# Cross-Platform Security and Privacy Analysis of Emerging Systems

---

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

---

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Faysal Hossain Shezan

August 2023

# Abstract

Emerging systems are characterized by their diverse range of functionalities and applications, from routine smartphone usage to Internet of Things (IoT) applications. While such advancements offer remarkable benefits, they also expose users to significant security and privacy risks, necessitating rigorous research to identify and address these challenges. This dissertation focuses on the security and privacy challenges that are present in detecting untrusted applications in the rapidly evolving world of emerging cross-platform technologies. In this dissertation, we use the term *'cross-platform'* to encompass settings that involve sharing of knowledge among multiple platforms, such as, web, IoT, as well as capturing the interaction among different platforms like PHP, JavaScript, SQL, and HTML. The behavior of untrusted applications can manifest in various ways, including, but not limited to, gathering excessive user information, being riddled with vulnerabilities (such as, integer and buffer overflows), and failing to adequately safeguard user data. This thesis is motivated by two key challenges in detecting security and privacy threats in emerging technologies– *limited labeled data* and *cross-language analysis.*

*First,* **limited labeled data:** Given that emerging technologies continuously develop at an unprecedented pace, there is a lack of labeled data to study the potential security and privacy threats of emerging technologies. This limitation prevents us from leveraging existing data-driven machine learning-based detection tools. So far, these approaches have been successful in well-studied platforms individually. But those are not generalized well to the new platforms due to the diverse system implementations. For example, prior works can unfold applications asking for unnecessary access to user-sensitive data in the Android platform, but they cannot be extended to IoT applications (*e.g.,* IF-This-Then-That, SmartThings) to detect similar threats. In this dissertation, we overcome the first challenge by introducing data-driven ML-based approach where we transfer security and privacy knowledge across multiple platforms. We successfully find 329 applications from the web and IoT that request access to unnecessary user-sensitive data. Later, we experience that solely relying on ML-based techniques does not always unfold security and privacy issues in cross-platforms. Hence, we improvise the detection tool by designing ML augmented program analysis-based approach. Using this tool, we discover 59 zero-day vulnerabilities acknowledged by Google LLC. Our research findings have resulted in the publishing of 12 Common Vulnerabilities and Exposures. *Second,* **cross-language analysis:** Due to the interaction among multiple programming languages it becomes very challenging to identify security and privacy violations in many applications. In such cases, analyzing a single platform is not enough, as it does not provide a comprehensive understanding of the application, leading to numerous mispredictions of violations. In light of this, the dissertation presents an end-to-end framework that captures information flow within web applications. We use the *General Data Protection Regulation* as a case study to assess the compliance of these applications. With the help of our developed tool, we identify 381 web applications that do not comply with such regulations. Both challenges underscore the risks associated with untrusted applications in modern and emerging systems. The development of these generalized detection tools marks a significant step towards more secure and privacy-conscious use of emerging systems. Furthermore, it lays a foundation for future research in this field, facilitating the development of more robust security and privacy measures as technology evolves.

# Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

faysal hossain shezan

Faysal Hossain Shezan

This dissertation has been read and approved by the Examining Committee:

Matthew Dwyer, Committee Chair

Yuan Tian, Advisor

Yangfeng Ji, Committee Member

Yixin Sun, Committee Member

Cong Shen, Committee Member

Gang Wang, Committee Member

*To my mother, father, sisters, brother-in-law, wife, and son for their continuous support, guidance, motivation, encouragement, and love.*

# Acknowledgements

This dissertation is the result of several years of dedicated effort, involving the invaluable guidance, support, and collaboration of many exceptional individuals. I express immense gratitude to my advisor, Yuan Tian, for her guidance, advice, support, and encouragement throughout my graduate career. Yuan has trained me well in all aspects of an academic career, starting from doing research, exploring new ideas, investigating challenging problems, writing manuscripts, and mentoring students. Thanks, Yuan for showing me the paths.

I am very much grateful to my Ph.D. dissertation committee members (chronologically): Dr. Cong Shen, Dr. Gang Wang, Dr. Matthew Dwyer, Dr. Yangfeng Ji, and Dr. Yixin Sun, for their guidance and valuable suggestions that help me shape my doctoral research. I would like to thank all my wonderful research collaborators: Dr. Abhishek Bichhawat, Dr. Camille Cobb, Dr. David Evans, Dr. Gang Wang, Dr. Limin Jia, Dr. Lujo Bauer, Dr. Seongkook Heo, Dr. Yingjie Lao, Dr. Yinzhi Cao, Dr. Yu Feng. I am especially grateful to Dr. Gang Wang for his mentorship and support all these years. I have learned a lot from him. Many others also provided valuable feedback and discussions along the way; it has been a tremendous privilege working with and learning from them.

Apart from academician, I also got a chance to work with awesome people from industries: Lihao Liang, Michelangelo van Dam, Mingming Sun, Minlong Peng, Ping Li, Sebastian Porst, and Xin Wang. Their guidance helps me a lot to investigate real-world problems and develop solutions.

I would like to thank all the past and current members of the UVA security research group and Linklab. Especially, Mainuddin Ahmad Jonas, Fnu Suya, Jianfeng Chi, and Bargav Jayaraman. Thank you for the laughs and happy days in the lab. I feel fortunate to work with talented and wonderful student collaborators: David Hasani, Hang Hu, Jiamin Wang, Kaiming Cheng, McKenna McCall, Mingqing Kang, Minjun Long, Mitchell Yang, Nicholas Phair, Patrick William Thomas, Yanju Chen, Zhen Zhang, Zihao Su.

All the members of my family and friends have been a source of inspiration in my life. Living outside of my country was tough at the beginning, but slowly I got used to it with the help of my friends in Charlottesville, Virginia. I would like to thank some of my friends for their tremendous help and support: Ankur Sarker, Israt Jahan Duti, Md Aashikur Rahman Azim, Saikat Chakraborty, Tanmoy Sen, Tasneem Fatema. I would also like to give a big shout-out to all my friends from Fayetteville, Arkansas.

Thank you to my parents, for their love, continuous support, and encouragement throughout my entire life. None of my achievements would have been possible without their infinite support and steely determination to give me the best possible education. I extend my heartfelt thanks to my sisters and brother-in-law, whose continuous support, insight, humor, and relentless push for me to transcend my boundaries have been instrumental in my pursuit of advanced knowledge. I also express my appreciation to my parents-in-law for their love and encouragement.

A special thank you to my wife, Fariha Tasnim, who has been my companion through the trials and triumphs of my graduate career. I appreciate her constant presence during my most challenging times. Lastly, but certainly not least, my son, Fayyadh Hossain, my little champion - his spirited naughtiness, support, love, and affection have been my source of joy and inspiration throughout this arduous journey. His laughter has been my solace during the most demanding days, his firm affection my anchor. This achievement is not mine alone - it is ours, as a family. Together, we have reached this milestone, and for that, I am eternally grateful.

