: *where* the attacks come from, *what* they target, and *how* they emerge. To answer these questions three core categories for attack surface components are defined: *Entry points*, *Targets*, and *Mechanisms*. We extract attack surface concepts related to each category from collected vulnerability information using the GT analysis and provide a comprehensive categorization that represents attack surface components of software systems from various perspectives. This research introduces 254 new attack surface components that did not exist in the literature. In this study, we propose two new generic approaches based on Language Models (LM) that can be used to detect different types of attack surface components. 1) A probability-based classification approach using novel term weighting technique; 2) A novel *Natural Language Inference (NLI)* model based on pre-trained CodeBERT model. We evaluate the approaches for identifying nine different types of attack surface components using a java dataset collected from GitHub. The experimental results show that the term weighting approach can detect attack surface components with Fscore higher than 80%. The proposed CodeBERT NLI approach can detect the attack surface components with Fscore higher than 92% and for some attack surface components the Fscore is 100%. We also evaluate ChatGPT performance in identifying the attack surface components. ChatGPT responses show that its capability in identifying attack surface components is different. It can detect different attack surface components with Fscore between 30%- 90%. Finally, we compare three approaches and show that the proposed CodeBERT NLI has the best performance in comparison to the term weighting approach and ChatGPT.

Acknowledgments

I am deeply grateful to my advisor Dr. Mehdi Mirakhorli for his constant support. Dr. Mirakhorli provided me useful feedback on my research and helped me to navigate through the rough paths.

I also would like to thanks the committee members of this thesis for their guidance: Dr. Mohamed Wiem Mkaouer, Dr. Christian Newman, Dr. Zhe Yu, and Dr. Dan Phillips. Moreover, I would like to thank my collaborator Dr. Ahmet Okutan, as well as my labmates and colleagues.

I am thankful for my family, relatives, and friends that help me go through this journey.

This dissertation is dedicated to my family. Thank you for your love, support, and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Attack Surface Analysis

Nowadays, many cyber attacks are rooted in software vulnerabilities and software security is more increasingly becoming a day-to-day concern for organizations. A wide range of security analysis techniques have been used by software security practitioners to improve the confidentiality, integrity, and availability of software systems. These techniques often directly or indirectly rely on understanding a set of critical points on the boundary of a software system, where an attacker can try to enter, cause an effect on, or extract data from [42, 60] them which are called application's *attack surfaces.*

Attack surface analysis—the process of identifying applications' attack surface components (a.k.a points) plays a key role in numerous security analysis techniques such as risk analysis [9, 36, 42, 60, 61, 62], vulnerability detection [13, 45, 52, 54, 73, 84, 98, 99, 105], and software testing [8]. Although prior software vulnerability detection and testing approaches consider parts of code such as *Sink*, *Source*, *Entry point*, and *API/Function calls* as attack surface, but these studies primarily focus on the analysis itself, rather than identifying attack surface components. There are a few studies that elaborate on the notion of the attack surface [36, 42, 43, 60, 61, 62, 73]. These studies consider entry points, exit points, channels, etc. as attack surface components, and test and validate them as existing theories [42,43,61,62,73]; However, they focus on limited-scope and describe attack surfaces of a specific system such as operating systems [42,43,60,62] and web applications [7]. These studies show that applications with smaller attack surfaces are less vulnerable. While there has been a significant interest by practitioners and in the literature to study attack surface components of a given system, unfortunately, we lack a generic comprehensive guidance to support security research engineers in identifying attack surfaces of a given system. To the best of our knowledge, there is no prior research that takes a comprehensive approach to characterize and identify attack surface components in software systems.

In this thesis, we leverage a Grounded Theory (GT)-based approach [29, 31, 95] to characterize software attack surfaces and develop an extensive list of attack surface components that can be used by researchers and practitioners. Grounded theory is a social science research method that extracts theories from unstructured data and leads to discoveries directly supported by empirical evidence [28,30,31,82,95]. In this research, 810 vulnerability reports from Common Vulnerabilities and Exposures (CVE) data published by MITRE Corporation [69] are extracted and analyzed. In addition, 634 entries in Common Weakness Enumeration (CWE) data [70], which is an extensive catalog of different types of software and hardware weaknesses, are analyzed. By leveraging the Grounded Theory and reviewing vulnerability reports and weaknesses, high-level concepts which are related to software systems' attack surface are defined. We use Straussian GT [18, 96] as a systematic inductive method for conducting qualitative research of identifying attack surface components.

We consider three core theories, which are *Entry Points*, *Targets*, and *Mechanisms*. We extract the concepts related to each theory from collected vulnerability data and define a generic taxonomy for each core theory. The defined concepts related to each theory are categorized in four major groups: software source code (*Code*), its executable (*Program*), the *System*, and the *Network* environment. Then, we identify attack surface components under each category and compare the proposed attack surface categorization model with the literature. The comparison results indicate that the proposed attack surface model covers almost all attack surface components defined in the literature, while prior works cover only a small portion of the concepts identified by our analysis. The result of quantitative comparison shows that on average, only 6.7% of the studied *Code* level attack surface components are covered by previous works. In the best case only 50% and 20% of the *Network* and *Program* level mechanisms and 20% of the *Network* level entry points identified by this paper are covered in the literature.

The other aspect of this thesis focuses on automating the identification of attack surface components in source code. To reach this goal we propose two generic automatic approach to identify attack surface components using Language Models (LM). First, we propose a novel approach based on *Natural Language Processing (NLP)* term frequency technique to classify methods that contain attack surface components. It identifies indicator terms for each attack surface component and then uses a probability-based classification approach to identify parts of the code that contain attack surface components; Second, we propose a novel model based on *Natural Language Inference (NLI)* and pre-trained CodeBERT model for detecting attack surface components in source code. To train and evaluate the approaches, we collect data for nine attack surface components defined in the first step of the study which are: *Input_Stream_File*, *Execute_OS_Command*, *Execute_SQL_Command*, *User_Input*, *Serialization_Deserialization*, *Read_Socket*, *OS_Signal_Handler*, *Weak_Encryption*, and *Reflection*. Two types of data was collected: 1) Standard Java APIs, and 2) Methods collected from GitHub repository. We totally collected 38,568 java APIs and 524 java method samples. We manually reviewed data for labeling them. The experimental results show that in the best threshold the term weighting approach can detect the attack surface components with average Fscore of 90%. The average Fscore in CodeBERT NLI model is 96%. We also evaluate the performance of the ChatGPT Large Language Model(LLM) in identifying attack surface components. The results show that ChatGPT can detect some attack surface components with Fscore of 90% but it does not perform well in identifying some other attack surface components – the worst Fscore was 30% for *Execute_OS_Command* and *Reflection* components.

1.2 Research Goals

The research goals of this dissertation work are:

Goal 1 Characterizing Software Attack Surface Components As a first step towards the goal of identifying software attack surface components, first need to characterize attack surface components and provide an extensive list of attack surface components. To achieve this goal, this Ph.D. work leverages a qualitative approach, which is a social science research method, and reported vulnerabilities available in public repositories. The section 4.1 from chapter 4 discusses the steps performed to identify and define concepts related to attack surface from reported vulnerabilities. Research Questions:

- RQ1: Where are the critical entry points in a software system that are used by attackers to get in?
- RQ2: What assets or components in a software system are targeted by attackers?
- RQ3: How do attack surfaces emerge, and what types of mechanisms are utilized to reach the targets?

Goal 2 Detecting Software Attack Surface Components in Source Code The second goal of this research work is to propose an automatic approach to identify attack surface components, which are defined in the first step, in source code. This step of the research involves collecting data, proposing, and evaluating different attack surface detection models. We propose two approaches: 1) Novel term weighting and probability-based classification approach; and 2) NLI model. We also evaluate the performance of ChatGPT model in identifying attack surface components. The section 4.2 in chapter 4 discusses how the detection models work.

Research Questions:

- RQ4: Can a term weighting and probability-based approach detect software attack surface components?
- RQ5: How precisely the probability_based approach detects different attack surface components?
- RQ6: Can a NLI_based approach detect attack surface components?
- RQ7: How effective is the NLI_based approach in detecting different attack surface components?
- RQ8: Can ChatGPT model detect attack surface components?
- RQ9: How precisely the ChatGPT can detect different attack surface components?
- RQ10: Which language model (term weighting, NLI, and ChatGPT) performs better in detecting different attack surface components?

1.3 Contributions

- Proposing a comprehensive attack surface model by leveraging a qualitative approach. The proposed model can be used by researchers and security practitioners in different software security analysis approaches.
- Introducing 254 new attack surface concepts that have not been studied before.
- Proposing a comprehensive code level attack surface components. A small portion of the code-level attack surface concepts (6.7%) defined in the model are introduced in the literature.

- Proposing a new approach by leveraging text processing and machine learning techniques for identifying software attack surface components in source code.
- Proposing a new CodeBERT NLI approach by leveraging pre-trained models for identifying software attack surface components in source code.
- Evaluating the performance of ChatGPT model in identifying software attack surface components in source code.

1.4 Outline

The remainder of this thesis is organized as follows: In the next chapter, software vulnerability databases, grounded theory, and NLP approach which are used in this thesis are reviewed. Chapter 3 reviews the literature. Chapter 4 presents how a qualitative analysis approach has been used to identify attack surface components from reported vulnerabilities and describe the automatic approaches proposed for detecting attack surface components in source code. Then, in chapter 5, current results are discussed. This includes data collection and experimental results that has been performed to evaluate the model. Finally, chapter 6 summarizes this thesis and a list of publications and proposed publications have been provided.

Chapter 2

Background

This chapter defines the main concepts used throughout this thesis to ensure that this work can be understood by a broader audience.

2.1 Software Vulnerability and Vulnerability Databases

Software vulnerabilities are defects that affect a system's intended security properties. These security weaknesses are typically disclosed and discussed across online forums, and many other websites, as well as tracked by vulnerability databases. A well-known vulnerability database is the *Common Vulnerabilities and Exposures (CVE)*. The Mitre corporation CVE is an open platform to list publicly disclosed vulnerabilities [69]. An instance of vulnerability recorded in NVD is shown in figure 2.1.

As demonstrated in this excerpt, each entry in NVD includes a short description of the vulnerability as well as a list of references that are links to other Web sites (such as issue tracking systems, advisories, exploit databases, etc.) that may contain more details about the CVE instance. NVD also indicates the software's releases affected by the vulnerability. In this example, multiple openwrt versions were affected. Some of the CVE instances may also include CWE IDs that indicate the vulnerability type. The CWE tag refers to an entry from the *Common Weakness Enumeration (CWE)* dictionary. CWE enumerates a list of common security weaknesses and categorizes them based on different views to help practitioners in securing their applications [70]. As shown in figure 2.2, it provides a concise description of the weakness, common consequences, likelihood of

CVE-2020-7248

Description: libubox in OpenWrt before 18.06.7 and 19.x before 19.07.1 has a **tagged binary data JSON serialization** vulnerability that may cause a **stack based buffer overflow**.

References:
https://github.com/openwrt/openwrt/commits/master
https://openwrt.org/advisory/2020-01-31-2
https://nvd.nist.gov/vuln/detail/CVE-2020-7248#range-4512438

Affected Software Configurations:
openwrt from 18.06.0 to 18.06.7, openwrt:19.07.0, openwrt:19.07.0:rc1, openwrt:19.07.0:rc2

Related CWE:
CWE-787: Out-of-bounds Write (source: NIST)

...

Figure 2.1: CVE-2020-7248: An instance of vulnerability recorded in NVD

CWE-787: Out-of-bounds Write

Description: The software writes data past the end, or before the beginning, of the intended buffer.
Extended Description: Typically, this can result in corruption of data, a crash, or code execution. The software may modify an index or perform pointer arithmetic that references a memory location that is outside of the boundaries of the buffer. A subsequent write operation then produces undefined or unexpected results.
Common Consequences: Modify Memory; DoS: Crash, Exit, or Restart; Execute Unauthorized Code or Commands
Likelihood Of Exploit: High
Applicable Platforms: C, C++

Example: In the following example, it is possible to request that **memcpy** move a much larger segment of memory than assumed:
(bad code)
Example Language: C

```
int returnChunkSize(void *) {
/* if chunk info is valid, return the size of usable memory,* else, return -1 to indicate an error*/
...}
int main() {...
        memcpy(destBuf, srcBuf, (returnChunkSize(destBuf)-1));
... } . . . . . . .
```

Figure 2.2: CWE-787: An instance of a weakness in CWE dictionary

exploitation, demonstrative examples, and reference to other resources.

2.2 Grounded Theory

Grounded theory (GT) is a method originally described by Glaser and Strauss [29] in their book titled ”The Discovery of Grounded Theory” [38]. The goal of GT is to generate theory from unstructured data. It is used to define new theories rather than test or validate existing theory. GT is a suitable approach for investigating questions such as what's going on here? [2].

GT has provided an extremely useful methodological approach in numerous areas such as medical sociology [11], nursing [5], education [80] and management [47].

2.2.1 Versions of Grounded Theory

Glaser's GT (classic or Glaserian GT)

Glaser's version of GT is called *Classic GT* [28]. In this version of the GT research questions are not defined at the beginning and they emerge from the research. In classic GT, literature review should be delayed until the end of the analysis. It helps that the concepts defined in the literature does not affect the emerging theories. This version includes three coding process:

- *Open Coding:* In open coding process all data are analyzed and define codes which are related to the problem statement. Code is a phrase that summarizes the key points in a descriptive way.
- *Selective Coding:* During selective coding core variables are defined based on the relationships between codes defined in the open coding process. These core variables guide further data collection.
- *Theoretical Coding:* It establishes the conceptual relationships between substantive codes and define the final theory.

This approach is guided by asking questions like " *What do we have here?*"

Strauss and Corbin's GT(Straussian GT):

In this version of the GT the research questions might be asked upfront [18]. The literature can be reviewed during the analysis process and concepts from the literature may be used. The Straussian GT includes three coding process:

- *Open Coding:* In open coding process the key points are extracted from data and the codes or are defined.

- *Axial Coding:* In axial coding coded data are defined in new ways by identifying their relationship. It is like theoretical coding in the Glasert's version.
- *Selective Coding:* During selective coding central categories are defined that all major categories can be linked to theses central categories

This approach is guided by asking questions like " *When,where, and under what conditions phenomena occur?*"

Charmaz's constructivist GT:

In this version of the GT analysis begins by asking initial research questions which evolve throught the research [12]. Charmaz considers the Glaser's approach for delaying literature review as impractical and mentions that a literature review should be tailored which fit the GT process. This version includes three coding process:

- *Initial Coding:* In open coding process, the data is examined and coding is performed.
- *Focused Coding:* In focused coding, important and frequent codes and categories are selected and categorization is performed based on them.
- *Theoretical Coding:* During theoretical coding the relationship between categories are identified and a cohesive theory is defined.

This approach is guided by asking questions like " *What do the data suggest?*"

2.3 Natural Language Processing

Natural Language Processing (NLP) is the sub-filed of Artificial Intelligence (AI). NLP is used to make machine capable of identifying, processing, and understanding information in human language. NLP has different application areas such as document summarization, sentiment analysis, automatic speech recognition, machine translation, etc. It also has been used in software security analysis such as vulnerability detection [104, 119]. In NLP approaches, first, a set of text preprocessing steps are performed on text data then different embedding approaches are used to convert the text into machine readable vectors. Text preprocessing includes following steps:

- **Tokenization:** The first step in text processing is word identification. In tokenization phase the text paragraphs are splitted into their word components.
- **CamelCase Split:** In this step, camelcase type strings are converted into a slice of words. For instance, *removePropertyChangeListener* which is a method name is converted into *"remove Property Change Listener"*.
- **Stop Word Removal:** A stop word is a commonly used word in languages such as 'the', 'in', 'a', etc. In this step, the stop words are removed because they does not provide meaningful insights in text identification.
- **Lemmatization:** The text may contain different grammatical forms of a word. For instance, the text may contain *speak*, *speaking*, *spoke*. Lemmatization technique reduces these forms to a common base form which is called *Lemma*.

Once the required data is available in the right format, and level of quality, we start building the Language model. The Language model understands and processes the textual data. Word Embedding or Language Models are the algorithms which are able to convert texts into vectors. In text processing, term weights are mainly used to represent the usefulness of a term. For instance, *Term Frequency (TF)* is a weighting approach which is used to measure that how many times a term is present in a document.

Chapter 3

Related Work

3.1 Vulnerability Detection

Most software security analysis approaches are focused on vulnerability detection. Vulnerability discovery approaches can be categorized into two main categories:

3.1.1 Program Analysis Approaches

Static Analysis

Static analysis techniques have been used to detect software defects and vulnerabilities. They provide an Intermediate representation of programs such as Abstract Syntax Tree (AST), Control Flow Graph (CFG) and Program Dependency Graph (PDG). The perform different data-flow, control-flow analysis [118] and also rule matching [14] to detect defects or vulnerabilities. Static analysis approaches detect vulnerabilities based on patterns of known vulnerabilities. Therefore, they are not able to detect unseen vulnerabilities. Since they do through program path analysis, they have complexity for large code bases and converting a program to symbolic formula is computationally expensive. They also result in a large amount of false positives [58].

Dynamic Analysis

Dynamic analysis approaches detect software vulnerabilities by providing different inputs and monitoring the run time behaviour of the program [48,75,97]. Fuzzing [97] is a dynamic analysis approach that generate inputs to trigger security flaws. Dynamic analysis approaches cover some part of the code and requires considerable effort to generate inputs.

Hybrid Approaches

Many studies have combined the properties of static and dynamic analysis to provide a more complete solution [85, 106]. However, all these approaches are difficult to scale up to large applications and have path explosion problem.

3.1.2 Machine learning-based approaches

Pattern-based Approaches

These approaches provide an abstract representation of source code by tokens [48], tree or graph structures [49]. They provide the same representation for vulnerable code fragments to characterize vulnerabilities. They use code similarity algorithms [51] to identify vulnerabilities in programs. Then, they use vector comparison [110] or approximate matching (ex. sub-graph matching [49]) to detect vulnerable location of the code. These approaches uses program syntax, semantic or both to create abstract representation. These approaches need different aspects of software information such as syntactic, control-flow and data-flow information that makes them complex.

Deep Learning-based Approaches

Limited number of studies proposed deep learning based models to detect rich feature from source code automatically. This will relieve human expert from defining and selecting features. These approaches provide an intermediate representation of source code and then map it to a vector that is used as input for deep learning model. Vuldeepecker [55] is a deep-learning based vulnerability detection system that use code gadjets as intermediate representation. It extracts program slices which are related to forward and backward API calls using data flow analysis. It uses the semantic of the program to create intermediate representation. Sysevr [53] use the syntactic and semantic of the source code to create intermediate representation. The intermediate representation is created based on program dependence graph. It considers the vulnerability related syntactic and semantic information but it is dependent on another pattern based tool to extract vulnerability patterns. Therefore, it has the limitations of pattern based approaches and it also has challenges of program dependence graph analysis. Lin et al. [56] considered Abstract Syntax Tree as an intermediate representation to provide input vector for deep learning models. They provide deep features at file level and did not consider any dependency between files. Besides that, they did not consider any vulnerability information to provide the representation.

Metric-based Approaches

Software metric is a measure of some property of a piece of software. Software metrics such as complexity, code churn and developer activity metrics have been widely used for software fault prediction [3, 10, 25, 32, 83, 111, 117]. In recent years, they have been used to predict vulnerability-prone components [17, 21, 22, 44, 76, 88, 91, 92, 94, 101]. All metric-based approaches except [22] used supervised learning models. However, training data that are provided for vulnerability prediction are highly imbalanced because of rare number of discovered vulnerabilities. Therefore, supervised models can not perform well. Besides that, Jimenez et al. [44] showed that training data that is provided for vulnerability prediction are based on unrealistic labelling and using realistic labelling provides poor performance. All these studies except [22] used different sets of regular metrics that have been used for fault prediction. However, they could not provide acceptable accuracy for vulnerability prediction. Patrick et al. [71] showed that these metric-based approaches can not perform well enough to provide actionable results.

Text Mining-based Approaches

Zhang et al. [119] and Walden et al. [104] utilized text mining to predict vulnerable location of the code. They considered source code as text and tokenized the source code. Frequency of Items considered as metrics for vulnerability prediction. This approaches also can not perform well from the perspective of accuracy and takes a lot of time to calculate the frequency of tokens.

Supervised Prediction Models

Most initial studies on software defect prediction used multiple releases of a project. They built prediction models on a project and evaluated the model on the same project. Basili et al. [4] assessed the usefulness of CK metrics [16] for predicting fault-prone classes in a management information system. Using logistic regression, these object-oriented (OO) metrics could provide high performance in fault prediction. Briand et al. [7] also evaluated the effectiveness of CK metrics and other OO design metrics to predict faulty components in an industrial project. To evaluate the prediction model they performed 10-cross validation. Gyimothy et al. [33] evaluated the applicability of the CK metrics to predict the number of bugs in classes. Their experiment on seven versions of Mozilla showed that CBO was the best metric in predicting the fault-proneness of classes. Nagappan et al. used code churn, LOC, and code complexity metrics from Windows XP to estimate the post-release failure-proneness of Windows Server 2003 [74]. Ostrand et al. [82] used code metrics such as LOC and file change history to predict the number of faults in a multiple release software system. They used a negative binomial regression model for fault prediction. Denaro et al. [19] used regression models and data from the Apache 1.3 project to predict defects on the Apache 2.0 project. These studies were conducted in the context of project defect prediction. However, these approaches are unsuitable for projects that do not have historical data available to be used to train the prediction models. Therefore, some researchers proposed cross-project defect prediction models.

In cross-project defect prediction studies, prediction models were created based on one or more projects and the models were evaluated on other projects. Zimmerman et al. [121] found that among 622 cross-project experiments only 3.4% worked. Turhan et al. [103] showed that cross-company defect predictors could not outperform within-company defect predictors. Rahman et al. [88] evaluated cross-project predictors based on cost-sensitive measures rather than usual classification measures. They showed that inspection of a smaller fraction of the code – in cross-project defect prediction- is as good as within-project defect prediction and better than a random model. Canfora et al. [9] proposed multi-objective cross-project predictor based on the Rahman et al.'s study. They found that multi-objective predictors perform better than single-objective in cross-project defect prediction. He et al. [34] and Herbold [35] used distributional characteristics of datasets to select suitable training data for projects without historical data. Singh and Verma [93] showed that cross-project predictors that were built using design metrics are good predictors for software faulty modules. Panichella et al. [86] proposed a combined approach based on different classifiers that improve the performance of cross-project predictors. Xia et al. [109], Ryu et al. [90] and Li et al. [50] proposed methods to improve the performance of cross-project defect predictors. Kamei et al. [46] proposed cross-project predictors that identify source code changes that have a high risk of introducing a defect. The review reveals that cross-project defect prediction studies have challenges of selecting suitable training data because of heterogeneity [41] and used different methods to improve the performance of these kinds of predictors.

Unsupervised Prediction Models

Unsupervised approaches try to detect defective components from unlabeled data. They used clustering algorithms to capture software defect clusters. Zhong et al. [120] used k-means and Neural-Gas clustering algorithms to cluster software modules into a small number of coherent group. Then, the software engineering expert inspected different clusters to label them as either fault-prone or not fault-prone. Their results showed that this unsupervised method achieves comparable classification accuracies with other classifiers. Bishnu and Bhattacherjee [6] used k-means algorithm for fault prediction when the fault data for modules are not available. They used Quad Tree-based method and the concept of clustering gain to assign the appropriate initial cluster centers. They showed that the performance of the Quad-tree based fault predictor is comparable with the supervised learning approaches. Park and Hong [87] built unsupervised models for fault prediction using clustering algorithm. They tried to solve the issue of clustering algorithms that is the number of clusters, by using Expectation–Maximization (EM) and Xmeans, which determine the number of clusters automatically. Zhang et al. [114] proposed connectivity-based unsupervised classifier using spectral clustering. They considered the connectivity among software entities based on similarity between metric values. They showed that this unsupervised approach achieves impressive performance in a cross-project defect prediction. In these studies, labeling the clusters as defect-prone or non-defect-prone is a challenging problem. Zhang et al. [115] tried to solve the labeling issue by using average metrics value in each cluster. They considered the cluster with higher average value as defective. Fu and Menzies [26] and Yang et al. [113] used unsupervised approaches, which are not based on clustering algorithms, for change-level defect prediction. Yan et al. [112] used the same approach for file-level defect prediction. Despite the importance and ease of use of unsupervised approaches, limited studies have been conducted in this area.

3.2 Attack Surface Analysis

Measurement of the *attack surface* is one of the elements in the design phase of Trustworthy Computing Security Development Lifecycle (SDL) introduced at Microsoft in 2004 [57]. The phrase software attack surface having been introduced only a year earlier by Howard [42]. Software security researchers used Howard's concept of the attack surface to measure the overall security of a system. After that, a variety of definitions are provided for the phrase attack surface:

- "...union of code, interfaces, services, protocols, and practices available to all users, with a strong focus on what is accessible to unauthenticated users." [43]
- "...the system's actions that are externally visible to its users and the system's resources that each action increases or modifies." [60]
- "...a list of attack features: Open sockets, Open RPC endpoints, Open named pipes, Services, etc." [42]

This shows that there is not a generic definition for attack surface and researchers have different views on what the phrase attack surface means. Besides that, the phrase has been used at different granularity levels: [100]

- *Function*: The phrase attack surface has been used with methods, functions or individual lines of code. For example, the attack surface defined as certain set of functions to be accessible through the Application Program Interface (API).
- *File:* The phrase attack surface applied to source code files. For example, having source code in certain files vulnerable to particular type of attack.
- *Binary:* The phrase attack surface with source code packages such as binaries, packages, modules or components.
- *System:* The phrase attack surface is used when reasoning about entire systems. For example, enabling certain features in an operating system such as Windows and Linux.
- *Computer Network:* The phrase attack surface is applied to entire networks. For example, evaluating the notion of isolating certain set of sensitive hosts to a sub-network.
- *Theoretical:* The phrase attack surface is used in a theoretical capacity. For example, quantifying the attack surface of an entity based on theoretical notions.

Limited studies have been proposed for identifying attack surface of a software system. Some of these studies tried to approximate the attack surface using stack traces [98, 99]. Theisen et al. [98] have approximated the attack surface and run vulnerability prediction models on the attack surface to improve the performance of vulnerability prediction models. They considered information in stack traces of windows crashes as approximation of attack surface. They mentioned that stack traces can be approximation of attack surface because they contain both direct and indirect entry point and control and data flow graph. They extracted binaries and source files from stack traces and replicated Zimmerman et al. metric-based vulnerability prediction models(VPM) [122] on these parts of the code. They could improve the performance of the VPM. Recall improved from 0.07 to 0.1 for binaries and from 0.02 to 0.05 for source files. Precision remained at 0.5 for binaries, while improving from 0.5 to 0.69 for source files. In this study, they used stack traces for approximating attack surface which is not available for all software systems. Howard et al [43] described attack surface in three abstract dimensions which are *targets and enablers*, *channels and protocols* and *access rights*. They defined *targets and enablers* as resources that attacker can use such as process and data. For *channels* they considered two types of channels: message passing and shared memory. They also considered account, privilege level and trust-relationship as *access rights*. They added three attack vectors to the 17 attack vectors that Howard [42] identified for windows system. They also proposed state machines to model the system and threats. Theses attack surfaces were used to define a metric to compare two versions of a system. This study is system level because they considered the services that run on a system. Manadhata and wing [4, 63] proposed an attack surface metric to compare the security of two versions of a system. They proposed a model of system and its environment using state machine and considered any component that can be used to send/receive data to/from environment as an attack surface for Linux systems. They considered *methods* of the system, *channels* and *data items* as resources. They used the model to identify the accessible subset of resources (in terms of access rights) that contribute to the system's attack surface. In this study, the attackability of a resource is defined based on its potential damage and the effort required to acquire an access right level. Therefore, they defined different attack classes based on the attackability level and defined that attack surface of the system in three dimensiosn: data, channel, and system attackability. They validated the attack surface components by performing regression test on Microsoft Security Bulletin (MSB) (to measure correlation between attack surface components and severity of vulnerabilities) and also performing expert user surveys. They also evaluated the attack surface metric for Firefox and Proftp patches and showed that for each project, in order, 67 and 70 percent of patches reduced the attack surface. Huemann et al. [36] defined the components of the attack surface for web applications. They proposed attack surface of web application as a vector that has 22 dimensions. The dimensions are categorized in 7 groups and a weight is considered for each components. They proposed Euclidean norm of the vector as attack surface indicator. They evaluated the proposed metric on six applications. The results showed that the indicator values are close for the sites that use similar technologies. For detailed analysis they calculated intermediate square sums for the seven parameter groups to see which factors are more effective in attack surface of each web site. Nuthan and Meneely [73] proposed function and file level attack surface metrics. They considered entry and exit points and also dangerous system calls as attack surface components. They provided static and static+dynamic call graphs and calculated the proximity and risky walk metrics based on the call graph. They proposed three proximity-based metrics which are proximity to entry points, exist points and dangerous points. They also computed Risky walk of a function/file is the PageRank of that node in the call graph that simulate the behaviour of the attacker. They evaluated their metrics for Wireshark and FFmpeg in two ways: First, they performed association between metrics and post-release vulnerabilities in NVD; Second, they used the metrics to predict vulnerabilities at file/function level. The experimental results showed statistically significant association with historical vulnerabilities and showed that prediction models which are based on these attack surface metrics outperformed prediction models which are based on SLOC and coupling metrics. These studies proposed attack surface metrics and then evaluate their proposed metrics on vulnerability resources of one or two projects.

Theisen et al. [100] performed a survey on attack surface. They categorized the papers based on define, supported use and unsupported use based on the definition of attack surface. They categorized the attack surface into 6 categories which are methods, adversaries, flows, features, barriers and reachable vulnerabilities. They mentioned that researchers should focus on one of these categories based on their area of the research.

3.3 Grounded Theory in Software Engineering

Qualitative research methods have been increasingly employed in Software Engineering (SE) research [20, 23]. *Grounded Theory* is one of the qualitative methods that is attracting particular attention [2,40]. Some studies evaluated the challenges of using GT in SE and proposed approaches to solve these challenges. Stol et al. [96] surveyed 100 articles in nine prominent SE journals that used GT. They provided a comparison between three variant of the GT. Their evaluation results showed that many SE studies do not generate a theory and do not clearly indicate which variant of the GT is used. In this study, they enumerated substantial challenges related to applying GT research in software engineering, including the proliferation of heterogeneous unstructured, semi-structured and structured data. Hoda [37] proposed a method called Socio-Technical Grounded Theory (STGT) to overcome the challenges of using GT in SE studies such as not acknowledging the version of GT being applied, combining guidelines without rationales, etc. This study also provided guidelines to provide ease of use of the method and improve the quality of outcome.

Some other studies used GT in different areas of SE. Hoda and Noble [38] studied a theory of software development teams transitioning to agile development. They evaluated how development teams transition to agile practices by evaluation data of 18 development teams. Using GT they found five dimensions for agile transition: software development practices transitions from traditional to agile practices; team practices changes from manager-driven to team-driven; the management approach changes from driving to empowering; the reflective practices changes from being limited to becoming embedded as a means of guiding continuous improvement; and culture changes from hierarchical to open. Other study [107] used GT to study how much up-front architecture design effort is enough in agile software development to decrease risk and chance of failure. Danilova et al. evaluated the security warnings which shows security issues. They used GT to find developers' wishes concerning how and when they would like to receive security warnings. They conducted the study with 33 participants from different fields of software development. They defined the theory that context is the key to designing and presenting security warnings to developers. They also found that developers' preferences are based on the coding purpose, their characteristics, team, and organization context.

Chapter 4

Methodology

4.1 Attack Surface Analysis

Attack surface refers to the amount of code, functionality, and interfaces of a system exposed to attackers [43]. In this thesis, we rely on publicly available vulnerability repositories to identify common attack surface components in software systems. Vulnerability databases describe vulnerabilities using natural language and contain unstructured data. In order to identify attack surface components, we use an approach based on the **Grounded Theory (GT)** [29, 31, 95].

The Straussian GT is preferred over the Classic Glaserian GT approach [28, 30], because the study is led by research questions and existing concepts in the literature are used during the analysis [18, 89, 96]. The Straussian grounded theory encompasses the following activities: (1) defining research questions, (2) theoretical sampling, (3) open coding, (4) constant comparisons, (5) memoing, (6) axial coding, and (7) selective coding. Figure 4.1 shows how the Straussian GT was applied. Over

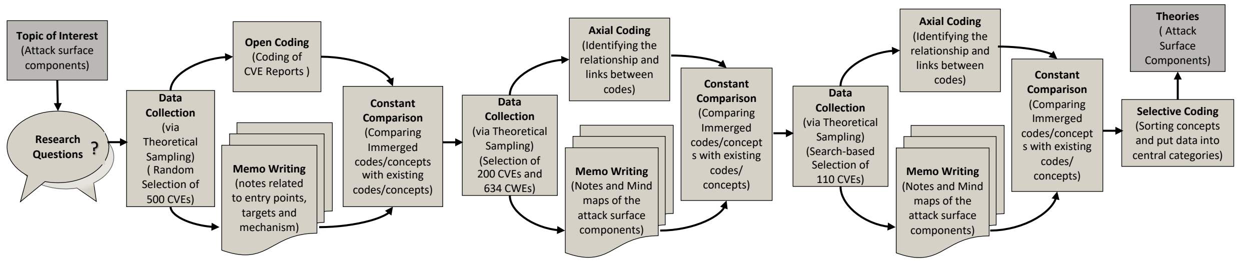


Figure 4.1: The Grounded Theory approach applied to my work

one year, the authors met weekly [72] to discuss, merge and finalize the codes, concepts, and categories. All collected data, derived intermediate data that contains codes and concepts, and the final attack surface categories are shared with the research community through a public GitHub repository [1].

4.1.1 Research Questions

In Straussian grounded theory approach [18], research questions may defined upfront. In this study, we consider broad and open-ended research questions for detecting attack surface components by inspiring from the apparent attack surfaces of a house. For example, in a house, front and back doors, windows, garage door, climbable trees or tables can be entry points and the attacker would consider precious items in the house, such as safe box, as target. There might be some mechanisms in building a house such as emergency stairs that could make the house more vulnerable. By considering similarities between a software system and a house from the perspective of a cyberattacker, the core concepts which help in defining attack surface components are identified. We focus on three research questions during the GT process and try to do coding in a way that can find theories from data to answer these questions:

- RQ1: Where are the critical *entry points* in a software system that are used by attackers to get in?
- RQ2: What assets or components in a software system are *targeted* by attackers?
- RQ3: How do attack surfaces emerge, and what types of *mechanisms* are utilized to reach the targets?

4.1.2 Data Collection

Given the topic of interest of this study, we need access to software vulnerability reports, the description of these vulnerabilities, in-depth analysis of how they occurred, as well as vulnerable code snippets and their patches. Thus, publicly available vulnerability repositories are targeted that contain different types of vulnerabilities and vulnerable code snippets.

Theoretical Sampling

It is a data collection process that data is gathered in different stages based on the concepts derived in each stage [12, 18, 96]. In theoretical data sampling, unlike conventional approaches, all data are not collected at the beginning. Data collection and analysis is a circular process. Concepts that are identified in each cycle guide the data collection process until the saturation occurs [96].

Data Sources

Vulnerability information is obtained from two publicly available vulnerability repositories: CWE and CVE. CVE list and additional information provided in the National Vulnerability Database (NVD) [77] are used in this study to extract vulnerability meta-data. Besides that, all information pieces provided in CWE dictionary were reviewed to extract attack surface components.

Issue tracking systems were used to obtain further discussions about CVEs, exploit information to identify how the vulnerability is being exploited, source code repositories to identify patches, and further resources to extract other related information if existed. Figure 4.2 shows the information model of the vulnerability data collected. The summary of the data sources used:

① **Retrieve vulnerabilities from MITRE and NVD**: We obtained vulnerability reports from MITRE CVE and NVD by consuming their public data feeds. Vulnerabilities disclosed in CVE are assigned a unique Identifier (CVE ID), a concise description, a list of affected software releases, and a list of references that can be used to obtain further details about the CVE, such as Issue Tracking Systems.

② **Obtain vulnerability details from issue tracking systems:** Although CVE reports provide information about different attributes of a vulnerability, they do not contain enough information to identify attack surface components at code level. Thus, we reviewed associated issue tracking systems for vulnerabilities that are related to open source projects. We leveraged the list of "references" to identify URLs to the corresponding bug entry of the issue tracking system and we read the developers' discussion about the problem, original code fragments, and their proposed solution(s).