# Contents

**Abstract** **i**

**Acknowledgements** **vi**

**Contents** **vii**

List of Tables . . . x

List of Figures . . . xi

**1 Introduction** **1**

1.1 First Challenge: Limited Labeled Data . . . 3

1.2 Second Challenge: Cross-language Analysis . . . 5

1.3 Contributions . . . 6

1.4 Dissertation Road Map . . . 7

**2 Detecting Untrusted Applications Accessing Unnecessary Sensitive Data** **8**

2.1 Introduction . . . 9

2.2 A Motivating Example . . . 10

2.3 Background . . . 13

2.4 Related Work . . . 14

2.4.1 Transfer Learning . . . 14

2.4.2 Permission Based Access Control . . . 14

2.5 System Design . . . 15

2.5.1 Source Domain Selection . . . 15

2.5.2 Source Model Training . . . 17

2.5.3 Data Selection . . . 18

2.5.4 Building Target Domain Model . . . 19

2.6 Implementation . . . 20

2.6.1 Datasets . . . 20

2.6.2 Models and Hyperparameters . . . 23

2.7 Evaluation . . . 25

2.7.1 Evaluation Questions and Metrics . . . 25

2.7.2 Overall performance of TKPerm . . . 26

2.7.3 Source domain selection module performance . . . 27

2.7.4 Data selection module performance . . . 29

2.7.5 Computation Overhead . . . 29

2.7.6 Factors impacting transfer learning . . . 30

2.8 Case Study . . . 32

2.8.1 IFTTT . . . 32

2.8.2 Chrome Extension . . . 33

2.8.3 SmartThings . . . 33

2.9 Summary . . . 34

**3 Identifying Vulnerable Code in Large-Scale Software by Combining Machine Learning with Symbolic Execution 35**
3.1 Introduction . . . 35
3.2 Motivating Example . . . 37
3.3 System Design . . . 39
3.3.1 System Architecture of VulDetector . . . 39
3.3.2 Slice Generator . . . 40
3.3.3 Normalization . . . 43
3.3.4 Bidirectional Gated Recurrent Unit-based Neural Architecture . . . 46
3.3.5 Symbolic Engine . . . 49
3.4 Evaluation . . . 54
3.4.1 Data Collection . . . 54
3.4.2 Performance Evaluation of VulDetector . . . 56
3.4.3 Comparison against Baselines . . . 57
3.4.4 Ablation Study . . . 58
3.5 Zero-day Vulnerability Detection . . . 59
3.6 Discussion . . . 63
3.7 Related Work . . . 63
3.8 Summary . . . 65

**4 Checking Privacy Policy Compliance by Performing Cross-language Code Analysis 66**
4.1 Introduction . . . 67
4.2 A Motivating Example . . . 69
4.3 Related Work . . . 71
4.3.1 GDPR Analysis . . . 71
4.3.2 Program Analysis . . . 72
4.3.3 Website Privacy Analysis . . . 73
4.4 System Design . . . 73
4.4.1 Overall System Architecture . . . 73
4.4.2 CCPG Generation . . . 74
4.4.3 GDPR Compliance Checker . . . 78
4.5 Implementation and Dataset . . . 86
4.5.1 Implementation . . . 86
4.5.2 Dataset Collection . . . 88
4.6 Evaluation . . . 90
4.6.1 Evaluation Questions and Metrics . . . 90
4.6.2 RQ1: Performance Evaluation of ChkPlug . . . 90
4.6.3 RQ2: Measurement Analysis . . . 92
4.6.4 RQ3: Computational Overhead . . . 92
4.7 Case Study . . . 94
4.8 Discussion & Limitations . . . 95
4.9 Summary . . . 96

**5 Conclusion 98**

**Bibliography 100**

# List of Tables

2.1 Manually-labeled Data distribution of 20 different domains from Android, IFTTT, Chrome Extension, and SmartThings. Android is the source platform and the rest are target platforms. 22

2.2 Performance, *i.e.,* accuracy, precision, recall, and F1 score, of source models on different source domains in Android platform. . . . . 25

2.3 Detailed Performance of TKPerm in different target domains. . . . . 26

2.4 Performance improvement (based on F1 score) analysis of TKPerm compared with "No Transfer" (baseline) in 11 different target domains with the highest improvement of 33.77% using transfer learning. . . . . 27

2.5 Performance comparison of $\mathcal{H}$-divergence with Greedy Selection algorithm for selecting best source domain for transferring learning using TKPerm in IFTTT platform. . . . . 28

2.6 Performance (F1 Score) comparison among three settings- no transfer, without data selection, and data selection in three different target platforms and comparison of performance improvement compared to "No Transfer" (baseline). . . . . 29

2.7 Performance improvement analysis of fine-tune data selection process of two different techniques- with and without data selection (DS) module in Chrome extension platform. The results clearly show that DS improves the F1 score of transfer learning on all three target domains. 30

2.8 End-To-End evaluation for the overall computation cost of TKPERM. Note that the computation cost includes source model training time and 29 iterations of transfer learning. All the times are in (hh:mm:ss) format. . . . . 30

2.9 Performance comparison of two different word embeddings (one is built from source platform, while the other one is from target platform) while training the target model. . . . . 31

3.1 Normalization. . . . . 45

3.2 CVE dataset used for training the model. . . . . 55

3.3 Seven large real-world software for evaluating VulDetector. . . . . 56

3.4 Performance comparison with Baseline. . . . . 57

3.5 12 accepted CVEs discovered by VulDetector. . . . . 60

4.1 Retrieving personal data using Regex. . . . . 79

4.2 Security Functions. . . . . 80

4.3 Remote Request Functions. . . . . 81

4.4 Storing data in file using PHP functions. . . . . 82

4.5 GDPR policies covered by ChkPlug. . . . . 82

4.6 Ground truth labeled data of 200 plugins for evaluation. ✓= violation ✗ = non-violation. . . 88

4.7 RQ1: Detailed performance of ChkPlug on 200 plugins for identifying different GDPR violations. Here, TP= True Positive, FP = False Positive, TN = True Negative, and FN = False Negative. . . . . 91

4.8 Breakdown PIIs into violations based on the relations to the WP database. WP=wordpress core database, non-WP=custom database. . . . . 93

4.9 RQ2: Measurement analysis results of 2,722 plugins using ChkPlug. . . . . 93

# List of Figures

2.1 The "Oplao weather" application from Chrome Extension. The application shows a correlation of its description, *e.g.,* location weather forecast, to its request for location permission. We use this application as an example to show that knowledge can be transferred across domains. . 11
2.2 An illustration of the transfer learning idea upon a neural network model. In the figure, we transfer two particular layers, i.e., word embedding and pooling as depicted in the dashed boxes, from the source domains to the target. All the rest layers are left untouched. . . . . . 12
2.3 System overview of TKPerm. TKPerm has four major sub-procedures: domain selection, source model training, find-tune data selection, and target model training. . . . . . . . . . . . 16