③ **Gather patches from code repositories:** To retrieve patches that fixed vulnerabilities, we gathered the commits whose message explicitly mentioned the related bug id in the issue tracking systems or directly referred to the associated CVE. These patches often contained more

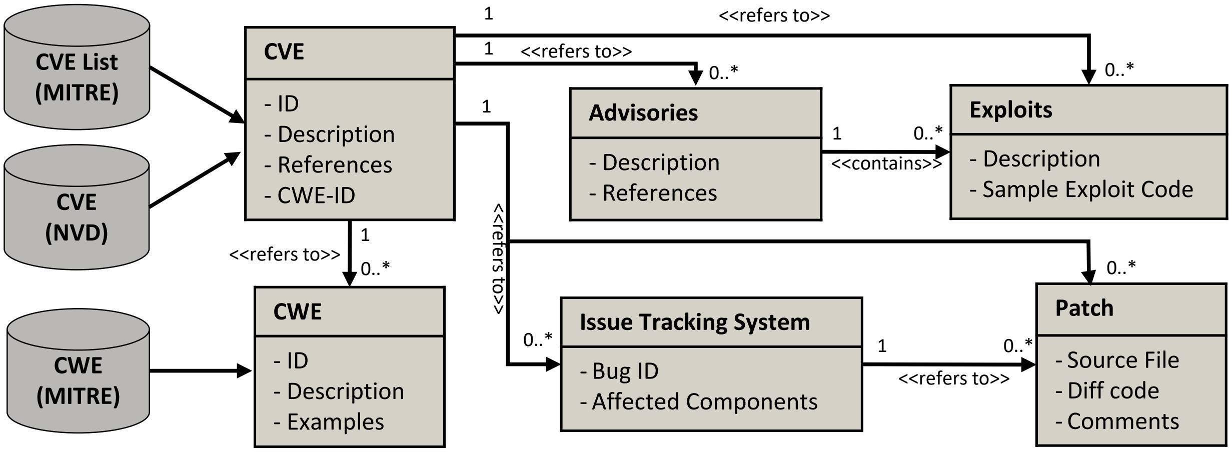


Figure 4.2: Information Model for the Collected Data

information about the vulnerability, and the files that were affected, *i.e.*, modified, added or removed during the fix. Identifying patches helped us to identify entry points, targets and mechanisms at the code level.

④ **Collect vulnerability details from other references:** In addition to information that are provided in CVE website, we analyzed all the URLs that are provided as references for each vulnerability. These references include links to vulnerability reports, advisories or exploit information. These references provide more information for attack surface analysis of the vulnerability.

⑤ **Get vulnerability details from related CWEs** We identified related CWEs for each vulnerability. The CWEs helped us to understand the security issue of the CVE and extract attack surface components.

Data Collection Process

Number of reported vulnerabilities in NVD has increased from 6500 in 2016 to above 18000 in 2020 [78]. We collected data form the vulnerabilities that have been reported from 2016 to 2020 to cover different types of vulnerabilities. For a more comprehensive attack surface analysis, we also collected data from *Introduced During Design* and *Introduced During Implementation* views [64,65] in *CWE* [70].

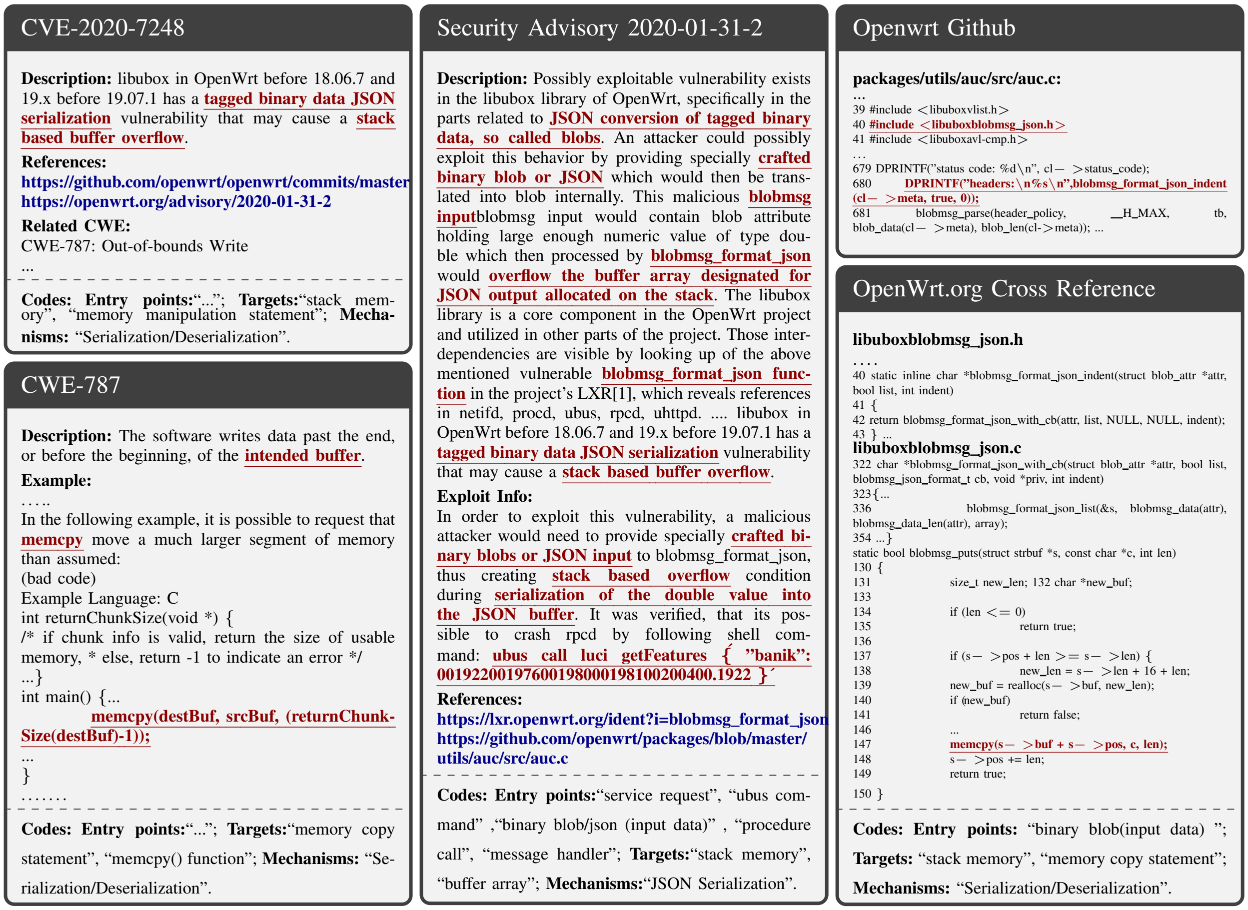
First stage: At the beginning, meta-data of 500 CVEs were collected by selecting 100 random CVE ID from each year. For some of these vulnerabilities only a CVE description was available (without any patch or advisory info), therefore, we were able to collect limited amount of information from the descriptions of such vulnerabilities. During the first stage of data collection, coding process was performed and initial concepts were extracted.

Second stage: During this stage, 200 CVEs were randomly selected from 2016-2020 but the areas that were theoretically saturated [12] during the first stage of the analysis were omitted. Therefore, we didn't do coding for the CVEs that were related to vulnerabilities such as *SQL injection*, *Buffer Overflow*, *Cross-site scripting*, and *Command injection* (36 CVEs), because reviewing more data related to these types of vulnerabilities no longer provided new theoretical insights about attack surface components at the end of the first stage. CVEs collected during the first stage, covered limited number of CWE branches (70 CWE IDs), therefore we also collected data from *Introduced During Design and Introduced During Implementation* views [64,65] in CWE (which totally contain 634 weaknesses after removing their common CWE IDs) to be more comprehensive. During our analysis in the second stage, we defined all components that can be part of an *Entry Point*, *Target* or *Mechanism.*

Third stage: In this stage, to identify new CVEs for emerged concepts that seemed incomplete and needed further analysis, keyword-based selection of CVEs was performed. We selected 110 CVEs related to these concepts. For example, for program architecture category we searched CVEs based on ”Architecture”, ”Model”, ”Event Driven”, ”Master Slave”, ”Client Server”, etc.

4.1.3 Open Coding

Open coding process analyzes collected data for each vulnerability and annotates them with codes (concepts) [18, 31]. In this step of the GT, the information gathered for each vulnerability (description, discussions, exploitation mechanism, and patches, *etc.*) are reviewed, key points that are related to *Entry Point*, *Target* or *Mechanism* are annotated, and codes are assigned to the annotated key points. Code is a phrase that summarizes the key point in a descriptive way. Defined codes are assigned to the three general groups of attack surface components based on their relevance. Figure 4.3 shows the data that was collected for CVE-2020-7248. In addition to the information available in MITRE website, data were collected from its security advisory (for exploit information), related GitHub repository (for source code), and CWE. The key points that are highlighted in red were extracted from both source code and descriptions. Codes which are defined based on the identified key points are assigned to the *Entry Point*, *Target*, and *Mechanism*:



CVE-2020-7248

Description: libubox in OpenWrt before 18.06.7 and 19.x before 19.07.1 has a **tagged binary data JSON serialization** vulnerability that may cause a **stack based buffer overflow**.

References:
https://github.com/openwrt/openwrt/commits/master
https://openwrt.org/advisory/2020-01-31-2

Related CWE:
CWE-787: Out-of-bounds Write
...

Codes: Entry points:"..."; **Targets:**"stack memory", "memory manipulation statement"; **Mechanisms:** "Serialization/Deserialization".

CWE-787

Description: The software writes data past the end, or before the beginning, of the **intended buffer**.

Example:
. . . ..
In the following example, it is possible to request that **memcpy** move a much larger segment of memory than assumed:
(bad code)
Example Language: C

```
int returnChunkSize(void *) {
/* if chunk info is valid, return the size of usable memory, * else, return -1 to indicate an error */
...}
int main() {...
        memcpy(destBuf, srcBuf, (returnChunkSize(destBuf)-1));
...
}
```

. . . .

Codes: Entry points:"..."; **Targets:**"memory copy statement", "memcpy() function"; **Mechanisms:** "Serialization/Deserialization".

Security Advisory 2020-01-31-2

Description: Possibly exploitable vulnerability exists in the libubox library of OpenWrt, specifically in the parts related to **JSON conversion of tagged binary data, so called blobs**. An attacker could possibly exploit this behavior by providing specially **crafted binary blob or JSON** which would then be translated into blob internally. This malicious **blobmsg input**blobmsg input would contain blob attribute holding large enough numeric value of type double which then processed by **blobmsg_format_json** would **overflow the buffer array designated for JSON output allocated on the stack**. The libubox library is a core component in the OpenWrt project and utilized in other parts of the project. Those interdependencies are visible by looking up of the above mentioned vulnerable **blobmsg_format_json function** in the project's LXR[1], which reveals references in netifd, procd, ubus, rpcd, uhttpd. .... libubox in OpenWrt before 18.06.7 and 19.x before 19.07.1 has a **tagged binary data JSON serialization** vulnerability that may cause a **stack based buffer overflow**.

Exploit Info:
In order to exploit this vulnerability, a malicious attacker would need to provide specially **crafted binary blobs or JSON input** to blobmsg_format_json, thus creating **stack based overflow** condition during **serialization of the double value into the JSON buffer**. It was verified, that its possible to crash rpcd by following shell command: **ubus call luci getFeatures ´{ "banik": 00192200197600198000198100200400.1922 }´**

References:
https://lxr.openwrt.org/ident?i=blobmsg_format_json
https://github.com/openwrt/packages/blob/master/utils/auc/src/auc.c

Codes: Entry points:"service request", "ubus command" ,"binary blob/json (input data)" , "procedure call", "message handler"; **Targets:**"stack memory", "buffer array"; **Mechanisms:**"JSON Serialization".

Openwrt Github

packages/utils/auc/src/auc.c:

```
...
39 #include <libuboxvlist.h>
40 #include <libuboxblobmsg_json.h>
41 #include <libuboxavl-cmp.h>
. . .
679 DPRINTF("status code: %d\n", cl— >status_code);
680      DPRINTF("headers:\n%s\n",blobmsg_format_json_indent (cl— >meta, true, 0));
681      blobmsg_parse(header_policy,      __H_MAX,      tb, blob_data(cl— >meta), blob_len(cl->meta)); ...
```

OpenWrt.org Cross Reference

libuboxblobmsg_json.h

```
. . . .
40 static inline char *blobmsg_format_json_indent(struct blob_attr *attr, bool list, int indent)
41 {
42 return blobmsg_format_json_with_cb(attr, list, NULL, NULL, indent);
43 } ...
```

libuboxblobmsg_json.c

```
322 char *blobmsg_format_json_with_cb(struct blob_attr *attr, bool list, blobmsg_json_format_t cb, void *priv, int indent)
323{...
336                 blobmsg_format_json_list(&s,   blobmsg_data(attr), blobmsg_data_len(attr), array);
354 ...}
static bool blobmsg_puts(struct strbuf *s, const char *c, int len)
130 {
131             size_t new_len; 132 char *new_buf;
133
134             if (len <= 0)
135                     return true;
136
137             if (s— >pos + len >= s— >len) {
138                     new_len = s— >len + 16 + len;
139             new_buf = realloc(s— >buf, new_len);
140             if (new_buf)
141                     return false;
146             ...
147             memcpy(s— >buf + s— >pos, c, len);
148             s— >pos += len;
149             return true;
150 }
```

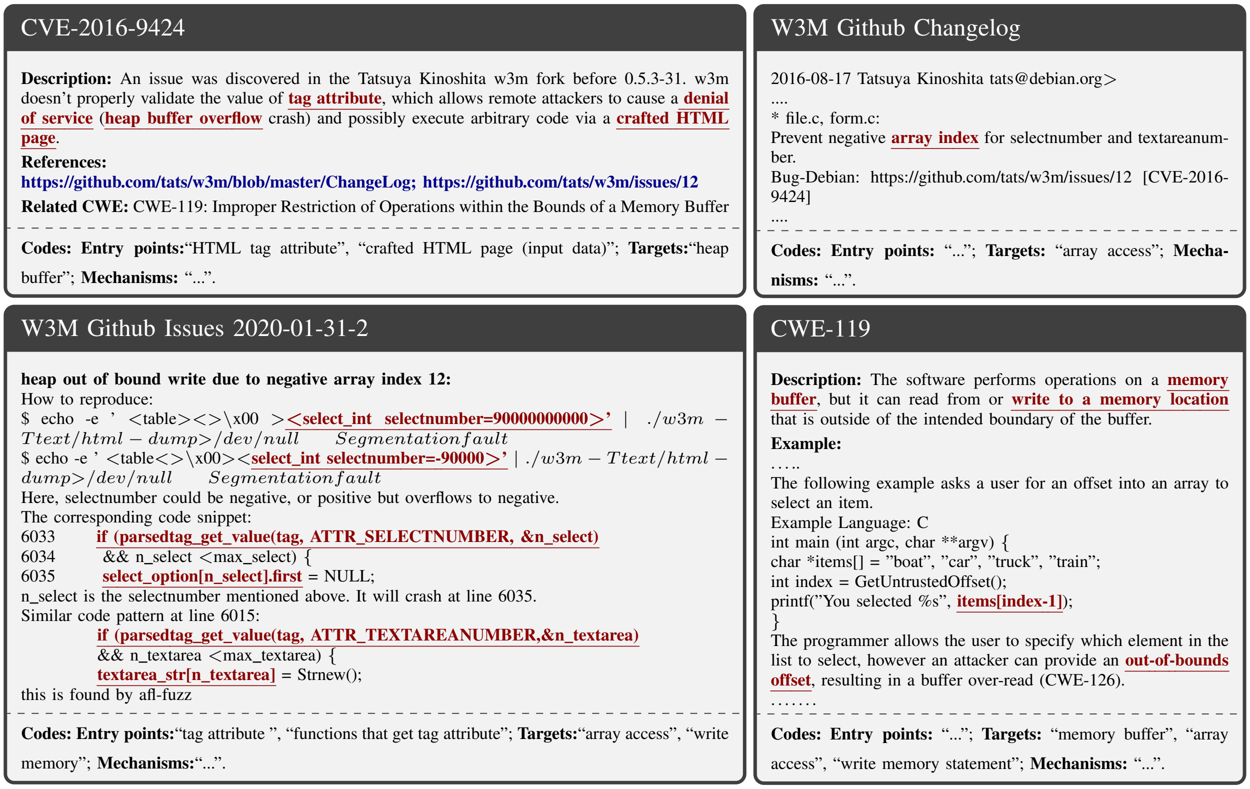
Codes: Entry points: "binary blob(input data) "; **Targets:** "stack memory", "memory copy statement"; **Mechanisms:** "Serialization/Deserialization".

Figure 4.3: Open coding of data collected for CVE-2016-9424

- *Entry Point*: service request, ubus command, binary blob/json (input data);
- *Target*: stack memory, memory manipulation statement, memory copy statement, memcpy() function, buffer array;
- *Mechanism*: Serialization/Deserialization, JSON serialization.

Figure 4.4 also shows the data collected for the other CVE (CVE-2016-9424), highlighted key points, and the defined codes. Codes which are defined for this CVE during open coding process are:

- *Entry Point*: tag attribute, crafted HTML page, functions that get tag attribute;
- *Target*: heap, array access, write memory, memory buffer;



CVE-2016-9424

Description: An issue was discovered in the Tatsuya Kinoshita w3m fork before 0.5.3-31. w3m doesn't properly validate the value of **tag attribute**, which allows remote attackers to cause a **denial of service** (**heap buffer overflow** crash) and possibly execute arbitrary code via a **crafted HTML page**.

References:
https://github.com/tats/w3m/blob/master/ChangeLog; https://github.com/tats/w3m/issues/12

Related CWE: CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer

Codes: Entry points:"HTML tag attribute", "crafted HTML page (input data)"; **Targets:**"heap buffer"; **Mechanisms:** "...".

W3M Github Changelog

2016-08-17 Tatsuya Kinoshita tats@debian.org>
....
* file.c, form.c:
Prevent negative **array index** for selectnumber and textareanumber.
Bug-Debian: https://github.com/tats/w3m/issues/12 [CVE-2016-9424]
....

Codes: Entry points: "..."; **Targets:** "array access"; **Mechanisms:** "...".

W3M Github Issues 2020-01-31-2

heap out of bound write due to negative array index 12:
How to reproduce:
$ echo -e ' <table><>\x00 >**<select_int selectnumber=90000000000>'** | ./w3m − Ttext/html − dump>/dev/null Segmentationfault
$ echo -e ' <table<>\x00><**select_int selectnumber=-90000>'** | ./w3m − Ttext/html − dump>/dev/null Segmentationfault
Here, selectnumber could be negative, or positive but overflows to negative.
The corresponding code snippet:

```
6033    if (parsedtag_get_value(tag, ATTR_SELECTNUMBER, &n_select)
6034     && n_select <max_select) {
6035     select_option[n_select].first = NULL;
```

n_select is the selectnumber mentioned above. It will crash at line 6035.
Similar code pattern at line 6015:

```
        if (parsedtag_get_value(tag, ATTR_TEXTAREANUMBER,&n_textarea)
        && n_textarea <max_textarea) {
        textarea_str[n_textarea] = Strnew();
```

this is found by afl-fuzz

Codes: Entry points:"tag attribute ", "functions that get tag attribute"; **Targets:**"array access", "write memory"; **Mechanisms:**"...".

CWE-119

Description: The software performs operations on a **memory buffer**, but it can read from or **write to a memory location** that is outside of the intended boundary of the buffer.

Example:
.....
The following example asks a user for an offset into an array to select an item.
Example Language: C

```
int main (int argc, char **argv) {
char *items[] = "boat", "car", "truck", "train";
int index = GetUntrustedOffset();
printf("You selected %s", items[index-1]);
}
```

The programmer allows the user to specify which element in the list to select, however an attacker can provide an **out-of-bounds offset**, resulting in a buffer over-read (CWE-126).
....

Codes: Entry points: "..."; **Targets:** "memory buffer", "array access", "write memory statement"; **Mechanisms:** "...".

Figure 4.4: Open coding of data collected for CVE-2020-7248

- *Mechanism*: -

Identified codes are constantly refined during the open coding process, leading to core categories and their associated concepts.

4.1.4 Constant Comparisons

Vulnerability reports were annotated either by using the existing codes or creating new ones (if existing codes were not suitable for a newly analyzed vulnerability report). During the analysis of vulnerability reports, the existing concepts/categories were compared against vulnerability reports to evolve categories and data interpretations. The overall goal of the open coding and constant comparison analysis is to identify the core concepts and categories related to the attack surface.

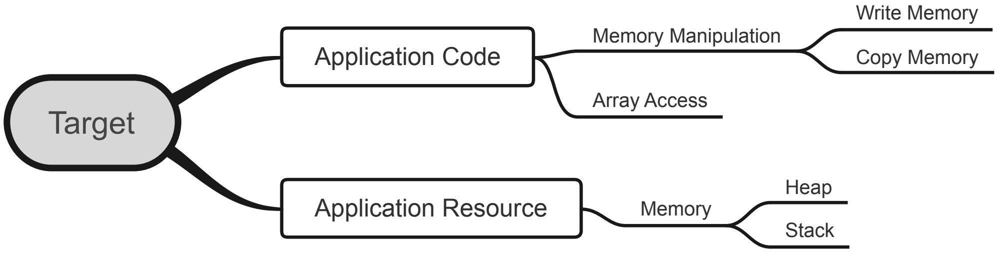


Figure 4.5: Mindmap example for *Targets*

4.1.5 Memoing

Memos are notes, diagrams, or sketches that researchers use to describe their preliminary ideas about the concepts and conceptual relationship between categories. After memoing, the researcher has stacks of memos in hand and puts them in an organized order by doing memo sorting [18]. Memoing is performed throughout the entire process of data coding and defining taxonomies. In this research, we used mind map diagrams to show the relationship between codes/concepts to identify core categories.

4.1.6 Axial Coding

During axial coding, new codes are defined as a result of identifying new relationships [18, 96] between defined codes and categorizing them into higher level concepts. Axial coding was performed based on the codes/concepts defined for each core category separately. For instance, as shown in Figure 4.5, codes gathered from CVE-2020-7248 (Figure 4.3) and CVE-2016-9424 (Figure 4.4) for the *Targets* category were categorized into two higher level concepts: 1) *Application Code* (*Memory Manipulation*, *Array Access*) and 2) *Application Resource* (*Memory*). Three higher level concepts were defined for the *Entry Points*: 1) *Input data* (e.x. *BLOB/JSON*, *HTML Page*, and *HTML Tag*), 2) *Methods Receive Inputs*, and 3) *Service/Server Requests* (Figure 4.6). In this step, CVE instances were revisited in order to further refine their codes and attack surface components.

Data Analysis Instruments

For the GT analysis, we developed a custom-built web-based tool to support our coding activities. This tool presents the information retrieved for each vulnerability report, codes, and defined concepts.

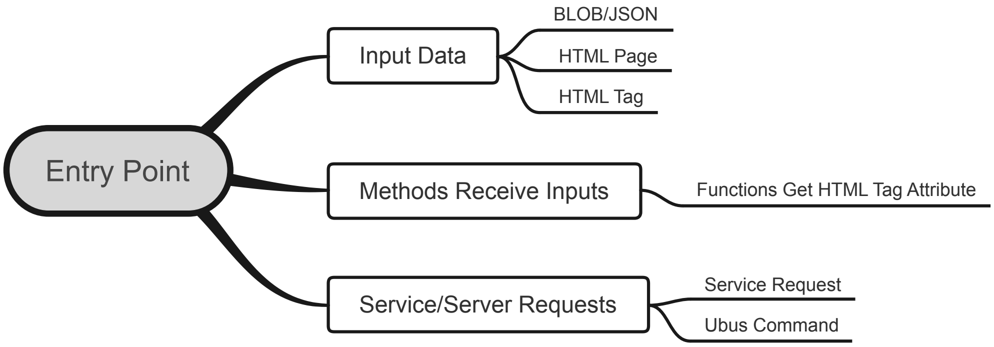


Figure 4.6: Mindmap example for *Entry Points*

4.1.7 Selective Coding

In the last step of the GT analysis, final categories are defined by sorting the defined concepts and associating them with the central branches, *i.e.*, the Program (P), Code (C), System (S) and Network (N) [96]. During this process, we integrate previously identified concepts and structure them into higher level of abstraction (theories) if needed. Selective coding re-organizes categories developed during axial coding and define final taxonomies for entry point, target and mechanism.

4.2 Attack Surface Detection

4.2.1 Term weighting Approach

This section discusses an approach based on Natural Language Processing (NLP) to automatically identify attack surface components in source code (to answer RQ4). The proposed approach relies on the frequent terms related to each attack surface component to identify attack surface components in source code. The proposed approach has been evaluated for Java programming language. In order to identify frequent terms related to each attack surface components a concise description of the components are required. To collect this information, we rely on Standard Java APIs. We collect all *methods* of the standard java classes in Oracle database [81] and manually review them to identify methods which are related to each component. The total number of methods are 38,568. For each method we considered *Class Name*, *Identifiers and Return Type*, *Method Name*, *Input Parameters*, and *Description of the Method* as features that describe it. Figure 4.7 and 4.8 show some of the sample methods collected to describe the *Read_Input_Stream* component defined in the *entry point* category and the *Execute_SQL_Command* component defined in the *target* category.

To evaluate the model, for each attack surface components, sample files collected from GitHub repository. Methods in each file are extracted and labeled. For each attack surface component label "0" shows the methods that are not associated with that attack surface component and label "1" shows the methods which are associated with that attack surface component. For instance, method *getSoundbank* which is shown in Figure 4.9 is considered as positive for *Input_Stream_File* component because it reads from *AudioInputStream* in line 8 and 15. Method *runTest* in the Figure considered as positive function for *Execute_SQL_Command* component since it executes SQL queries in lines 4, 5, 16, and 23.

Training

All data is preprocessed using standard information retrieval techniques (*tokenization*, *removing stop words*, *camel case split*, and *lemmatization*) and each API description is transformed into a vector of terms. The training phase takes a set of API descriptions for each attack surface component as input, and produces a set of indicator terms for that attack surface component. These terms are considered as representative of the attack surface component. For example, a term such as *stream*, is found more commonly in the APIs related to the *Input_Stream_File*. More formally, let c be a specific attack surface component such as *Input_Stream_File*. Indicator terms

Entry Point ->Input_Stream_File

DataInputStream byte readByte() See the general contract of the readByte method of DataInput.

DataInputStream char readChar() See the general contract of the readChar method of DataInput.

DataInputStream double readDouble() See the general contract of the readDouble method of DataInput.

DataInputStream float readFloat() See the general contract of the readFloat method of DataInput.

FileReader FileReader(File file) Creates a new FileReader, given the File to read from.

FileReader FileReader(FileDescriptor fd) Creates a new FileReader, given the FileDescriptor to read from.

FileReader FileReader(String fileName) Creates a new FileReader, given the name of the file to read from.

FileImageInputStream int read() Reads a single byte from the stream and returns it as an int between 0 and 255.

FileImageInputStream int read(byte[] b, int off, int len) Reads up to len bytes from the stream, and stores them into b starting at index off.

Streamable void _read(InputStream istream) Reads data from istream and initalizes the value field of the Holder with the unmarshalled data.

StreamSource InputStream getInputStream() Get the byte stream that was set with setByteStream.

StringHolder void _read(InputStream input) Reads the unmarshalled data from input and assigns it to the value field of this StringHolder object.

StringSeqHolder void _ read(InputStream i) Reads data from istream and initalizes the value field of the Holder with the unmarshalled data.

StringValueHelper Serializable read_value(InputStream istream)

...

Figure 4.7: Description of Java APIs related to *Input_Stream_File* component in entry point category

of component c are mined by considering all API descriptions that are related to that component. Each term t is assigned a weight score Pr_c(t) that corresponds to the probability that a particular term t identifies the attack surface component c. Pr_c(t) is defined as:

$$Pr_c(t) = \frac{\sum_{APIdesc \in c} freq(t, APIdesc)}{N_c} * \frac{N_c(t)}{N(t)} * \frac{N_c(t)}{N_c} \tag{4.1}$$

where $N_c(t)$ is the number of APIs related to attack surface component c that contain term t, $N(t)$ is total number of APIs contain term t, $N_c(t)$ is number of APIs related to attack surface component c and *freq(t, APIdesc)* is frequency of term t in APIdesc c. After computing the probability of the terms top 0.1 percent of the terms are considered as indicator term for each attack surface component.

Target ->Execute_SQL_Command
PreparedStatement boolean execute() Executes the SQL statement in this PreparedStatement object, which may be any kind of SQL statement.
PreparedStatement default long executeLargeUpdate() Executes the SQL statement in this PreparedStatement object, which must be an SQL Data Manipulation Language (DML) statement, such as INSERT, UPDATE or DELETE; or an SQL statement that returns nothing, such as a DDL statement.
PreparedStatement ResultSet executeQuery() Executes the SQL query in this PreparedStatement object and returns the ResultSet object generated by the query.
RowSet void execute() Fills this RowSet object with data.
Statement boolean execute(String sql) Executes the given SQL statement, which may return multiple results.
Statement boolean execute(String sql, int autoGeneratedKeys) Executes the given SQL statement, which may return multiple results, and signals the driver that any auto-generated keys should be made available for retrieval.
Statement boolean execute(String sql, int[] columnIndexes) Executes the given SQL statement, which may return multiple results, and signals the driver that the auto-generated keys indicated in the given array should be made available for retrieval.
Statement int[] executeBatch() Submits a batch of commands to the database for execution and if all commands execute successfully, returns an array of update counts.
Statement default long[] executeLargeBatch() Submits a batch of commands to the database for execution and if all commands execute successfully, returns an array of update counts. ...

Figure 4.8: Description of Java APIs related to *Execute_SQL_Command* component in target taxonomy

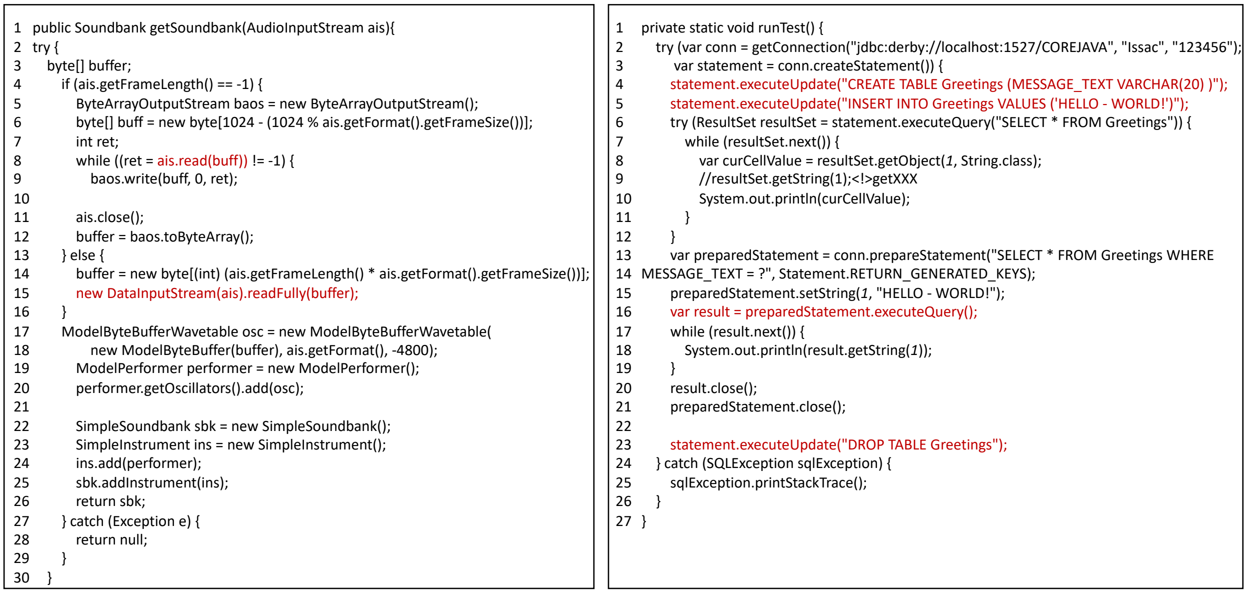
Classification

During the classification phase, the indicator terms computed in training phase are used to evaluate the likelihood $\mathrm{Pr_c(m)}$ that a given method m is associated with the the attack surface component c. The classification score that method m is associated with attack surface component c is defined as follows:

$$Pr_c(m) = \frac{\sum_{t \in f \cap I_c} Pr_c(t)}{\sum_{t \in I_c} Pr_c(t)} \tag{4.2}$$

where $\mathrm{I_c}$ is indicator terms for attack surface component c. The probability score is computed as the sum of the term weights of all indicator terms of attack surface component c that are contained in method m divided by the sum of the term weights for all attack surface component c. The probabilistic classifier for a given attack surface component c will assign a higher score $\mathrm{Pr_c(m)}$ to method m that contains several strong indicator terms of the component c.

Methods are considered to be related to a given attack surface component c if their classification score is higher than a specific threshold.



```java
public Soundbank getSoundbank(AudioInputStream ais){
try {
   byte[] buffer;
      if (ais.getFrameLength() == -1) {
         ByteArrayOutputStream baos = new ByteArrayOutputStream();
         byte[] buff = new byte[1024 - (1024 % ais.getFormat().getFrameSize())];
         int ret;
         while ((ret = ais.read(buff)) != -1) {
            baos.write(buff, 0, ret);

         ais.close();
         buffer = baos.toByteArray();
      } else {
         buffer = new byte[(int) (ais.getFrameLength() * ais.getFormat().getFrameSize())];
         new DataInputStream(ais).readFully(buffer);
      }
      ModelByteBufferWavetable osc = new ModelByteBufferWavetable(
            new ModelByteBuffer(buffer), ais.getFormat(), -4800);
         ModelPerformer performer = new ModelPerformer();
         performer.getOscillators().add(osc);

         SimpleSoundbank sbk = new SimpleSoundbank();
         SimpleInstrument ins = new SimpleInstrument();
         ins.add(performer);
         sbk.addInstrument(ins);
         return sbk;
      } catch (Exception e) {
         return null;
      }
   }
```

```java
private static void runTest() {
   try (var conn = getConnection("jdbc:derby://localhost:1527/COREJAVA", "Issac", "123456");
        var statement = conn.createStatement()) {
      statement.executeUpdate("CREATE TABLE Greetings (MESSAGE_TEXT VARCHAR(20) )");
      statement.executeUpdate("INSERT INTO Greetings VALUES ('HELLO - WORLD!')");
      try (ResultSet resultSet = statement.executeQuery("SELECT * FROM Greetings")) {
         while (resultSet.next()) {
            var curCellValue = resultSet.getObject(1, String.class);
            //resultSet.getString(1);<!>getXXX
            System.out.println(curCellValue);
         }
      }
      var preparedStatement = conn.prepareStatement("SELECT * FROM Greetings WHERE
MESSAGE_TEXT = ?", Statement.RETURN_GENERATED_KEYS);
      preparedStatement.setString(1, "HELLO - WORLD!");
      var result = preparedStatement.executeQuery();
      while (result.next()) {
         System.out.println(result.getString(1));
      }
      result.close();
      preparedStatement.close();

      statement.executeUpdate("DROP TABLE Greetings");
   } catch (SQLException sqlException) {
      sqlException.printStackTrace();
   }
}
```

Figure 4.9: Sample functions related to input stream/file and execute SQL command attack surface components components

Evaluation

To answer RQ5 Precision, Recall, and Fscore in Equations 4.3, 4.4, and 4.5 were used as performance measures. In Equation 4.3 , precision represents the quality of a positive prediction made by the model. In Equation 4.4 represents the percentage of positive methods associated with the attack surface component that are correctly identified. Fscore is harmonic mean of Precision and Recall.

$$Precision = \frac{\text{TP}}{\text{FP+TP}} \tag{4.3}$$

$$Recall = \frac{\text{TP}}{\text{FN+TP}} \tag{4.4}$$

$$Fscore = 2 * \frac{\text{Precision*Recall}}{\text{Precision+Recall}} \tag{4.5}$$

4.2.2 CodeBERT NLI Approach

Natural Language Inference (NLI) is a task that involves assessing the relationship between two components: the premise and the hypothesis. The primary objective of NLI is to determine whether the premise entails, contradicts, or remains neutral towards the hypothesis. In our study, we

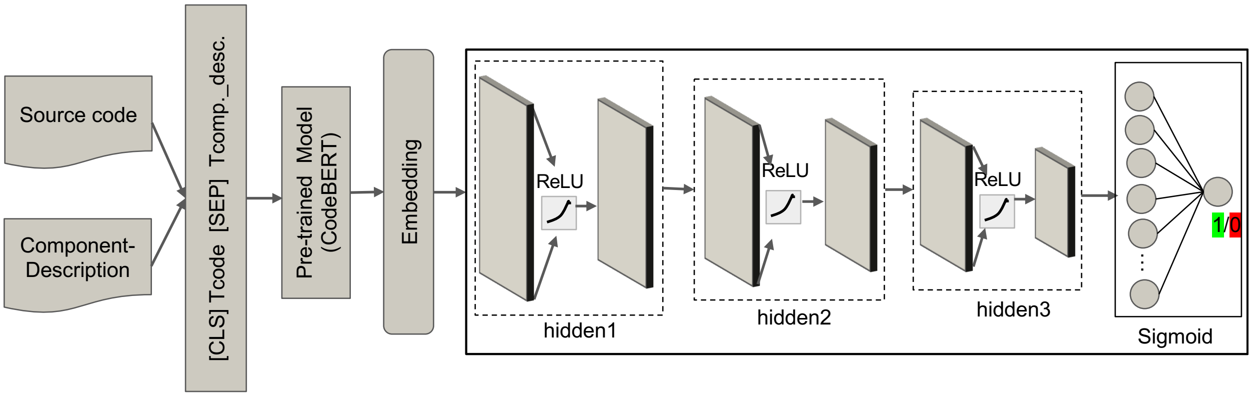


Figure 4.10: CodeBERT NLI model

formulate the identification of attack surface component into an entailment task, and utilize an NLI approach to tackle this (to answer RQ6). To this end, we train the model with a source code and a description of the attack surface component. The model's objective is to determine whether the given source code contains the attack surface component or not. The output of the model is binary, with "True" indicating the presence of the component and "False" representing its nonexistence.

In this model, we utilized CodeBERT [24], a language-source code pretraining model. CodeBERT has been trained on large corpus of code and code-related text, which allows it to learn representations that capture both syntactic and semantic aspects of code. The model and preprocessing steps are shown in figure 4.10.