3.1 A motivating example. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.2 System overview of VulDetector. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
3.3 Generating PDG diagram. $\dashrightarrow$ indicates data dependency edge and $\rightarrow$ indicates control dependency edge. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.4 K-bounded slice, where K=2. $\dashrightarrow$ indicates data dependency edge and $\rightarrow$ indicates control dependency edge. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
3.5 Bidirectional Gated Recurrent Unit architecture used in VulDetector. . . . . . . . . . . . 45
3.6 Syntax for Toy Symbolic Execution Language. . . . . . . . . . . . . . . . . . . . . . . . . 49
3.7 Major Inference Rules for Symbolic Execution. † and $\oplus$ represent unary and binary operators, respectively. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
3.8 Symbolic Merging Rules. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
3.9 Ablation study of VulDetector. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

4.1 Motivating example for detecting data deletion violation in WordPress plugin. . . . . . . . . . 68
4.2 Queries used for identifying data deletion violation. . . . . . . . . . . . . . . . . . . . . . . 71
4.3 The system architecture of ChkPlug. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
4.4 RQ3: Cumulative distribution function (CDF) of Computation overhead of ChkPlug. . . . . 94

# Chapter 1

# Introduction

Emerging systems are constantly transforming the global landscape, broadening the horizon of possibilities across diverse sectors. This evolution entails an array of devices and their corresponding applications, which span from traditional platforms like smartphones, tablets, and web browsers, to advanced platforms such as, SmartThings [188] and IF-This-Then-That (IFTTT) [82]. Due to their multifaceted functionalities, these applications and devices frequently require access to sensitive operations. These operations span a wide-spectrum, including, but not limited to, location information, contact information, calendar details, and even control over smart appliances (such as, smart door locks) to augment daily convenience and lifestyle.

While the myriad applications derived from these diverse devices and platforms offer substantial benefits to users, they concurrently introduce notable security and privacy risks. Malicious applications, for instance, can request access to extraneous user information, often masquerading as benign but behaving maliciously. Users have had to grant permissions even when applications ask for more than they need instead of deciding not to install the applications on their devices [60, 58, 163]. Even worse, users' private information gets misused or shared without their knowledge or agreement [234, 56, 6]. Web applications often improperly use or leak this private information, usually to track users for things like targeted ads and similar purposes [7]. The situation escalates when legitimate applications employ third-party libraries riddled with vulnerabilities, presenting a substantial threat to user security and privacy. While the application itself might behave legitimately, the presence of vulnerabilities in the third-party libraries renders them susceptible to attacks (*e.g.,* denial-of-service attacks).

Prior works have conducted extensive data-driven security and privacy analysis using machine learning (ML) based tools, facilitated by a large amount of data on well-studied platforms, such as Android [151, 163, 198]. However, it is challenging to model such data-driven security and privacy analysis on emerging new platforms (*e.g.,* SmartThings). Because in the new platforms, we do not have large amount of labeled data available. But those ML-based approaches require extensive labeled data to train from scratch. In such cases, one naive solution one might think of is to utilize the existing tools trained on the already available data to detect similar security and privacy threats in the new platforms. However, off-the-shelves ML models will fall short due to the heterogeneity of the platforms. For instance, the characteristics of smartphone applications differ significantly from IoT applications, as they follow diverse system designs and implementations. So, it is very difficult to develop one solution that fits all. Things get even more complicated when there happen interactions among multiple platforms. Developers introduce such complex cross-platform interactions to better serve user purposes. One of the most popular technologies that have such cross-platform interaction is the web. On the web, an individual application engages with users via HTML forms, while processing user requests in the cloud using PHP. This layered interaction further complicates the analysis of such applications due to their complex interactivity. For example, HTML$\rightarrow$JavaScript is based on document object model operations and JavaScript$\rightarrow$PHP is based on HTTP requests.

This dissertation presents work that identifies and addresses cross-platform security and privacy challenges introduced by untrusted applications on different platforms such as the web (*e.g.,* Chrome), mobile (*e.g.,* Android), and IoT (*e.g.,* IFTTT, SmartThings). Note that, we refer to all the settings as *cross-platform* which indicates the involvement of sharing of knowledge among different platforms (*e.g.,* web, IoT), and the interaction of multiple platforms (*e.g.,* PHP, JavaScript).

Taken together, the following chapters consider two significant security and privacy challenges that arise in developing security and privacy threat detection tools in emerging systems:

1. *Limited Labeled Data.* Evolvement of emerging systems is very important and useful but it also brings lots of security and privacy risks. Previous studies have presented data-driven ML-based approaches based on large amount of labeled data [163, 151, 114, 238, 100]. However, this is not the case for the new platform due to the unavailability of the labeled data. Simultaneously, it is necessary to have sufficient available data on these platforms to leverage existing works. This dissertation addresses this limited labeled data challenge from two perspectives. Firstly, we introduce a generalized data-driven ML-based approach (discussed further below) for developing a detection tool that is applicable across multiple platforms (Chapter 2). Next, we recognize that relying solely on ML to solve these problems is not sufficient always. Hence, we explore the development of a data-driven ML-augmented program analysis tool to detect security and privacy issues (Chapter 3).

2. *Cross-language Analysis.* Modern applications have become increasingly complex, incorporating multiple platforms that interact with each other to serve various purposes. Merely analyzing a single platform does not provide a comprehensive understanding of the entire analysis process. Neglecting the analysis of the complete pipeline increases the likelihood of encountering numerous mispredictions. To tackle this challenge, this dissertation introduces an end-to-end framework for capturing information flow in cross-platform interaction (Chapter 4).

Together, these two challenges account for many of the risks posed by untrusted applications in modern and emerging platforms. We discuss in detail how we overcome the first challenge in Chapter 2 and Chapter 3. In both of the works described in these chapters, we analyze individual platforms to detect security threats and privacy violations. But that is not enough. Because interaction among multiple platforms can also cause lots of security and privacy issues. Next, in Chapter 4, we discuss in detail how we analyze the interaction among multiple platforms and investigate those issues. This dissertation improves the current state-of-the-art and sets the foundation for creating better detection tools that can identify untrusted applications in cross-platform emerging systems.

## 1.1 First Challenge: Limited Labeled Data

Existing data-driven ML-based detection tools for identifying security and privacy issues are trained on large amounts of labeled data from individual platforms [151, 163, 198, 100, 114]. However, it is very difficult to leverage these tools to detect similar issues in the new emerging technologies. Because directly applying those tools will not work due to diverse system implementation. At the same time, starting to train those models from scratch will not be possible due to limited labeled data. It is very expensive and requires lots of human effort to have separate detection tools for each new platform. To address the challenge of limited labeled data, we investigate this problem from two perspectives. First, we introduce a generalized data-driven solution that requires less human effort to identify the security and privacy issues in the new platform. However, it is not feasible all the time to solely depend on the ML model to solve those issues. Hence, we develop ML augmented program analysis-based approach to building a generalized solution that identifies those issues.