In order to provide the model with appropriate data format, we adopt a data preparation process that conforms to the format required by the model. This transformation of the input data is depicted in Equation 4.6:

$$[Code, AttackComponentDescription] \rightarrow [CLS]T_{\text{code}}[SEP]T_{\text{Att. Comp. Desc.}}[SEP]$$

In Equation 4.6, the tokenizer takes the raw input, which includes both the source code and its corresponding component description. Table 4.1 shows the description provided for each attack surface components. Through this tokenization process, the model generates two token sequences: T_{code} and $T_{\text{Att. Comp. Desc.}}$. These sequences are accompanied by a [CLS] token at the beginning and a [SEP] token to separate the code from the text description. This resulting representation is

Attack Surface Component	**Description**
Input_stream_file	This method reads data from inputstream, istream or file.
Execute_OS_Command	This method executes a command or run, submit, cancel, or stop a task or service on operating system (OS).
Execute_SQL_Command	This method executes a sql query or sql statement (insert, selete, update) or a batch of commands and return results or fill row sets.
User_Input	This method reads, gets, scans user input from console via reader or scanner.
Serialization_Deserialization	This method marshals or serailizes data objects or unmarshals or deserializes them to an object such as xml tree.
Read_Socket	This method reads a sequence of bytes form socket buffer, channel, or input stream.
OS_Signal_Handler	This method handles system signals and interrupts such as aquire, read, signal, or await for lucks.
Weak-Encryption	This method does encoding or decoding by weak encription algorithm such base64, sha1, md5, dsa using hashing or encription algorithms (such as cipher or digest classes).
Reflection	This method finds or lookups classes or interfaces. instantiates new objects, load classes, invokes methods or constructors and get or set field values in classes using reflection.
None	This method does not contain any attack components.

Table 4.1: Attack surface components' description

then fed into the pretraining model for further processing. In order to adapt the pretrained model, we considered three hidden layers with ReLU activation function and an output layer with Sigmoid activation. Afterward, the processed samples are passed to the pretraining model, which generates vector representations for each sample. These representations are then inputted into a feed-forward model to perform the inference task. During training, the labels associated with each model is used as the groundtruth, and will be utilized to train the model. The CodeBERT NLI model produces a binary output, represented by 0 or 1, signifying the entailment relationship between the given source code and the corresponding textual attack component description.

The CodeBERT NLI Approach was implemented using python 3.8 and torch version 0.14.0. All parameters are shown in Table 4.2. As a base model we use RobertaModel using *codebert-base* as pretrained model. We conduct all the experiments on a 2.8 GHz Quad-Core Intel Core i7 CPU with an Intel Iris Plus Graphics 655 1536 MB.

Parameter	Value
N_EPOCHS	10
HIDDEN_DIM	768
batch_size	16
max_seq_length	512

Table 4.2: CodeBERT NLI implementation parameters

4.2.3 Using ChatGPT to Identify Attack Surface Components

Chat Generative Pre-Trained Transformer (ChatGPT) [79], released by OpenAI in November 2022, has become one of the most advanced language models in existence, with numerous applications [79]. It is a conversational large language model (LLM) based on Reinforcement Learning from Human Feedback (RLHF). ChatGPT attracted widespread attention from software engineering community to leverage its capabilities in solving problems related to code and security. Cheshkov et al. [15] evaluated the capability of ChatGPT model in identifying vulnerabilities. Ma et al. [59] evaluated the capability of ChatGPT in understanding the syntax, static, and dynamic behaviour of the code . Xia and Zhang [108] leverage ChatGPT model for path generation and program repair. Tian et al. [102] showed that ChatGPT is effective in code generation, program repair, and code summarization tasks .

In order to evaluate the performance of ChatGPT in identifying different attack surface components (to answer RQ8), we used the API provided by OpenAI to send requests to ChatGPT. These prompts were designed to ask whether the Java method contains a specific attack surface component, with the goal of generating a ”Yes” or ”No” response from both. Table 4.3 shows the questions that are asked from ChatGPT for identifying the methods that contain attack surface components. For some of the methods, ChatGPT didn’t provide the explicit ”Yes” or ”No” response. Therefore, we manually reviewed this responses to see if the ChatGPT response is ”Yes” or ”No”. Table 4.4 describe sample questions asked from ChatGPT and their related responses. First example, shows the *deleteStudentfromDM* method that executes an update sql query. Based on the question related to *Execute_SQL_Command* that has been asked, ChatGPT clearly answers ”Yes” and explain the method. The second example is a *deserialize()* method. Based on the question related to *Serialization_Deserialization* that has been asked ChatGPT answers ”No” but in the description it explains that it is a deserialization method for byte array. This example shows that ChatGPT responses are very sensitive to the questions that we ask. It asnwers ”No” based on the first part of question. The third example is *textDrawableMazeArch()* method that was asked from

Attack Surface Component	**Prompt**
Input_stream_file	Does this code snippet directly read data from fileinputstream,inputstream, istream, file or any other type of input stream?
Execute_OS_Command	Does this method execute a command, task, or service or execute, run, submit, cancel, shutdown, or stop a task, thread or service on operating system (OS)?
Execute_SQL_Command	Does this method execute a sql/database query or statement (insert, selete, update) or a batch of sql commands and return results or fill row sets?
User_input	Does this method directly read, get, scan user input from console via reader or scanner?
Serialization_deserialization	Does this method directly marshal or serailize data objects or unmarshalor deserialize them to an object such as xml tree?
Read_socket	Does this method read a sequence of bytes form socket buffer, channel, or socket stream, or connect to or open any network channel?
Os_signal_handler	Does this method handle system signals and interrupts such as aquire, read, signal, or await for lucks or doing synchronization?
Weak-encryption	Does this method perform encoding, decoding, encryption, or decryption using weak encription algorithm such base64, sha1, md5, dsa using hashing or encription algorithms?
Reflection	Does this method directly perform code reflection such as lookup classes/interfaces, invokes class methods/constructors and get or set class fields using reflection?

Table 4.3: Prompts asked from ChatGPT for each attack surface component

ChatGPT if it is related to *OS_Signal_Handler* class or not. ChatGPT answers ”Yes” followed by ”... does not directly handle system signals ...”. The forth example is *getTotalAmbulance()* method that is related to *Weak-Encryption* attack surface component because it uses Base64 algorithm. ChatGPT does not answer ”Yes” or ”No” and answer the question by explaining it. Therefore, we did not only rely on ”Yes” or ”No” part of responses and checked the description of the answer. For instance, for the third example we considered the response of ChatGPT as ”No’ for evaluating the performance of ChatGPT in identifying attack surface components.

Prompt	Response

Q: Does this method execute a sqldatabase query or statement (insert, select, update) or a batch of sql commands and return results or fill row sets?
public static void deleteStudentFromDB(String lastname String name) {
String delete = ”DELETE FROM ” + Constant.STUDENT_TABLE +
” WHERE lastname = ’” + lastname + ”’ and name = ’” + name + ”’”
try {
PreparedStatement preparedStatement = getConnection().
prepareStatement(delete)
preparedStatement.executeUpdate()
} catch (SQLException | ClassNotFoundException e) {
e.printStackTrace()
}}
	R: Yes, the provided method executes a SQL database query. It specifically performs a DELETE operation on the specified table (Constant.STUDENT_TABLE) based on the provided lastname and name values. The deleteStudentFromDB method does not return any results or fill row sets; it simply executes the delete statement to remove the matching rows from the database table.

Q: Does this method directly marshal or serailize data objects or unmarshalor deserialize them to an object such as xml tree?
public ObjectInputStream deserialize(ObjectName name byte[] data)
throws OperationsException
{
return mbeanServer\.deserialize(name data)
}
	R: No, the method you provided does not directly marshal or serialize data objects. Instead, it is used to deserialize or unmarshal a byte array representing an object, using an ObjectInputStream. The method takes an ObjectName and a byte array as parameters, and returns an ObjectInputStream.

Q: Does this method handle system signals and interrupts such as aquire, read, signal, or await for lucks or doing synchronization?” private static boolean testDrawableMazeArch() { boolean pass = true int test = 1 int cnt DrawableSquare[][] squares = new DrawableSquare[1][1] squares[0][0] = new DrawableSquare(true true true true 0 0) DrawableMaze drMaze = new DrawableMaze(squares 1 1)	**R:** Yes, this method does not directly handle system signals and interrupts. It is a test method designed to check the architecture of a DrawableMaze class. It checks various properties of the class such as the number of constructors and interfaces, the superclass, the number and visibility of methods and fields. It does not perform any synchronization or handle system signals and interrupts.

Q: Does this method perform encoding, decoding, encryption, or decryption using weak encryption algorithm such base64, sha1, md5, dsa using hashing or encryption algorithms (such as cipher or digest classes)?
public @ResponseBody JsonResponse<Object> getTotalAmbulance
(Model model @RequestParam(”country”) String encodecountry
@RequestParam(”date”) String encodedate @RequestParam(”type”)
String encodetype HttpSession session) {
logger.info(”Method : getTotalAmbulance starts”)
byte[] encodeByte1 = Base64.getDecoder().decode(encodecountry.getBytes())
byte[] encodeByte2 = Base64.getDecoder().decode(encodedate.getBytes())
byte[] encodeByte3 = Base64.getDecoder().decode(encodetype.getBytes())
String country = (new String(encodeByte1))
String date = (new String(encodeByte2))
String type = (new String(encodeByte3))
...
}
	R: This method performs decoding using the Base64 algorithm. It decodes the values of the parameters ”encodecountry”, ”encodedate”, and ”encodetype” from Base64 encoded strings into their original values. The decoded values are then used in the REST API call to retrieve the total ambulance data.
	The method does not perform any encryption, decryption, or weak encryption algorithms such as SHA-1, MD5, or DSA. It mainly focuses on decoding using the Base64 algorithm.

Table 4.4: Sample prompts asked from ChatGPT and their response

Chapter 5

Results

5.1 Results of Grounded Theory Analysis

Based on the concepts emerged at the end of our analysis, we find that each core category *Entry Point*, *Target*, and *Mechanism* can be divided into four major groups, *i.e.*, Code (C), Program (P), System (S), and Network (N). From *Where* the attackers are entering into a system (entry points), *what* they are targeting for (targets) and *how* they are reaching the targets (mechanisms) are all related to the source *Code* of a software, its executable version named *Program*, the *System* that application is installed on, and the *Network* that the system is interacting with. The results of this qualitative study learned from reviewing CVEs and CWEs are presented based on three research questions associated with *Entry Points*, *Targets* and *Mechanisms*:

5.1.1 Entry Points

Figure 5.1 shows the key categories were defined for entry points based on the concepts identified at the end of our analysis to answer to Research Question #1. Entry points are defined based on four core categories:

Code

This category represents parts of source code that an attacker can leverage to enter a system. As shown in Figure 5.1 they are categorized into three sub categories:

① **User Interface (UI)** defines components in the UI that can be used by attackers to enter a system. For example, an attacker can interact with an application through components in the graphical user interface (*Input Box*, *File Upload* [36], *RSS Feed* [36], *etc.*) or *Console*. ② **Methods/Directives** defines methods or directives that receive input. They can be parts of the code that directly receive input (*Direct Entry*) such as *Input Methods* [62,73] that receive inputs directly from *User*, *Device*, or *File* or *Handlers* that handle different requests such as *OS Signal* (Interrupts) or web requests (REST API, Java Servlet). *Indirect Entry* covers parts of the source code that indirectly receive input by loading *Code*, reading *Indirect Inputs* (such as Environment Variable, System Attributes, *etc.*) or *User Created Resources*. ③ **Configuration File** category contains accessible configuration files of an application that can act as an indirect entry point for software application.

Program

This category considers an application as an executable and defines attack surface components related to that:

① **Components** refer to special software components that open the doors for attackers, such as application components which are designed during the design phase of software development. For example, *Plugin*, *Installer Components*, *Chatting Component*, and *Authentication/login* components are software components that can be considered as entry points at the design level. ② **Maintenance/Deployment** category covers any action that is performed during *Deployment* or *Maintenance* of programs that can open the doors for attackers. *Install*, *Configuration*, and *Update* operations are defined in this category. ③ **Direct Input** category covers data that is sent to the program. Application can receive *Direct Input* from *User*, *Device*, *Operating System (OS)*, or as *Messaging Object* from other components/applications (Intent in Android). It can also receive ④ **Indirect Input** by reading/loading *Environment Variables*, *DLL Files*, OMX buffer, *System Properties*, *Virtual Machine Properties*, and *Cookies* [36].

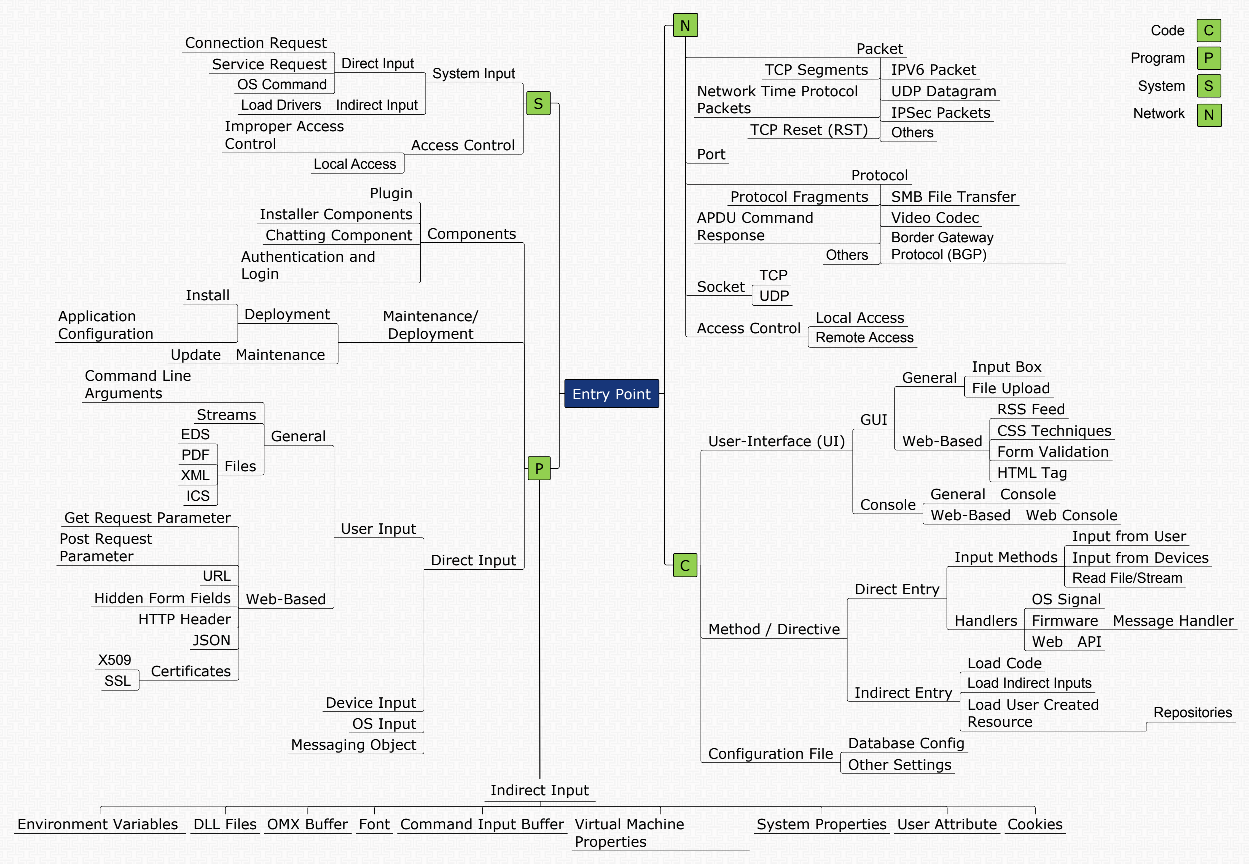


Figure 5.1: Identified *Entry Points* during attack surface analysis

System

As a platform for running software applications, can provide entry points:

① **System input** contains both *Direct Input* and *Indirect Input*. *Direct Input* represents the requests that are sent to the system and is categorized into *Connection Requests* (SSH request), *OS Commands*, and *Service Requests*. *Indirect Inputs* represent the types of data imported by the system such as *Load Driver*. ② **Access Control** contains actions that may open the door for attackers. It contains *Local Access* to the system or *Improper Access Control*.

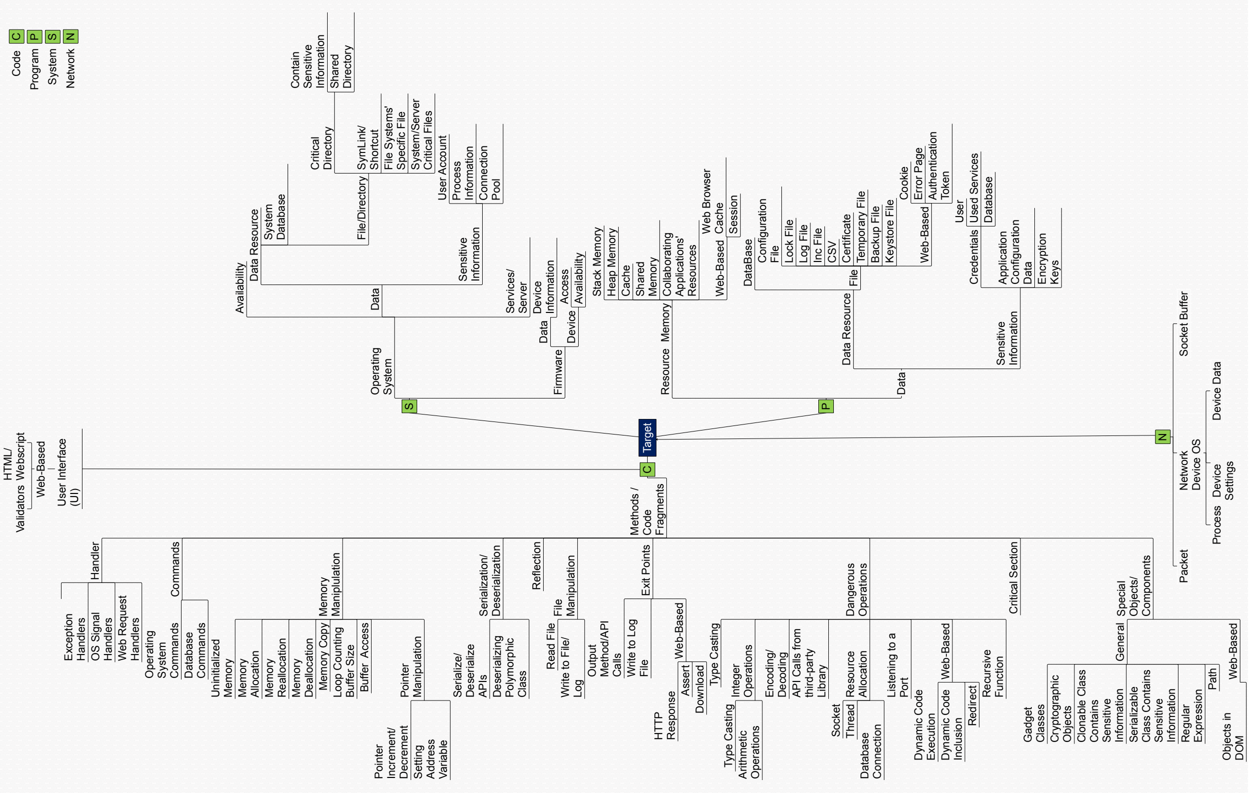


Figure 5.2: Identified *Targets* during attack surface analysis

Network

category contains the *Packet*, *Port*, *Protocol*, *Socket*, and *Access Control sub categories* : ① **Packet** represents the input data at the network level. ② **Socket** [?, ?, 60], ③ **Port**, and ④ **Protocol** [36, 43, 62] could provide entry points at the network level. ⑤ **Access Control** contains actions that open the doors for attackers such as *Local Access* to the network.

5.1.2 Targets

Figure 5.2 represents the categorization model created for *Targets* to answer to Research Question #2.

Code

This category defines source code related components that can be target of attacks. Attacker might try to access parts of source code to do malicious action. As shown in Figure 5.2 these components are categorized into two categories: ① **User Interface (UI)** category refers to components in the user interface that can be target of an attack. The analysis identified target components in this category such as *Validators* and *HTML/Webscript* that are related to *Web-Based* applications. ② **Method/Code Fragment** represents methods or other related parts of the code that can be target of attacks. As shown in Figure 5.2, parts of source code that handle requests (*Handlers*), execute *Commands* (*Database* or *OS* [73]), do *Memory Manipulation*, *Serialization/Deserialization*, *Reflection*, *Dangerous Operations* such as *Type Casting*, *Integer Operations*, *Encoding/Decoding*, *etc.*can be attractive targets for attackers. Besides that, code fragments such as *Exit points* [62,73], *Critical Section*, some *Special Objects* such as *Gadget Classes*, *Cryptography Objects*, and *Path* are other targets at the code Level.

Program

The concepts under *Program* are categorized into two general categories: ① **Resource** contains resources allocated or used by the application such as *Memory. Stack*, *Heap*, *Cache*, *Shared Memory* [43] and other memory types that are allocated, used, or read by the application. ② **Data** covers important application data that are identified as target during GT analysis. This category considers application data from two perspectives: 1) *Data Resource* which represents the location where data is stored (*Database*, *File*, *etc.*) and 2) *Sensitive Information* that represents various kinds of important data that an attacker may look for. We found important files that can be target of attacks such as *Lock File*, *Log File*, *Certificate*, and *Keystore File*. *Credentials* such as *User*, *Used Service* (Notarization Service), and *Database* credentials, *Application Configuration Data*, and *Encryption Keys* (ECDSA Secret, Master Key, *etc.*) are types of sensitive information identified during the coding process.

System

This category defines components in the OS or firmware that can be target of attacks. They can be directly accessed through attacks against the system or indirectly through attacks against software programs that use these components. These components are categorized into two major categories:

① **Operating System (OS)** contains OS and server related target components. They are categorized into different abstraction levels. *System Availability* covers actions that affect the availability of systems. For instance, a malicious System Reboot could interrupt a system and affect its availability. *System Data* categorizes different types of data in a system that can be target of attacks. It categorizes *System Data* based on the resource it is stored (*Data Resource*) and the type of the data (*Sensitive Information*). *Data Resources* can be a database on the OS (like Windows Registry [43, 62]) or an important *File/Directory* on the file system. *Critical Directory* may contain sensitive information (like etc/passwd in Linux), *Symlinks/Shortcuts* (like Unix Hard Link or Symbolic Link [60] and Windows Shortcut), *File System Specific Files* (like Data/Resource Fork of a File in HFS+ file system, Alternate Data Streams (ADS) in NTFS file system, *etc.*), and *System/Server Critical Files* (WSDL File in Web Server, Zone File in DNS Server, Node Catalogue in Distributed System, *etc.*). *User Account* information, *Process Information*, and *Connection Pool* are *Sensitive Information* in a system. *Services/Server* defines types of servers or services on a system that are usually target of attacks. For instance, SSL and NAS Servers are identified as targets in various vulnerability reports [66, 68].

② **Firmware.** category covers parts of firmware that contain *Device Information* or control *Device.*

Network

① **Packets** and information on ② **Network Devices OS** such as *Process* (Routing Engine on Routers), *Device Setting*, and *Device Data*, and also ③ **Socket Buffer** could be the target of attacks at the network level.

5.1.3 Mechanisms

This category answers the Research Question #3 by discussing mechanisms that are used at the source code, program, system, and network level that could lead to the emergence of vulnerable attack surfaces. Figure 5.3 represents the categorization model for *Mechanism.* We briefly summarize the mechanisms:

Code

Mechanisms used at the code level are categorized in three major categories:

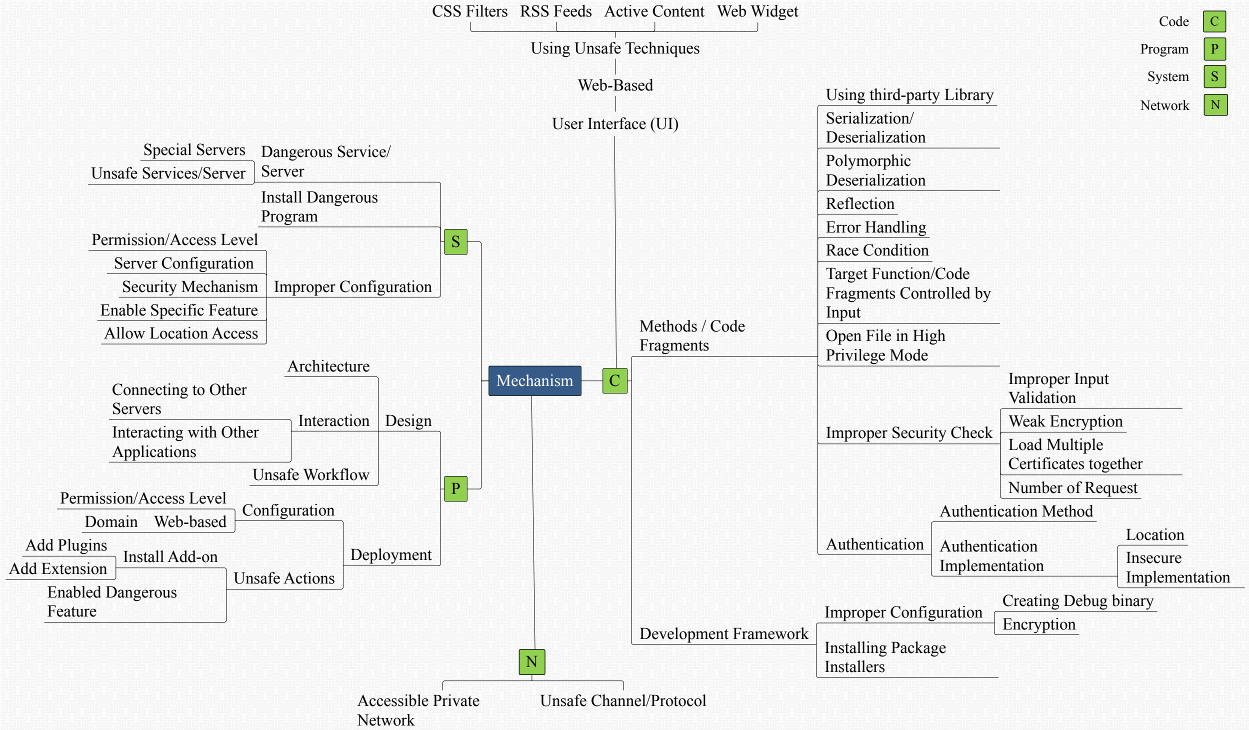


Figure 5.3: Identified *Mechanisms* during attack surface analysis

① **User Interface (UI)** defines mechanisms used in the UI to open the doors for attackers. The concepts under this category are related to *Using Unsafe Techniques* in *Web-Based* applications such as *CSS Filters*, *RSS Feeds* [36], *Active Content* [36], and *Web Widget*.

② **Methods/Code Fragments** category discusses vulnerable mechanisms used during coding such as *Using Third-party Library*, *Serialization/Deserialization*, *Polymorphic Deserialization*, *Improper Security Check*, and *Authentication*, etc. *Improper Security Check* focuses on security mechanisms that are missed or implemented incorrectly such as *Improper Input Validation* [36], *Weak Encryption*, *Load Multiple Certificates* (like system and SSL certificate), and *Number of Requests*. *Authentication* category contains *Authentication Methods* (Password, Token-based, Certificate-based, *etc.*) and *Authentication Implementation* mechanisms such as *Location* (No Server Side Authentication) and *Insecure Implementation* (use == operator instead of === for Hash Comparison or unsafe info such as IP address for Authentication). ③ **Development Framework** category discusses vulnerable mechanisms which are related to the software development frameworks, like *Improper Configuration* (*Creating Debug Binary* or Improper *Encryption*) and *Installing Package Installers*.

Program

This category categorizes the mechanism that are related to the design and deployment phase of a software system: ① **Design** category discusses mechanisms related to the design such as program *Architecture* (Client/Server, *etc.*), *Interaction* (connecting to *Other Servers* or *Other Applications*), and insecurely designed *Workflow*. ② **Deployment** category covers mechanisms during program deployment such as improper *Configuration* and *Unsafe Actions* that users can do on a system such as *Install Add-on* and *Enable Dangerous Features* [43].

System Level

category covers the following concepts:

① **Dangerous Services/Server** category refers to activating special *Servers* (e.x. SSL, NAS, and Mail) or installing/activating *Unsafe Services/Server* (like Unsafe FTP server) that can lead to emergence of attacks. ② **Dangerous Program** refers to installing special programs that can open the door for attackers (e.x. Android application that allows disabling/enabling WIFI to co-located apps [67]). ③ **Improper Configuration** refers to the configuration mechanisms that make the system vulnerable. These include *Permission/Access Level* (for File, Registery, *etc.*) [**?**, 43, 60], improper *Server Configuration*, *Security Mechanism* (Non-strict Security Mechanism in Firewall or Proxy), *Enable Specific Feature*, and *Allow Location Access*.

Network Level

mechanisms include ① **Accessible Private Network** and ② **Using Unsafe Channel/Protocol** [36, 43, 62] (using HTTP instead of HTTPS).

5.1.4 Comparing to Related Work

The search strategy of our systematic literature review [116] consists of a manual search of five sources: the ACM Digital Library, IEEE Explore Library, ScienceDirect, Springer Link, and Google Scholar. Our inclusion criteria are as follows: the work is (i) a full paper; and (ii) focus on discussing software system attack surface. Exclusion criteria are (i) position papers, short papers, keynotes, reviews, tutorial summaries, and panel discussions; (ii) not fully written in English; (iii) duplicated study; (iv) focused on attack surface outside the domain of software system; and (v) focused on attack surface of a specific type of system (e.x. IoT). We use the following search query: (Software OR Application) AND (Attack Surface OR Attack-Surface). From our manual search, we collected a total of 2,150 papers. Inclusion and exclusion criteria were applied through reading the paper's title, abstract, and keywords (if present), resulting in 30 papers. Then, in this round the inclusion and exclusion criteria were applied by reading the full papers, resulting in a remaining 8 papers. Some of the papers that were removed for further analysis, have misused the term "attack surface" (e.g. referring to software vulnerabilities). The remaining papers were carefully reviewed, to verify the extent to which the findings from our study were supported by the literature or were complementary. Limited studies have been proposed for identifying attack surface of a software system.

To evaluate the proposed attack surface categorization, it is compared with the attack surface components proposed in the literature. The comparison results (Table 5.1) show that the categorization provided by this paper covers all attack surface components introduced in the literature. The concepts which are missed in our categorization such as *Search* in *Program* and *RPC and Named Pipe* in *Network* level entry points are specific concepts that can be covered by the proposed core categories. The comparison results indicate that the proposed attack surface model differs from the previously introduced components in that it:

- Provides a comprehensive attack surface categorization that considers different aspects in *System*, *Network*, *Program*, and *Source* levels. Previous works mostly focused on defining attack surface components by low level concepts.
- Defines clear concepts that can be part of an attack surface. For instance, *Data Item* which is defined as an attack surface component in [62] is a vague concept. Based on the examples mentioned for it, our categorization clearly indicates that the data item could be program or system data.
- Provides comprehensive *Code* Level attack surface components. For example, previous studies

		Core Category	Low Level Concepts
Entry Points	**C**	UI	GUI: General (Input Box, File Upload [36]), Web-Based (RSS Feed [36], CSS Techniques, Form Validation, HTML Tag) Console: General (Console), Web-Based (Web Console)
		Method [62]/ Directive	Direct Entry: Input Methods [36, 62, 73] (Input from User, Input from Devices, Read File/Stream), Handlers (OS Signal Handler, Firmware (Message Handler), Web (API)) Indirect Entry: (Load Code, Load Indirect Inputs, Load User Created Resources (Repositories))
		Config. File	Database Config, Other Settings
	P	Components	Plugin, Installer Components, Chatting Component, Authentication and Login, Search [36]
		Maintenance/ Deployment	Deployment: Install, Application Configuration Maintenance: Update
		Direct Input	User Input: General (Command Line Arguments, Streams, Files (EDS, PDF, XML, ICS)), Web-Based (Get Request Parameter [36], Post Request Parameter, URL, Hidden Form Fields [36], HTTP Header, JSON, Certificates (X509, SSL)) Device Input, OS Input, Messaging Object
		Indirect Input	Environment Variables, DLL Files, OMX Buffer, Font, Command Input Buffer, Virtual Machine Properties, System Properties, User Attribute, Cookies [36, 62]
	S	System Input	Direct Input: (Connection Requests, Service Requests), Indirect Input: Load Drivers
		Access Control [?, 62]	Improper Access Control, Local Access
	N	Packet	IPV6 Packet, UDP Datagram, IPSec Packets, TCP Segments, TCP Reset (RST) Packet, Network Time Protocol Packets
		Port	
		Protocol [?, 36, 62]	SMB File Transfer, Video Codec, Border Gateway Protocol (BGP), APDU Command Response, Protocol Fragments
		Socket [43, 60, 61]	TCP [43, 60, 61], UDP [43, 60], RPC endpoint, named pipe [?, 61]
		Access Control	Local Access, Remote Access
Mechanisms	**C**	UI	Web-Based: Using Unsafe Techniques (CSS Filters, RSS Feeds [36], Active Content [36], Web Widget)
		Methods / Code Fragments	Using third-party Library, Serialization/Deserialization, Polymorphic deserialization, Reflection, Error Handling, Race Condition, Target Function/Code Fragments Controlled by Input, Open File in High Privilege Mode, Improper Security Check (Improper Input Validation [36], Weak Encryption, Load Multiple, Certificates together, Number of Request), Authentication (Authentication Method, Authentication Implementation (Location, Insecure Implementation))
		Development Framework	Improper Configuration (Creating Debug binary, Encryption), Installing Package Installers
	P	Design	Architecture, Interaction (Connecting to Other Servers, Interacting with Other Applications), Unsafe Workflow
		Deployment	Configuration: (Permission/Access Level, Web-Based (Domain) [36]), Unsafe Actions: (Install Add-on (Add Plugins, Add Extension) , Enabled Dangerous Feature [43])
	S	Dangerous Services/Server	Special Servers, Unsafe Services/Server
		Install Dangerous Program	
		Improper Configuration	Permission/Access Level [?, ?, 60], Server Configuration , Security Mechanism, Enable Specific Feature, Allow Location Access
	N	Accessible Private Network	
		Unsafe Channel/ProtocoL [36]	
Targets	**C**	UI	Web-Based: Validators, HTML/Webscript
		Methods / Code Fragments	Handler (Exception Handlers, OS Signal Handlers, Web Request Handlers), Commands (Operating System Commands [73] , Database Commands), Memory Manipulation (Uninitialized Memory, Memory Allocation, Memory Reallocation, Memory Deallocation, Memory Copy, Loop Counting Buffer Size, Buffer Access, Pointer Manipulation (Pointer Increment/Decrement, Setting Address Variable)), Serialization/Deserialization (Serialize/Deserialize APIs, Deserializing Polymorphic Class), Reflection, File manipulation (Read File, Write to File/Log), Exit Points (Output Method/API Calls [62, 73], Write to Log File, Web-Based (HTTP Response, Assert, Download)), Dangerous operations (Type Casting, Integer Operations (Type Casting, Arithmetic Operations), Encoding/Decoding, API calls from third-party Library, Resource Allocation (Socket, Thread, Database Connection), Listening to a Port, Web-Based (Dynamic Code Execution, Dynamic Code Inclusion , Redirect), Recursive Function), Critical Section, Special Objects/Components (General (Gadget Classes, Cryptographic Objects, Clonable Class Contains Sensitive Information, Serializable Class Contains Sensitive Information, Regular Expression, Path), Web-Based (Objects in DOM))
	P	Resource	Memory: Stack Memory, Heap Memory, Cache, Shared Memory [43], Collaborating Application Resources, Web-Based (Web Browser Cache, Session)
		Data	Data Resource: Database , File (Configuration File, Lock File, Log File, Inc File, CSV, Certificate, Temporary File, Backup File, Keystore File, Web-Based (Cookie [36, 62], Error Page, Authentication Token)) Sensitive Information: Credentials (User , Used Service, Database), Application Configuration Data, Encryption Keys
	S	Operating System	Availability, Data (Data Resource (System Database [43, 62], File/Directory (Critical Directory (Contain Sensitive Information, Shared Directory), SymLink/Shortcut [60], File Systems' Specific File, System/Server Critical Files), Sensitive Information (User Account, Process Information, Connection Pool)), Services/Server
		Firmware	Data: Device Information, Device: (Access, Availability)
	N	Packet	
		Network Device OS	Process, Device Settings, Device Data
		Socket Buffer	

Table 5.1: Comparison of the concepts in the proposed attack surface model with the literature

Category	Level	N	NL	PL
Entry Points	C	30	3	10
	P	42	3	7.14
	S	10	1	10
	N	20	4	20
Targets	C	59	2	3.4
	P	32	2	6.2
	S	23	2	8.7
	N	6	0	0
Mechanisms	C	30	3	10
	P	15	3	20
	S	10	1	10
	N	2	1	50

Table 5.2: Quantitative comparison of the concepts in the proposed attack surface model with the literature. N shows the number of concepts identified in the model. NL and PL represent the number and percentage of concepts covered in literature, respectively

considered I/O methods as *Entry Points* [62, 73] at the source level, however, we find that different *Handlers* and *Indirect Entry Points* can also be part of an attack surface. Nuthan and Meeneely [73] defined *System Calls* as *Dangerous Points*, however, we identify additional concepts as *Dangerous Points* such as *Type Casting*, *Integer Operations*, and *Encryption/Decryption*. We also define other code fragments such as *Serialization/Deserialization*, *Reflection*, *etc.*that can be target of attacks. Heumann et al. [7] define *URL Parameters* and Hidden Fields as input vectors, however, some other important input vectors such as *Post Request Parameter*, *HTTP Header*, and *Certificate* identified in our GT analysis are missed.