For the first case, we have taken a permission-based access control system as an example to build a generalized tool that identifies applications requesting access to more sensitive data than actually required. Permission-based access control helps to protect against unauthorized access to user-sensitive data. It provides sufficient information for the user for comprehending the reasons behind the requested permissions when installing a particular application. This knowledge is often presented in a human-readable language that outlines the application's functionalities and hints at its relation to the requested permissions. For instance, an Android application, providing local weather information to users, may justify its request for *location permission* by asserting the necessity of the user's location to deliver accurate local weather updates.

Previous works (such as, WHYPER [151], AutoCog [163], and SmartAuth [198]) have proposed techniques for parsing and understanding the information provided by applications, correlating it with the requested permissions, and identifying unnecessary requests for user's private information. However, with the rapid introduction of new platforms, it becomes very difficult for those existing approaches to construct a customized correlation system for each new emerging platform. Because such correlation system will require extensive human effort, including– data labeling and parameter tuning.

As previously mentioned, depending solely on a data-driven approach does not always work. We experience this situation while building detection tools for identifying vulnerable code in untrusted applications. To that end, we explore the possibility of using ML-augmented program analysis-based detection tools to identify vulnerable code in large-scale software.

Vulnerable code poses a great threat to the security and privacy of software in real-world applications. Vulnerabilities essentially function as loopholes that, when exploited by attackers, can lead to complete system control, and manipulation of user information. Manually checking for these vulnerabilities is inefficient, time-consuming, and requires lots of effort from human experts.

To reduce the burden of human effort and make the process of identifying vulnerable codes smoother and faster, researchers started building tools to detect vulnerable codes automatically. Despite the advancements in vulnerability detection in recent years through the utilization of program analysis techniques [22, 194, 154, 10, 169], and ML techniques [114, 112, 111], accurate detection of vulnerabilities in large-scale real-world software is still challenging. Because approaches based on program analysis (such as, symbolic execution) are precise and can capture vulnerabilities with high confidence. Unfortunately, it also has the well-known *state explosion* problem and scales poorly for large projects. On the other hand, ML-based approaches scale well but their performance is limited by the quality of data. And we do not have lots of labeled vulnerable data [142].

Unlike classic ML applications (*e.g.,* NLP, computer vision, robotics, etc.), labeling data on new software for vulnerability is very expensive. It requires specific security domain knowledge and time-consuming investigations. Finally, there is a huge semantic gap between codebases from different platforms. This makes it difficult to reuse ML models trained from existing labeled software datasets to other software. For instance, the widely available *CVE* (Common Vulnerabilities and Exposures) dataset [189] mostly contains code snippets with simple data types (*e.g.,* boolean, int, string), whereas real-world projects use complex data structures, such as *Tensor* in TensorFlow.

This dissertation first considers developing a transfer-learning-based solution, TKPerm for identifying unnecessary sensitive data access in different applications. TKPerm identifies 329 such untrusted applications in web and IoT platforms which request access to sensitive data without properly justifying it in Chapter 2. Next, we introduce a hybrid solution, VulDetector combining ML and program analysis-based approach to identify vulnerable code in real-world large-scale software. Using this tool, we detect 59 zero-day vulnerabilities (acknowledged by Google LLC) and publish 12 Common Vulnerabilities and Exposures. We discuss that in detail in Chapter 3.

## 1.2 Second Challenge: Cross-language Analysis

Modern applications are growing increasingly complex regarding their system implementation, often necessitating multiple platforms to develop a single application. To conduct an accurate analysis, it is crucial to incorporate the comprehensive building logic of each of those platforms. Failing to do so would lead to inaccurate judgments. For instance, in a web application, if we solely observe a button created using HTML code that is labeled as *'delete user data'* but in reality, it does not trigger the *delete operation* in the server, analyzing only the client-side interface would yield incorrect results. Hence, to infer the complete functionalities, a thorough evaluation of the entire pipeline is necessary, encompassing the capture of user actions through the interface and extending to the server-side implementation. In order to assess the compliance of web applications, we select checking GDPR (General Data Protection Regulation) as an example, as it introduces regulations that mandate end-to-end analysis of web applications.

There are many privacy regulations in place to protect consumers by ensuring that they have access and control over their personal data about the emerging system they use. These regulations are promising, and failure to comply with them may result in legal action (such as, paying a higher fine). Among all those regulations, GDPR is the oldest one and the other regulations (such as, California Consumer Privacy Act [23], Virginia Consumer Data Protection Act [205]) are built on top of this.

However, several research studies have shown that they can validate the partial functionalities of different applications to check whether those functionalities follow GDPR or not [176, 174]. However, these only cover a specific subset of GDPR requirements (such as, checking cookie consent or involving expensive manual review [24, 186]). As web applications are built using multiple programming languages, it requires parsing and analyzing languages that differ not only in semantics but also in functionalities. For example, one may collect user information using HTML form, whereas data may be stored in the server using SQL. In between, data flows across various programming languages (such as, PHP, JavaScript).

In this dissertation, we introduce ChkPlug to build a cross-language code property graph to capture data flows across multiple programming languages. With the help of ChkPlug, we are able to identify 381 web applications that violate at least one GDPR regulation. We have discussed in detail how we develop this tool in Chapter 4.

## 1.3 Contributions

This dissertation offers significant contributions addressing the challenges related to limited labeled data and cross-language analysis, encompassing a range of emerging systems. The key contributions of this research are as follows:

**Detect unnecessary sensitive data access in new emerging platforms.** In this thesis, to address the first challenge, we develop a transfer learning-based technique, TKPerm to transfer permission knowledge between diverse platforms (including– IoT, Chrome, and SmartThings) to identify applications requesting unnecessary sensitive data access. We discuss more how we have successfully transferred that permission correlation knowledge across multiple platforms in Chapter 2.

**Identify vulnerable code in large-scale software.** We revisit the first challenge introduced in this thesis where solely depending on a data-driven approach is not enough. We combine machine learning and a symbolic engine together to build VulDetector using which we are able to detect vulnerable code in real-world large-scale software with a limited amount of data in new platforms. We discuss the detailed procedure of combining machine learning with a symbolic engine for identifying vulnerable code in Chapter 3.

**Capture information flow across multiple platforms.** To address the challenge of capturing information flow across multiple platforms, we investigate web applications and introduce a novel cross-language code property graph. In this dissertation, we design and implement ChkPlug, which helps us to capture information flow across different platforms. Using our tool, we model GDPR policy to determine the compliance of those applications. We discuss in detail the complete procedure of constructing the cross-language code property graph and how we leverage that to build ChkPlug in Chapter 4.

## 1.4 Dissertation Road Map

Chapter 2 describes our detection tool for identifying applications requesting extraneous access to sensitive data. Chapter 3 introduces our hybrid solution for detecting vulnerable code in real-world software. Chapter 4 provides details on our program analysis-based tool to check GDPR compliance in web applications. Chapter 5 outlines a summary of the dissertation and discusses the potential future work.

# Chapter 2

# Detecting Untrusted Applications Accessing Unnecessary Sensitive Data

The chapter considers the first challenge highlighted in Chapter 1: the issue of limited labeled data. It explores this challenge within the scope of knowledge transfer among diverse platforms, such as, mobile, web, and IoT. More specifically, the chapter investigates how to identify applications in the new platform that fails to justify the usage of user-sensitive data in their descriptions. We refer to such problematic applications as *overprivileged applications.* While the correlation between permission and description has gained much attention in recent years, current solutions [151, 163, 198] are restricted to a single platform for identifying overprivileged applications. This chapter discusses the characterization and measurement of knowledge transfer regarding permission-description correlation across several platforms. This way, it eliminates the need to construct a new detection system for every new emerging technology. In pursuit of this objective, we introduce a data-driven ML-based approach, TKPerm, which transfers the permission knowledge extracted from Android to other platforms (including– web and IoT).

This chapter is based on the publication in the Proceedings of Network and Distributed System Security (NDSS) Symposium (NDSS) 2020, titled "TKPERM: Cross-platform Permission Knowledge Transfer to Detect Overprivileged Third-party Applications" ([184]).

## 2.1 Introduction

Permission-based access control systems are widely followed across popular platforms like mobile (*e.g.,* Android[73], web (*e.g.,* Chrome [36]), and IoT (*e.g.,* IFTTT [82], Samsung SmartThings [175]). These control mechanisms restrict the extent to which third-party applications can access users' sensitive information. Yet, before the research discussed in this chapter, all current permission detection systems are restricted to a single platform. For instance, WHYPER [151] and AutoCog [163] only focus on Android, while SmartAuth [198] is exclusive to Samsung SmartThings. Developing a correlation system customized for each newly-developed platform would require lots of human effort, including– data labeling and parameter tuning. Furthermore, these emerging platforms often have a limited number of third-party applications, meaning they might lack the necessary data to build a robust correlation system with reasonable accuracy. So, our goal is to introduce a generalized system that would work for all the newly emerged platforms with minimum human effort.

In this chapter, drawing inspiration from previous successful implementations of transfer learning in the image recognition domain [241, 148, 50], we investigate the potential of transferring permission knowledge across various platforms. This approach could simplify the creation of a new correlation system for a distinct platform using pre-existing knowledge. Our key insight is that despite these platforms having diverse use cases, they all interact with users directly. As such, they possess certain commonalities that are transferable across platforms.

To that end, we introduce TKPerm, which transfers semantic and permission correlation knowledge from a source platform to a target platform. We construct a model based on this transferred knowledge and then incorporate this model into a permission correlation system. Our approach is two-folded. First, TKPerm transfers semantic knowledge, such as, the semantics of words, from one platform to another. The key observation here is that numerous third-party applications across these platforms serve the same purpose or have comparable descriptions. For instance, both Android and Chrome have applications for weather and proxy; likewise, birthday reminder applications are found on both IFTTT and Android platforms. Consequently, TKPerm can transfer this semantic knowledge. To capture that contextual information, we leverage word embeddings. Our intuition is that words with the same or similar meanings are proximate in the embedding space, and this similarity can be transferred across platforms.

Then, TKPerm transfers permission correlation knowledge, like the connection between specific permissions and certain descriptions, from one platform to another. The key observation here is that different platforms, although with diverse use cases, sometimes share certain permissions. For instance, the location permission is found in both Android and Chrome, while access to the calendar is required on both IFTTT and Android.

TKPerm transfers this permission knowledge, particularly the correlation with application descriptions, in the form of parameters in the neural network layers close to the input layer.

While the concept of transferring semantic and permission knowledge seems straightforward, in practice it presents significant challenges. These permission-based platforms, despite sharing some commonalities, have distinctive differences. For example, Android has two kinds of location permissions: one coarse-grained and the other fine-grained. In contrast, Chrome only has one type of location permission. Some permissions, like the Evernote trigger on IFTTT, are platform-specific and do not have equivalent on other platforms.

TKPerm tackles this problem of diverse system implementations via two steps.

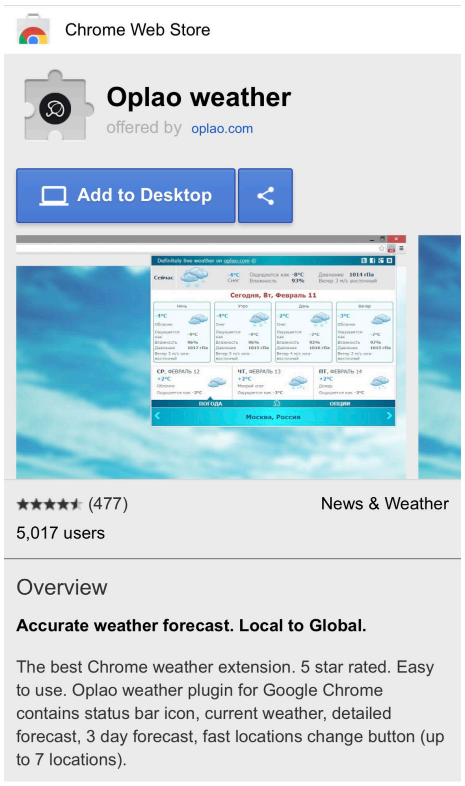
1. TKPerm fine-tunes the model transferred from the source platform using a small number of target domain data so that the subtle difference between platforms can be mitigated. Specifically, TKPerm freezes layers close to the input to preserve transferred knowledge. At the same time adjusts these layers closer to the output using fine-tuning data to make the transferred model optimized for the target platform.

2. It introduces a greedy selection algorithm to choose the source knowledge that is best suited for the target permission. Specifically, we define a domain as all the applications with a certain permission on a certain platform. TKPerm uses a greedy domain selection algorithm to choose the domains on the source platform with the highest F1 score and continue to add more domains on the source platform until the F1 score decreases.

In summary, we make several significant contributions. We are the first to introduce a general framework, TKPerm, that facilitates the transfer of knowledge between permission-based platforms. It transfers permission knowledge from Android to three distinct platforms. We manually label 36,193 sentences on Android, 4,705 on Chrome, 666 on IFTTT, and 292 on SmartThings. To facilitate future research, we open-source our labeled dataset at the following link: `https://drive.google.com/drive/folders/1Yfnz-ZpBpL8lftYIdM6JtH-QKE88NcSX?usp=sharing`. Through our measurement study, we discover 329 overprivileged applications which request excessive permissions across these three platforms.

## 2.2 A Motivating Example

Now, we illustrate our transfer learning idea with a real-world Google Chrome extension. ‘Oplao weather’, a chrome extension, is designed for accurately predicting the weather in either local or any arbitrary location



Chrome Web Store

**Oplao weather**

offered by oplao.com

Add to Desktop

(477) News & Weather

5,017 users

Overview

**Accurate weather forecast. Local to Global.**

The best Chrome weather extension. 5 star rated. Easy to use. Oplao weather plugin for Google Chrome contains status bar icon, current weather, detailed forecast, 3 day forecast, fast locations change button (up to 7 locations).

Figure 2.1: The "Oplao weather" application from Chrome Extension. The application shows a correlation of its description, *e.g.,* location weather forecast, to its request for location permission. We use this application as an example to show that knowledge can be transferred across domains.

specified by the user. Due to its functionalities of forecasting the local weather (as shown in Figure 2.1), it requests access to location permission from the user.