We compare the concepts defined by our GT analysis with the concepts defined in the literature. The comparison results are shown in Table 5.2. The results indicate that the literature covers a small percentage of *Code* level entry points, targets, and mechanisms, *i.e.*, 10%, 3.4%, and 10%, respectively. On average, at the *Code* level only 8 of the 119 concepts (6.7%) are covered by the literature. *Network* and *Program* level mechanisms, and *Network* level entry points are major categories that are covered in the literature with 50%, 20%, and 20%, respectively. In summary, the model proposed by this paper covers previously studied attack surface components and introduces 254 new concrete components that did not exist int the literature ($\sum_i (N_i - NL_i)$ in Table 5.2).

Attack Surface Component	No. of APIs
Input_Stream_File	525
Execute_OS_Command	51
Execute_SQL_Command	21
User_Input	27
Serialization_Deserialization	75
Read_Socket	11
OS_Signal_Handler	43
Weak_Encryption	44
Reflection	58

Table 5.3: Number of Java APIs related to each attack surface components.

5.2 Attack Surface Detection Results

5.2.1 Term weighting Approach

This section discusses the results of using the term weighting approach to automatically identify attack surface components in source code. In order to evaluate the model data was collected for nine different types of attack surface components: *Input_stream_file (entry point)* , *Execute_OS_Command (target)*, *Execute_SQL_Command (target)*, *User_input (entry point)*, *Serialization_deserialization (target)*, *Read_socket (entry point)*, and *OS_Signal_Handlers (entry point)*, *Weak-Encryption (mechanism)*, and *Reflection (mechanism)*. We collected two types of data: 1) Standard Java APIs. 2) Sample Methods related to each attack surface component from GitHub. Table 5.2.1 shows the number of java APIs related to each of the attack surface components which are used in training phase. We collected some sample files from Github repository that contain these nine attack surface components and some files that does not contain any attack surface component. We extracted 463 methods from these files to evaluate the models at method level. Number of methods associated with each attack surface components and number of methods that does not contain any attack surface component are shown in Table 5.4.

Table 5.5 shows the indicator terms identified for each attack surface components. The indicator terms identified for each attack surface components show that they are highly associated with these components. For instance, *read*, *input*, *stream*, *istream*, etc. are the terms identified for the *Input_Stream_File* attack surface component.

Attack Surface Component	**No. of Methods**
Input_Stream_File	36
Execute_OS_Command	28
Execute_SQL_Command	53
User_Input	25
Serialization_Deserialization	22
Read_Socket	22
OS_Signal_Handler	32
Weak_Encryption	38
Reflection	58
None	210

Table 5.4: Number of Java Methods collected from GitHub related to each attack surface components.

Attack Surface Component	**Indicator Terms**
Input_Stream_File	read, input, stream, istream, helper, byte, unmarshalled, initalizes, data, holder, int, static, len, next, value, seq, image, idl, sqlinput, unsigned, file, short, conceptually, fully, array, contract, b, scanner, field, void, token, scan, object, reader, general, long, concatenate, bytes, accord, programming, see, length, access, java, store, return
Execute_OS_Command	task, future, runnable, executor, execution, submit, submits, fork, join, executes, execute, pool, run, service, gt, lt, cancel, running, stop
Execute_SQL_Command	executes, statement, sql, execute, retrieval, signal, ddl
User_Input	scanner, token, scan, console, next, passphrase, input, radix, fmt
Serialization_Deserialization	unmarshal, jaxb, xml, marshal, unmarshaller, tree, content, object, deserialize, jaxbelement, result, pull, xmlstructure, unmarshals, data, impl, abstract
Read_Socket	dsts, channel, socket, read
OS_Signal_Handlers	interrupt, interruptibly, acquire, await, acquires, thread, interrupted, elapses, timeout, lock, signal, wait, condition
Weak_Encryption	base64, cipher, digest, encode, encodes, scheme, byte, final, opmode, encrypts, decrypt

Table 5.5: Indicator terms identified for attack surface components in training phase.

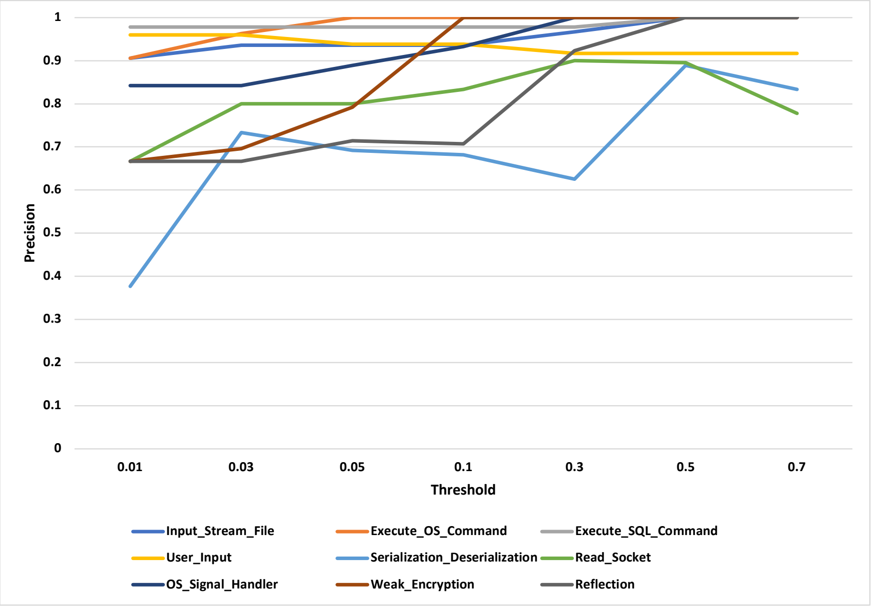


Figure 5.4: Precision of classifier based on different thresholds

The indicator terms are used to classify the methods considered as test data for each attack surface component. We run the classifier for each attack surface component separately as a binary classifier. First, the probability score $Pr_c(m)$ which is defined in section 4.2.1 is calculated for each method collected as test data. Second, the probability scores are normalized between 0 and 1. Third, the methods are classified as related or non-related using different thresholds (*0.01*, *0.02*, *0.05*, *0.1*, *0.3*, *0.5*, and *0.7*). We also considered small thresholds because small number of standard java APIs that are associated with each attack surface component in comparison to large number of total standard java APIs (38,568) may cause having small probability scores $Pr_c(m)$. The performance measures are shown in Figures 5.4, 5.5, and 5.6. Figure 5.4 shows that for all attack surface components except *Serialization_Deserialization* and *Reflection*, precision is higher than 0.80% in thresholds 0.05 and increases in all cases as threshold increases. After threshold 0.5 precision decreases. As shown in the figure 5.5 at threshold 0.03 recall is above 90% for attack surface components. By increase in threshold Recall decreases for all attack surface components. This shows that higher thresholds can not cover most of the methods which are associated to attack surface components. The results of Fscore in figure 5.6 shows that by threshold 0.02 the classifier can reach the Fscore of higher than 0.80 for all attack surface components. For *Input_Stream_File*, *Execute_SQL_Command*, *Execute_OS_Command*, *User_Input*, and *Read_Socket* Fscore is in order 0.943, 0.912, 0.957, 0.96, and 0.952%. The experimental results show that the model can detect different attack surface components with average Fscore of 0.90%.

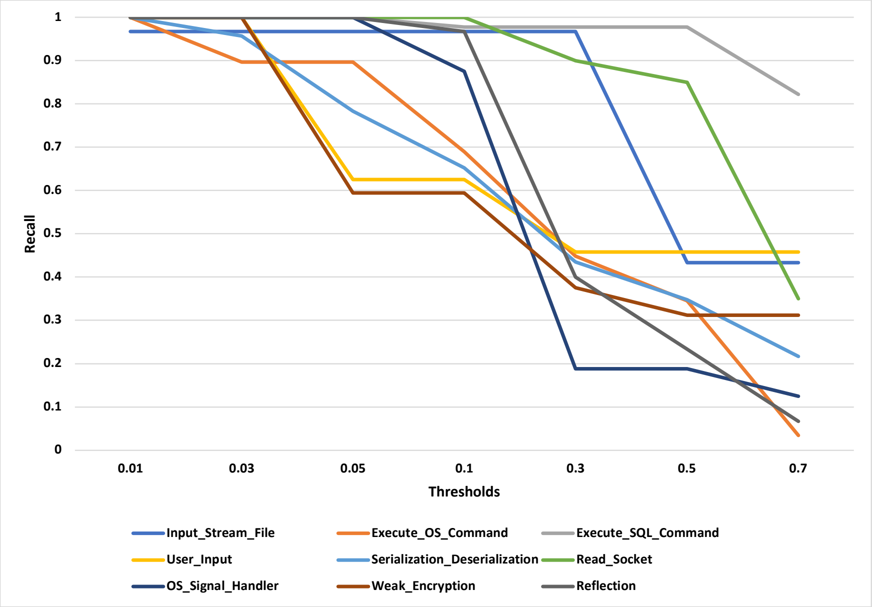


Figure 5.5: Recall of classifier based on different thresholds

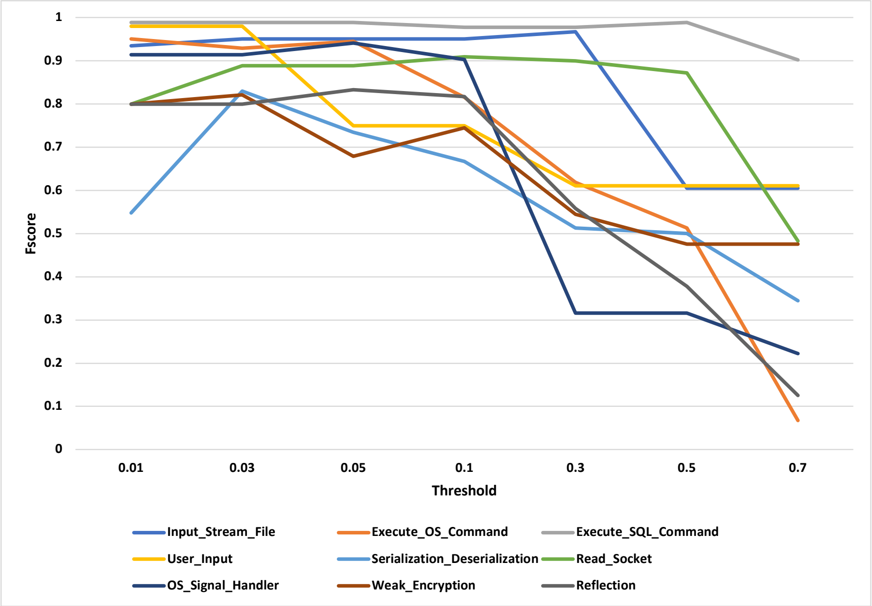


Figure 5.6: Fscore of classifier based on different thresholds

Attack Surface Component	**Precision**	**Recall**	**Fscore**
Input_Stream_File	0.9600	1.0000	0.9796
Execute_OS_Command	1.0000	0.8750	0.9333
Execute_SQL_Command	0.8621	1.0000	0.9259
User_Input	1.0000	0.9200	0.9583
Serialization_Deserialization	1.0000	1.0000	1.0000
Read_Socket	1.0000	1.0000	1.0000
OS_Signal_Handler	1.0000	0.9231	0.9600
Weak_Encryption	0.9200	0.9200	0.9200
Reflection	0.9667	1.0000	0.9830

Table 5.6: NLI Results

5.2.2 CodeBert NLI Approach

This section discusses the results of applying CodeBERT NLI approach in identifying attack surface components in methods collected from GitHub (Table 5.4). For evaluation we performed binary classification for each attack surface component at method level to answer RQ7. So, we performed nine experiments. In each experiment methods that contain related attack surface component considered as positive samples and methods that does not contain any attack surface component or contain other attack surface components as negative samples. The distribution of 80:10:10 was considered for train, validation, and test data. We considered two descriptions provided in 4.1 for data. The first description is the definition of the attack surface component and the second one is None description. We divided the positive samples related to the attack surface component into two groups. For the first group we used the description of the attack surface component as hypo and labeled them as "True" and for the second group we used the None description and labeled them as "False". We performed the same for the negative samples. We divided them into two groups. For the first group we used None description and labeled them as "True" and for the second group we used the component description as hypo and labeled them as "False". The result of the NLI approach has shown in 5.6. The results show that NLI can detect attack surface components with Fscore higher than 92% for all attack surface components. For *Execute_OS_Eommand*, *User_Input*, *Serialization_Deserialization*, *Read_Socket*, and *OS_Signal_Handler* the NLI approach could detect attack surface components with Precision of 100%. This model could detect *Input_Stream*, *Execute_SQL_Command*, *Serialization_Deserialization*, *Read_Socket*, and *Reflection* attack surface components with Recall of 100%.

Attack Surface Component	**Precision**	**Recall**	**Fscore**
Input_Stream_File	0.81	0.31	0.68
Execute_OS_Command	0.2	0.57	0.30
Execute_SQL_Command	0.83	0.96	0.89
User_Input	0.47	0.88	0.61
Serialization_Deserialization	0.67	0.91	0.77
Read_Socket	0.31	0.68	0.43
OS_Signal_Handler	0.59	0.72	0.65
Weak_Encryption	0.92	0.89	0.91
Reflection	0.18	0.97	0.30

Table 5.7: ChatGPT evaluation results

5.2.3 ChatGPT Approach

This sections shows how ChatGPT performs in identifying different attack surface components to answer RQ9. The evaluation results in table 5.7 shows that ChatGPT performance in identifying attack surface components are different. For some attack surface components such as *Execute_SQL_Command*, *Serialization_Deserialization*, and *Weak_Encryption* ChatGPT could detect attack surface components with Fscore of in order 89%, 77%, and 91% but, for some other attack surface components such as *Execute_OS_Command* and *Reflection* the fscore is 30%. The experimental results show that the ChatGPT performance is highly dependant on the attack surface component.

Figure 5.7 shows the comparison result of the Fscore for the three approaches (to answer RQ10). The comparison results shows that ChatGPT performs worse than CodeBERT NLI and term weighting approach in identifying attack surface components. For *Execute_SQL_Command* and *User_Input* attack surface components term weighting approach has higher Fscore than CodeBERT NLI approaches. For other attack surface components the CodeBERT NLI approach performs better than term weighting approach and ChatGPT. The comparison result shows that CodeBERT NLI is more stable in identifying different attack attack surface components and ,on average, performs better than term weighting approach and ChatGPT .

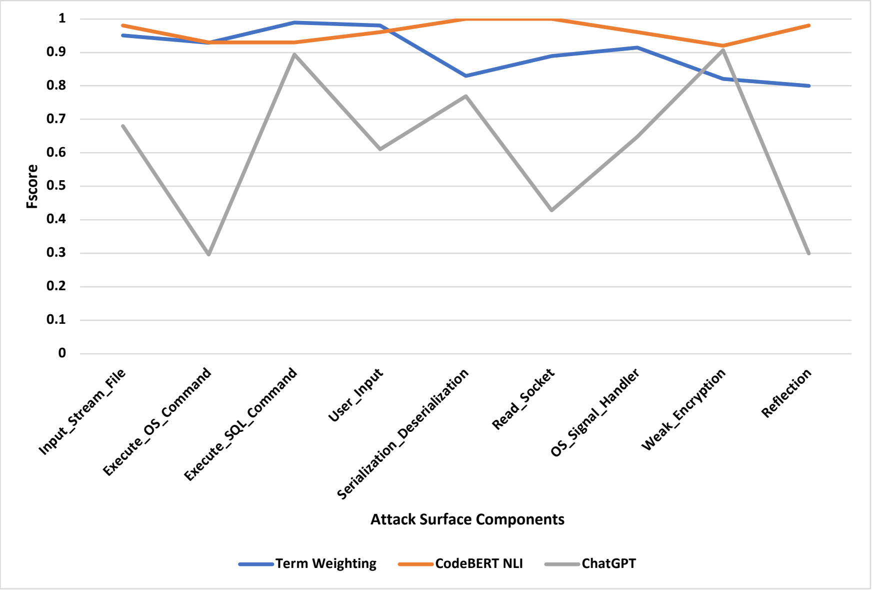


Figure 5.7: Compare Term weighting, Code_BERT NLI, and ChatGPT performance

Chapter 6

Conclusions and Future Directions

6.1 Contributions

This study focuses on characterizing and detecting software attack surface components. Getting inspired from the similarity between the attack surfaces of a house and a software system, and asking three key questions (*Where* the attacks come from, *What* they target, and *How* they emerge), this study develops a comprehensive attack surface model based on the *Entry Points* (Where), *Targets* (What), and *Mechanisms* (How). First, we leverage a grounded theory-based approach to study attack surface components of software systems. Specifically, we focus on the software *Entry Points* , *Targets*, and *Mechanisms* to define the attack surface components in our model. The identified attack surface components are categorized into four major categories for each of these three branches, *i.e.*, *Code*, *Program*, *System*, and *Network*. We conduct a systematic literature review to verify to what extent previous studies corroborate with our findings. Comparison results show that the proposed model covers all attack surface components defined in the literature, while prior works cover only a small portion of the concepts identified by our analysis. In the best case, the literature covers only 50% of *Network* level mechanisms, 20% of *Program* level mechanisms, and 20% of *Network* level entry points studied in this paper.

Second, we use NL models to automate the identification of attack surface components. First, we propose an approach using term weighting and probability_based classification. The evaluation results show the proposed approach can detect methods in source code which are associated with the attack surface component with Fscore higher than 0.80% for different attack surface components. Second, we propose a CodeBERT NLI approach that can detect different attack surface components with Fscore higher than 92%. Third, we evaluate the performance of ChatGPT in detecting different attack surface components. The experimental results show that ChatGPT performance is different for identifying different attack surface components and Fscore is between 30%- 90%.

The comparison results of the three studied LMs show that NLI performs better than term weighting and ChatGPT models in detecting software attack surface components. This shows that using both source code and description of the vulnerability provides better representation of data samples and helps in detecting different types of attack surface components. Besides that, since we use pre-trained CodeBERT model that was trained and tuned in large dataset and adapt it for attack surface detection problem, it can perform better than other two approaches in detecting attack surface components.

6.2 Limitations and Threats to Validity

While the aim of a GT study is to generate new theory, the verifiability of the theory can be inferred from the soundness of the research method. In this study, GT was strictly followed, each step was peer-reviewed and linked to the intermediary data to enable the reproducibility of findings. During the GT process, we implemented the triangulation concept to enhance the process validity [27]:

1. *Data triangulation*: We collected data from a diverse set of CVEs which report real vulnerabilities from a variety of domains and also CWE that describes software security weaknesses. Additionally, as shown in Figure 4.3, for each CVE we looked at three interrelated data: vulnerability description from the product advisory, patches (source code), and exploit information.
2. *Investigator triangulation*: Three authors [72] worked together and performed the same GT steps. Over a period of one year, the authors met weekly, discussed, peer-reviewed, and finalized the code, memos, and emerged concepts.