It is worth noting that if we do not apply transfer learning to build a permission correlation system, we need extensive human work to label extensions like this in order to build a model. The reasons are two-fold. First, the extension itself does not have any descriptions related to locations, such as, GPS and IP address, and therefore a direct correlation with the permission and extension description like what Whyper [151] does is not applicable here. Second, we cannot assume, like what AutoCog [163] does, that if many similar extensions on Chrome, like weather extensions, have access to location permission, this extension needs to do so as well. The reason is that the weather extension itself does not need access to location, but an extension forecasting local weather needs. In fact, there are some extensions on Google Chrome that require users to input a city and do not have access to the location permission. To sum up, the construction of a new permission correlation system on a different platform, like Chrome Extension, is challenging and needs human work in labeling.

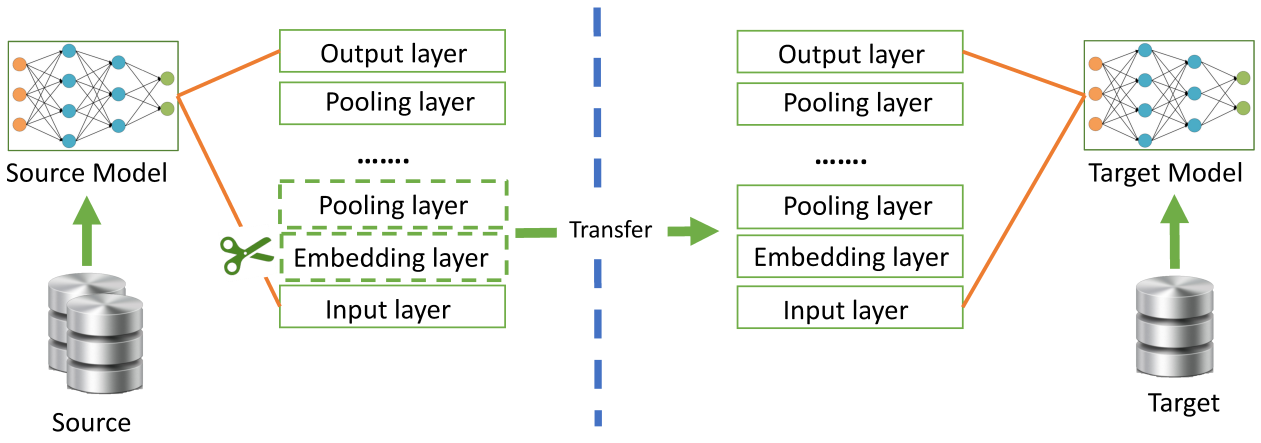


Figure 2.2: An illustration of the transfer learning idea upon a neural network model. In the figure, we transfer two particular layers, i.e., word embedding and pooling as depicted in the dashed boxes, from the source domains to the target. All the rest layers are left untouched.

We now illustrate the process of transferring the knowledge from the Android platform to Google Chrome and build a permission correlation system that links the extension's description with the location permission. There are two major types of transferred knowledge: semantics and permission. First, the extension is a weather forecast, which also exists on the Android platform. When our tool transfers knowledge, it transfers the semantics, such as, the word "weather" related and close to "forecast" and "city" close to "local", in the word embedding space from Android to Google Chrome. Second, our tool transfers permission knowledge, such as, the phrase "local weather" correlated to the location permission across the platform. For example, there exists an application on the Android platform, called "Garmisch-Partenkirchen", which is a tour guide of the German city. The application also mentions "local weather" and has access to the location permission.

One important task during transfers is to select corresponding domains, i.e., applications with certain permissions, on the source for the target so as to maximize the transferred knowledge. For this specific transfer, i.e., from Android to the location permission on the Chrome extension, our tool selects three domains on the Android, which are "fine location", "coarse location" and "read contact". It is natural that both fine and coarse locations on Android are close to the target location permission on the Chrome extension. The "read contact" permission can help as well because many applications on Android with this permission are related to event scheduling, which needs somewhat location information as well.

We now move towards background information in Section 2.3.

## 2.3 Background

In this section, we give a brief introduction to transfer learning and some terminologies and definitions related to transfer learning and used in TKPerm. Generally speaking, Transfer Learning (TL), as illustrated in Figure 2.2, is a type of Machine Learning technique that transfers knowledge from one party, called *source*, to another, called *target*, so as to improve the performance of the target. The most widely-used scenario is to transfer knowledge from a source neural network to a target by copying the weights of several layers close to the input layer. Figure 2.2 shows that we transfer two neural network layers, i.e., a pooling layer and an embedding, from a source to a target. Transfer Learning, in the past, has been widely used in solving many Computer Vision and Natural Language Processing problems [79, 233, 165].

Next, we present several terminology and definitions of transfer learning that are used for TKPerm.

- *Domain.* A domain in transfer learning refers to a set of data, *e.g.,* images or sentences, with similar properties. In the context of TKPerm, a domain is defined as a set of applications requesting certain permission, particularly all the positive and negative sentences extracted from these applications and labeled by humans. For example, we consider all the sentences extracted from applications requesting location permission on the Android platform, which are labeled as either positive, i.e., related to location permission, or negative, i.e., unrelated to location, as a location permission domain on Android.
- *Domain Selection.* Domain selection in transfer learning refers to the procedure of selecting a set of domains to be transferred to the target. In the context of TKPerm, domain selection is that TKPerm selects a set of domains on the source platforms and transfers knowledge of these domains to a permission model on the target platform. For example, in our motivating example, described in Section 2.2, TKPerm transfers knowledge from three domains, i.e., “fine location”, “coarse location” and “read contact”, on the Android platform to the location domain on the Chrome Extension platform.
- *Fine Tuning.* Finetuning is a procedure performed often after transfer to adjust the transferred parameters so that they can better fit in the target domain. In the context of TKPerm, it adopts a small number of data in the target domain to adjust these parameters in the copied layers and train parameters in new layers so as to better improve the performance of the transferred model.

## 2.4 Related Work

In the following, we first describe previous studies of applying transfer learning in the natural language processing and computer vision fields. And then, we present existing studies on permission-based access control.

### 2.4.1 Transfer Learning

In Computer Vision, transfer learning is widely used from general to task-specific cases [230]. In addition, most of the works achieve good performance over the target model just by transferring the first few layers of the source model [124, 181]. Recently, researchers working with language modeling, sentiment analysis, and question-answer modeling are adopting transfer learning in NLP domain [159, 79, 233, 165]. There are several ways to do the transferring process, including- reusing instances from source [19], and multi-task learning [95]. A commonly used approach is to pre-train embeddings that capture the contextual information which is used as different features or with the intermediate layer's input [157, 17, 49].

Following the same criteria, we consider our word embedding reusing process as part of the *Transfer Learning* methodology. In contrast to the existing transfer learning work in the natural language processing domain, none of the previous works captures the security and privacy context from the textual data. TKPerm extracts the security and privacy knowledge from one domain and helps to identify the overprivileged applications in a new target domain with small-scale fine-tuning data.

### 2.4.2 Permission Based Access Control

In mobile and IoT platforms, permission-based access system has received a lot of attention from the research community [56, 9, 139, 225, 59, 196, 58]. Rahmati et al. explored the benefit of context-specific access control in the Android platform [164]. Whereas, Acar et al. investigated the failure of the current concept of permission granting system [2]. Based on a user study on the Android platform, Backes et al. prioritized contextual integrity for the design of the permission system [216]. Researchers have also investigated overprivileged applications in IoT platforms [88] from both programming and description analysis perspective. Fernandes et al. built a static analysis tool to evaluate overprivileged applications [61].

In the access control area, the closest line of work to TKPerm is NLP-based analysis to detect overprivileged issues by analyzing descriptions. Previous works have used NLP techniques to conduct privacy & security analysis under different situations such as, mobile applications [150, 137], malware [195], privacy policy [119, 171], vulnerability in IoT applications [62, 140]. For the mobile platform, WHYPER [151] and AutoCog [163] also addressed this overprivileged problem. However, to protect the user from the overprivileged application, it becomes comparatively difficult for the IoT platform compared to the mobile platform due to the privacy implications of physical context in IoT platforms. Tian et al. redefined the overprivileged to be more user-centric and captured the corresponding problem in SmartThing platform [198].

To the best of our knowledge, most of the previous research in this area only studies one or two systems, which cannot be easily extended to other platforms. In comparison, TKPerm is a general framework for detecting overprivileged applications across different platforms, including- mobile, IoT, and web browser spaces. Thus, we need to extract common characteristics among all the overprivileged applications from all the platforms. Compared to previous works, TKPerm enables using the existing knowledge to identify the overprivileged problems in different platforms with small-scale data.

## 2.5 System Design

In this section, we introduce the overall design of TKPerm, as shown in Figure 2.3. Generally speaking, there are four sub-procedures of TKPerm: (i) Source Domain Selection, (ii) Source Model Training, (iii) Fine Tune Data Selection, and (iv) Target Model training.

Here are the details of these four sub-procedures. First, TKPerm takes all the possible domains on the source platforms as inputs (*i.e.,* circled one in Figure 2.3) and then outputs a subset of domains (*i.e.,* circled two) for transfer. Second, TKPerm trains a source domain model (circled three) based on the selected domains. Third, TKPerm selects a small number of data in the target domain using the source model (circled four) and data on the target platform (circled five) for the purpose of fine-tuning. Lastly, TKPerm builds a target model (circled eight) based on the source model (circled seven) and fine-tuning data (circled six).

In the rest of this section, we introduce these four sub-procedures separately from Subsections 2.5.1 to 2.5.4.

### 2.5.1 Source Domain Selection

TKPerm adopts a source domain selection algorithm to select the most useful source or a combination of source(s) that can help to boost the target model's performance at the target domain. While intuitively simple, the challenge of domain selection on permission-based platforms comes from the difficulty of mapping from one target domain to source domains with different permissions. That is, we follow several state-of-the-art approaches, including keyword overlapping checking, KL-divergence [105], $\mathcal{H}$-divergence [15]. Unfortunately, none of those techniques work in our problem settings. Keywords overlap checking method as domain relevancy measurement has an intrinsic drawback as KL-divergence. KL-divergence calculation is based on

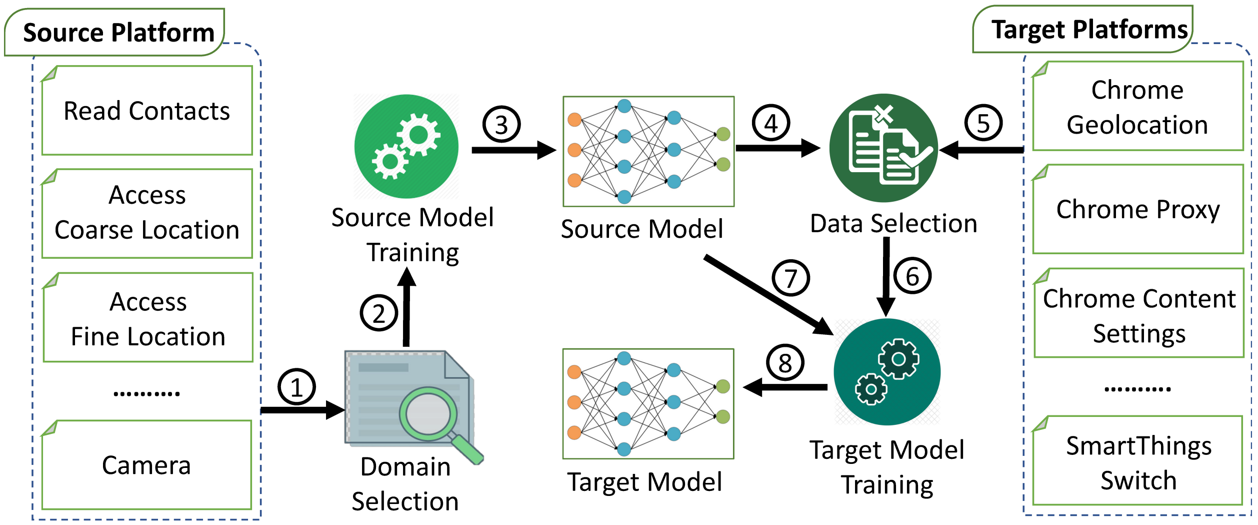


Figure 2.3: System overview of TKPERM. TKPERM has four major sub-procedures: domain selection, source model training, find-tune data selection, and target model training.

words distribution while $\mathcal{H}$-divergence takes sentences into consideration. As a result, it has the ability to capture the contextual information from the sentences. Though $\mathcal{H}$-divergence works comparatively better than others, however, it also fails to outperform our selection algorithm's performance on permission-based platforms.