The main limitation of our study is related to the GT method itself, because the validation phase of the GT process is challenging [39]. We mitigate this challenge partially by using literature as a source of validation. We evaluate the identified concepts by conducting a systematic literature review to explore how well these concepts fit to the previously studied software attack surfaces. The GT analysis includes an extensive manual analysis process and such manual analysis can be prone to biases. To help mitigate this threat, we followed the investigator triangulation method. Another limitation of this study is that the proposed model may reflect attack surfaces from recent vulnerability exposures, because the analysis covers CVEs between 2016 and 2020. To mitigate this threat partially, we included CWEs which are not time dependent as an additional data source.

The limitation of the automating detection of attack surface components is related to dataset. There is no other dataset available for attack surface detection and the models were tested on our collected data. We tried to mitigate it to some extent by collecting different samples from different projects.

6.3 Future Work

In future, we will investigate how attack surface components can be used to show how vulnerable a system is. It can be done by measuring attack surface components in a software system and studying how it is related to reported vulnerabilities.

6.4 Publications

Title	Year	Venue
A grounded theory based approach to characterize software attack surfaces. **Moshtari, S.**, Okutan, A. ,Mirakhorli, M.	2022	ICSE
Data Type Bugs Taxonomy: Integer Overflow, Juggling, and Pointer Arithmetics in Spotlight Bojanova, I., Galhardo C.E.C. , **Moshtari, S.**	2022	STC
Input/Output Check Bugs Taxonomy – Injection in Spotlight. Bojanova, I., Galhardo C.E.C. , **Moshtari, S.**	2021	IWSF & SHIFT
Looking for Software Defects? First Find the Nonconformists. **Moshtari**, **S.**, Santos, J.C., Mirakhorli, M. and Okutan, A.	2020	SCAM
An Automated Approach to Recover the Use-case View of an Architecture. Santos, C.J., **Moshtari**, **S.** and Mirakhorli, M.	2020	*ICSA-C*

Table 6.1: Published papers

Title	Year	Venue
Automatic Attack Surface Detection	2024	USENIX

Table 6.2: The target publication

Bibliography

[1] Attack surface analysis. `https://github.com/SoftwareDesignLab/attack_surface_analysis`, 2022. (Accessed on 1/30/2022).

[2] Steve Adolph, Philippe Kruchten, and Wendy Hall. Reconciling perspectives: A grounded theory of how people manage the process of software development. *Journal of Systems and Software*, 85(6):1269–1286, 2012.

[3] Amritanshu Agrawal and Tim Menzies. Is better data better than better data miners?: on the benefits of tuning smote for defect prediction. In *Proceedings of the 40th International Conference on Software engineering*, pages 1050–1061. ACM, 2018.

[4] V. R. Basili, L. C. Briand, and A W. L. Melo. Validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.

[5] Jeanne Quint Benoliel. Grounded theory and nursing knowledge. *Qualitative health research*, 6(3):406–428, 1996.

[6] P. S. Bishnu and V. Bhattacherjee. Software fault prediction using quad tree-based k-means clustering algorithm. *IEEE Transactions on knowledge and data engineering*, 24(6):1146–1150, 2012.

[7] L. C. Briand et al. *Investigating quality factors in object-oriented designs: an industrial case study. in Proceedings of the 21st international conference on Software engineering.* ACM, 1996.

[8] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1066–1071. IEEE, 2011.

[9] G. Canfora et al. *Multi-objective Cross-Project Defect Prediction. in 2013 IEEE Sixth International Conference on Software Testing.* Verification and Validation, 2013.

[10] G. Canfora et al. *Multi-objective Cross-Project Defect Prediction. in 2013 IEEE Sixth International Conference on Software Testing.* Verification and Validation, 2013.

[11] Kathy Charmaz. ‘discovering’chronic illness: using grounded theory. *Social science & medicine*, 30(11):1161–1172, 1990.

[12] Kathy Charmaz. *Constructing grounded theory: A practical guide through qualitative analysis.* Sage Publication, 2006.

[13] Deng Chen, Yan-duo Zhang, Wei Wei, Shi-xun Wang, Ru-bing Huang, Xiao-lin Li, Bin-bin Qu, and Sheng Jiang. Efficient vulnerability detection based on an optimized rule-checking static analysis technique. *Frontiers of Information Technology & Electronic Engineering*, 18(3):332–345, 2017.

[14] Deng Chen, Yan-duo Zhang, Wei Wei, Shi-xun Wang, Ru-bing Huang, Xiao-lin Li, Bin-bin Qu, and Sheng Jiang. Efficient vulnerability detection based on an optimized rule-checking static analysis technique. *Frontiers of Information Technology & Electronic Engineering*, 18(3):332–345, 2017.

[15] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. Evaluation of chatgpt model for vulnerability detection. *arXiv preprint arXiv:2304.07232*, 2023.

[16] S. R. Chidamber and A C. F. Kemerer. metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[17] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[18] Juliet Corbin and Anselm Strauss. *Basics of qualitative research: Techniques and procedures for developing grounded theory.* Sage publications, 2014.

[19] G. Denaro and M. Pezzè. *An empirical evaluation of fault-proneness models. in Proceedings of the 24th International Conference on Software Engineering.* ACM, 2002.

[20] Yvonne Dittrich, Michael John, Janice Singer, and Bjørnar Tessem. Editorial for the special issue on qualitative software engineering research. *Information and software technology*, 49(6):531–539, 2007.

[21] Maureen Doyle and James Walden. An empirical study of the evolution of php web application security. In *2011 Third International Workshop on Security Measurements and Metrics*, pages 11–20. IEEE, 2011.

[22] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 60–71. IEEE, 2019.

[23] Line Dubé and Guy Paré. Rigor in information systems positivist case research: current practices, trends, and recommendations. *MIS quarterly*, pages 597–636, 2003.

[24] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[25] Wei Fu and Tim Menzies. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 72–83. ACM, 2017.

[26] Wei Fu and Tim Menzies. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 72–83. ACM, 2017.

[27] Patricia Fusch, Gene E Fusch, and Lawrence R Ness. Denzin's paradigm shift: Revisiting triangulation in qualitative research. *Journal of social change*, 10(1):2, 2018.

[28] Barney G Glaser. *Basics of grounded theory analysis: Emergence vs forcing.* Sociology press, 1992.

[29] Barney G Glaser and Anselm L Strauss. *Discovery of grounded theory: Strategies for qualitative research.* Routledge Publication, 2017.

[30] B.G. Glaser. *Theoretical Sensitivity: Advances in the Methodology of Grounded Theory.* Advances in the methodology of grounded theory. Sociology Press, 1978.

[31] G Glaser Barney and L Strauss Anselm. The discovery of grounded theory: strategies for qualitative research. *New York, Adline de Gruyter*, 1967.

[32] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software engineering*, 31(10):897–910, 2005.

[33] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software engineering*, 31(10):897–910, 2005.

[34] Zhimin He, Fengdi Shu, Ye Yang, Mingshu Li, and Qing Wang. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19(2):167–199, 2012.

[35] Steffen Herbold. Training data selection for cross-project defect prediction. In *Proceedings of the 9th International Conference on Predictive Models in Software Engineering*, page 6. ACM, 2013.

[36] Thomas Heumann, Jörg Keller, and Sven Türpe. Quantifying the attack surface of a web application. *Sicherheit 2010. Sicherheit, Schutz und Zuverlässigkeit*, 2010.

[37] Rashina Hoda. Socio-technical grounded theory for software engineering. *IEEE Transactions on Software Engineering*, 2021.

[38] Rashina Hoda and James Noble. Becoming agile: a grounded theory of agile transitions in practice. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 141–151. IEEE, 2017.

[39] Rashina Hoda, James Noble, and Stuart Marshall. Developing a grounded theory to explain the practices of self-organizing agile teams. *Empirical Software Engineering*, 17(6):609–639, 2012.

[40] Rashina Hoda, James Noble, and Stuart Marshall. Self-organizing roles on agile software development teams. *IEEE Transactions on Software Engineering*, 39(3):422–444, 2012.

[41] Seyedrebvar Hosseini, Burak Turhan, and Dimuthu Gunarathna. A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering*, 45(2):111–147, 2017.

[42] Michael Howard. Fending off future attacks by reducing attack surface, 2003.

[43] Michael Howard, Jon Pincus, and Jeannette M Wing. Measuring relative attack surfaces. In *Computer security in the 21st century*, pages 109–137. Springer, 2005.

[44] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. The importance of accounting for real-world labelling when predicting software vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software*

Engineering Conference and Symposium on the Foundations of Software Engineering, pages 695–705, 2019.

[45] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 6–pp. IEEE, 2006.

[46] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016.

[47] Graham Kenealy. Management research and grounded theory: A review of grounded theory building approach in organisational and management research. *The Grounded Theory Review*, 7(2):95–117, 2008.

[48] Hongzhe Li, Hyuckmin Kwon, Jonghoon Kwon, and Heejo Lee. A scalable approach for vulnerability discovery based on security patches. In *International Conference on Applications and Techniques in Information Security*, pages 109–122. Springer, 2014.

[49] Jingyue Li and Michael D Ernst. Cbcd: Cloned buggy code detector. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 310–320. IEEE, 2012.

[50] Z. Li, X. Jing, X. Zhu, H. Zhang, B. Xu, and S. Ying. On the multiple sources and privacy preservation issues for heterogeneous defect prediction. *IEEE Transactions on Software Engineering*, 45(4):391–411, April 2019.

[51] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 201–213, 2016.

[52] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *arXiv preprint arXiv:1807.06756*, 2018.

[53] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *arXiv preprint arXiv:1807.06756*, 2018.

[54] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[55] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[56] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Yang Xiang, Olivier De Vel, and Paul Montague. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Transactions on Industrial Informatics*, 14(7):3289–3297, 2018.

[57] Steve Lipner. The trustworthy computing security development lifecycle. In *20th Annual Computer Security Applications Conference*, pages 2–13. IEEE, 2004.

[58] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. Software vulnerability discovery techniques: A survey. In *2012 fourth international conference on multimedia information networking and security*, pages 152–156. IEEE, 2012.

[59] Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. The scope of chatgpt in software engineering: A thorough investigation. *arXiv preprint arXiv:2305.12138*, 2023.

[60] Pratyusa Manadhata and Jeannette M Wing. Measuring a system's attack surface. Technical report, Carnegie-Mellon Univ Pittsburgh pa School of Computer Science, 2004.

[61] Pratyusa K Manadhata and Jeannette M Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, 2010.

[62] Pratyusa K Manadhata and Jeannette M Wing. A formal model for a system's attack surface. In *Moving Target Defense*, pages 1–28. Springer, 2011.

[63] Pratyusa K Manadhata and Jeannette M Wing. A formal model for a system's attack surface. In *Moving Target Defense*, pages 1–28. Springer, 2011.

[64] MITRE. Cwe view: Weaknesses introduced during design. `https://cwe.mitre.org/data/definitions/701.html`, 2008. (Accessed on 08/14/2021).

[65] MITRE. Cwe view: Weaknesses introduced during implementation. `https://cwe.mitre.org/data/definitions/702.html`, 2008. (Accessed on 08/14/2021).

[66] MITRE. Cve-2016-10259. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=2016-10259`, 2017. (Accessed on 08/14/2021).

[67] MITRE. Cve-2019-15336. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=2019-15336`, 2019. (Accessed on 08/14/2021).

[68] MITRE. Cve-2020-5319. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=2020-5319`, 2020. (Accessed on 08/14/2021).

[69] MITRE. Common vulnerabilities and exposures. `https://cve.mitre.org`, 2021. (Accessed on 08/14/2021).

[70] MITRE. Common weakness enumeration. `https://cwe.mitre.org/index.html`, 2022. (Accessed on 08/14/2021).

[71] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. Challenges with applying vulnerability prediction models. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, pages 1–9, 2015.

[72] Sara Moshtari, Ahmet Okutan, and Mehdi Mirakhorli. A grounded theory based approach to characterize software attack surfaces. In *Proceedings of the 44th International Conference on Software Engineering*, pages 13–24, 2022.

[73] Nuthan Munaiah and Andrew Meneely. Beyond the attack surface: Assessing security risk with random walks on call graphs. In *Proceeding of the 2016 ACM Workshop on Software PROtection*, pages 3–14, 2016.

[74] N. Nagappan, T. Ball, and B. Murphy. Using historical in-process and product metrics for early estimation of software failures. in software reliability engineering. In *2006.* ISSRE'06. 17th International Symposium on IEEE, 2006.

[75] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[76] Viet Hung Nguyen and Le Minh Sang Tran. Predicting vulnerable software components with dependency graphs. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, pages 1–8, 2010.

[77] NIST. National vulnerability database. `https://nvd.nist.gov`, 2022. (Accessed on 08/14/2021).

[78] NVD. Statistics Results. `https://nvd.nist.gov/vuln/search`, 2021. [Online; accessed 02-April-2021].

[79] OpenAI. Chatgpt.

[80] C Opie. Research approaches in c. opie (ed.) doing educational research (pp. 73-94), 2004.

[81] oracle. Java platform standard edition 7 api specification. `https://docs.oracle.com/javase/7/docs/api/overview-summary.html`. [Online; accessed 02-February-2023].

[82] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. *Where the bugs are. in ACM SIGSOFT Software Engineering Notes*. ACM, 2004.

[83] Rana Özakıncı and Ayça Tarhan. Early software defect prediction: A systematic map and review. *Journal of Systems and Software*, 144:216–239, 2018.

[84] Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan. Auditing buffer overflow vulnerabilities using hybrid static–dynamic analysis. *IET Software*, 10(2):54–61, 2016.

[85] Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan. Auditing buffer overflow vulnerabilities using hybrid static–dynamic analysis. *IET Software*, 10(2):54–61, 2016.

[86] Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. Cross-project defect prediction models: L'union fait la force. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 164–173. IEEE, 2014.

[87] M. Park and E. Hong. Software fault prediction model using clustering algorithms determining the number of clusters automatically. *International Journal of Software Engineering and Its Applications*, 8, 2014.

[88] F. Rahman, D. Posnett, and P. Devanbu. Recalling the "imprecision" of cross-project defect prediction. In North Carolina, editor, *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering ACM: Cary*, pages 1–11. p, 2012.

[89] Kendra L Rieger. Discriminating among grounded theory approaches. *Nursing Inquiry*, 26(1):e12261, 2019.

[90] Duksan Ryu, Jong-In Jang, and Jongmoon Baik. A transfer cost-sensitive boosting approach for cross-project defect prediction. *Software Quality Journal*, 25(1):235–272, 2017.

[91] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on software engineering*, 37(6):772–787, 2010.

[92] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 315–317, 2008.

[93] P. Singh and S. Verma. Cross project software fault prediction at design phase. *World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 9(3):800–805, 2015.

[94] Ben Smith and Laurie Williams. Using sql hotspots in a prioritization heuristic for detecting all types of web application vulnerabilities. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 220–229. IEEE, 2011.

[95] Leonardo Sousa, Anderson Oliveira, Willian Oizumi, Simone Barbosa, Alessandro Garcia, Jaejoon Lee, Marcos Kalinowski, Rafael de Mello, Baldoino Fonseca, Roberto Oliveira, et al. Identifying design problems in the source code: A grounded theory. In *Proceedings of the 40th International Conference on Software Engineering*, pages 921–931, 2018.

[96] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering*, pages 120–131, 2016.

[97] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[98] Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. Approximating attack surfaces with stack traces. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 199–208. IEEE, 2015.

[99] Christopher Theisen, Kim Herzig, Brendan Murphy, and Laurie Williams. Risk-based attack surface approximation: how much data is enough? In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 273–282. IEEE, 2017.

[100] Christopher Theisen, Nuthan Munaiah, Mahran Al-Zyoud, Jeffrey C. Carver, Andrew Meneely, and Laurie Williams. Attack surface definitions: A systematic literature review. *Information and Software Technology*, 104:94–103, 2018.

[101] Christopher Theisen and Laurie Williams. Better together: Comparing vulnerability prediction models. *Information and Software Technology*, 119:106204, 2020.

[102] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. Is chatgpt the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938*, 2023.

[103] Burak Turhan, Tim Menzies, Ayşe B Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.

[104] James Walden, Jeff Stuckman, and Riccardo Scandariato. Predicting vulnerable components: Software metrics vs text mining. In *2014 IEEE 25th international symposium on software reliability engineering*, pages 23–33. IEEE, 2014.

[105] Yaohui Wang, Dan Wang, Wenbing Zhao, and Yuan Liu. Detecting sql vulnerability attack based on the dynamic and static analysis technology. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 3, pages 604–607. IEEE, 2015.

[106] Yaohui Wang, Dan Wang, Wenbing Zhao, and Yuan Liu. Detecting sql vulnerability attack based on the dynamic and static analysis technology. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 3, pages 604–607. IEEE, 2015.

[107] Michael Waterman, James Noble, and George Allan. How much up-front? a grounded theory of agile architecture. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 347–357. IEEE, 2015.

[108] Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*, 2023.

[109] Xin Xia, David Lo, Sinno Jialin Pan, Nachiappan Nagappan, and Xinyu Wang. Hydra: Massively compositional model for cross-project defect prediction. *IEEE Transactions on software Engineering*, 42(10):977–998, 2016.

[110] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 359–368, 2012.

[111] M. Yan, Y. Fang, D. Lo, X. Xia, and X. Zhang. File-level defect prediction: Unsupervised vs. supervised models. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 344–353, Nov 2017.

[112] M. Yan, Y. Fang, D. Lo, X. Xia, and X. Zhang. File-level defect prediction: Unsupervised vs. supervised models. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 344–353, Nov 2017.

[113] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. Effort-aware just-in-time defect prediction: simple unsupervised models

could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 157–168. ACM, 2016.

[114] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proceedings of the 38th International Conference on Software Engineering*, pages 309–320. ACM, 2016.

[115] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 309–320. IEEE, 2016.

[116] He Zhang, Muhammad Ali Babar, and Paolo Tell. Identifying relevant studies in software engineering. *Information and Software Technology*, 53(6):625–637, 2011.

[117] Weiqiang Zhang, Shing-Chi Cheung, Zhenyu Chen, Yuming Zhou, and Bin Luo. File-level socio-technical congruence and its relationship with bug proneness in oss projects. *Journal of Systems and Software*, 156:21–40, 2019.

[118] Xin-hua Zhang and Zhi-jian Wang. A static analysis tool for detecting web application injection vulnerabilities for asp program. In *2010 2nd International Conference on E-business and Information System Security*, 2010.

[119] Yun Zhang, David Lo, Xin Xia, Bowen Xu, Jianling Sun, and Shanping Li. Combining software metrics and text features for vulnerable file prediction. In *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 40–49. IEEE, 2015.

[120] S. Zhong, T. M. Khoshgoftaar, and N. Seliya. Unsupervised Learning for Expert-Based Software Quality Estimation. in HASE, 2004.

[121] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.

[122] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 421–428. IEEE, 2010.