Let us introduce our domain selection which is a greedy selection algorithm that identifies the best source domain(s) for Transfer Learning as listed in Algorithm 1. Here, TKPERM computes the best source(s) by evaluating the performance on the target data $d_t$. Each time, we select one target domain from the target platform. Then, TKPERM computes the performance of the source domain on the selected target domain using $computeallds_{f1}$ listed in Line 5 of Algorithm 1. It takes the list of the source domain,$[\mathcal{D}_\mathcal{S}]$, and a single target domain, $d_t$ as input. And then, TKPERM creates the set which indicates the mapping of F1 score with source domain in $[\{D_S, d_{f1}\}]$. For example, let us consider the scenario, where we want to check the overprivileged applications of "geo-location" permission from the Chrome Extension platform. TKPERM computes all the nine sources with Chrome's "geo-locaiton" permission and calculates F1 score for all of the source domains. For future use, TKPERM stores them in $[\{D_S, d_{f1}\}]$. After that, TKPERM selects the source which has the highest F1 score compared to others. TKPERM removes the selected source domain, $d_s$ from $[\{D_S, d_{f1}\}]$ and aggregates it to the aggregated source list, $\mathcal{A}_\mathcal{S}$. TKPERM computes the $\mathcal{A}_\mathcal{S}$ performance on the target domain in Line 10 of Algorithm 1. In the next round, TKPERM again identifies the next best source with the highest F1 score listed in $[\{D_S, d_{f1}\}]$ and aggregates it with the previous best source list. And thus, TKPERM constructs $\mathcal{A}_\mathcal{S}$ and removes the selected source domain from $[\{D_S, d_{f1}\}]$. TKPERM repeats the whole process for aggregating source domain in $\mathcal{A}_\mathcal{S}$ (as listed between Line 7-15 of Algortihm 1),

**Algorithm 1** Source Domain Selection using Greedy Selection Algorithm

```
    Input: Source Domain Data List, $[\mathcal{D}_\mathcal{S}]$; Target Domain Data, $d_t$
    Output: Aggregated Source List, $[\mathcal{A}_\mathcal{S}]$
 1: procedure SelectSourceDomains
 2:     $[\mathcal{A}_\mathcal{S}] \leftarrow \emptyset$
 3:     $P_{best} \leftarrow -\infty$
 4:     $P_{current} \leftarrow$ initialize to $zero$
 5:     $[\{D_S, d_{f1}\}] \leftarrow computeallds_{f1}([D_S], d_t)$
 6:     while $size([\{D_S, d_{f1}\}]) > 0$ do
 7:         $d_s \leftarrow highest_{f1}([\{D_S, d_{f1}\}])$
 8:         remove $d_s$ from $[\{D_S, d_{f1}\}]$
 9:         add $d_s$ to $[\mathcal{A}_\mathcal{S}]$
10:         $P_{current} \leftarrow computeds_{f1}([\mathcal{A}_\mathcal{S}], d_t)$
11:         if $P_{current} < P_{best}$ then
12:             remove $d_s$ from $[\mathcal{A}_\mathcal{S}]$
13:             break
14:         end if
15:         $P_{best} \leftarrow P_{current}$
16:     end while
17:     Return $[\mathcal{A}_\mathcal{S}]$
18: end procedure
```

until F1 score drops below $P_{best}$. Otherwise, TKPerm keeps adding the next best source from $[\{D_S, d_{f1}\}]$ until there is no source left in $[\{D_S, d_{f1}\}]$.

### 2.5.2 Source Model Training

TKPerm trains binary classification models, particularly, Fully Connected Neural Networks (FCNN) for target permission using selected domains on the source platform.

FCNN is a type of Neural Network where each neuron in one layer has connections with all the neurons in the previous layer except the input layer. TKPerm starts the process of building our source model by combining all the data from the Android platform to build word-embedding [157]. TKPerm then uses these word embedding vectors for building the embedding layer for all the source models. Next, TKPerm builds other layers of the neural networks, particularly, two hidden layers, which consist of 300 and 200 neurons with *ReLU* (rectified linear unit) as the activation function. Particularly, TKPerm adopts CBOW (Continuous Bag-of-Words) encoder to translate each sentence into a vector and then trains other layers of the neural networks. Note that before using the CBOW encoder, TKPerm pre-processes all the sentences by following the standard NLP practice, such as, removing Unicode characters, punctuation, stop words (*e.g.,* "a," "this"), and converting letters to lowercase [34]. We believe that this will help our model to learn useful features for the textual data.

We now discuss two design choices for our source model, which are the CBOW encoder and FCNN. First, we choose the CBOW encoder, due to the scale limitation of our dataset for sentence encoder training. Although Existing universal sentence encoders, like InferSent [40] or SkipThoughts [102], are claimed to perform generally well, they did not outperform CBOW in our experiments. Data source mismatching and higher dimensionality of their outputs can be the possible reason [149]. Second, we choose FCNN for building our model structure for source domain knowledge distilling due to the following reason. Particularly, compared with more advanced models, such as, Long Short-Term Memory, it is easier to get a better performance on a small-scale dataset using FCNN model [172].

### 2.5.3 Data Selection

TKPerm, or in general transfer learning technique, needs a small amount of labeled data in the target domain to fine-tune the transferred model to better fit in the target domain. Such data is selected by TKPerm to reduce human efforts on labeling. Specifically, TKPerm takes a source model and unlabeled target data as inputs and then ranks all the data based on the prediction value on the listed sentences inside that application. The ranking score of an application indicates the possibility of whether an application is worth labeling or not. TKPerm selects sentences from these top-ranked applications for manual labeling.

We now describe the detailed ranking algorithm in Algorithm 2 to illustrate the process of selecting fine-tune data from the target domain. The ranking process will first divide descriptions of an application into sentences, which is shown in Line 2-3 of Algorithm 2. Then, the source model starts predicting each sentence. TKPerm increases the rank of an application by following Equation 2.1.

$$\begin{gathered}\mathcal{R}_{\mathcal{A}} = \sum_{j=1}^{len(\mathcal{A})} 1 \mid \lfloor \mathcal{P}(y_j|x_j) \rceil = 1 \\ \forall\, sentence, x_j \in\, document, \mathcal{A}\end{gathered} \tag{2.1}$$

If TKPerm gets the prediction result as positive for sentences inside an application, TKPerm increments the ranking score, $\mathcal{R}_{\mathcal{A}}$ of that application by one. Next, TKPerm counts ranking score ranges from zero to $\mathcal{R}_{max}$, where $\mathcal{R}_{max}$ represents the highest number of predicted positive sentences inside an application. Thus, TKPerm builds a list, $[\mathcal{D}_{\mathcal{R}}]$ containing a set of applications and their final rank. For example, in an application where four sentences have been predicted as positive, the ranking score of that application will be four. This process is based on two intuitions: (1) selected source models have enough knowledge to distinguish data in the target domain, thus it can recognize potential positive sentences in the target domain;

**Algorithm 2** Selecting fine-tune dataset for target model using Data Selection Module.

```
    Input: Source Model, $\mathcal{M}_{\mathcal{S}}$; Unlabeled Target Domain Dataset, $[\mathcal{A}_{\sqcup}]$
    Output: Fine-tune Dataset, $[\mathcal{D}_{\mathcal{F}}]$
 1: procedure SelectFineTuneDataset
 2:     for each document, $\mathcal{A} \in [\mathcal{A}_{\sqcup}]$ do
 3:         for each sentence, $d_t \in \mathcal{A}$ do
 4:             $pred \leftarrow \text{prediction}(d_t, \mathcal{M}_{\mathcal{S}})$
 5:             if $pred = 1$ then
 6:                 $\mathcal{R}_{\mathcal{A}} \leftarrow \mathcal{R}_{\mathcal{A}} + 1$
 7:             end if
 8:         end for
 9:         add $\{\mathcal{A}, \mathcal{R}_{\mathcal{A}}\}$ to $[\mathcal{D}_{\mathcal{R}}]$
10:     end for
11:     $[\mathcal{D}_{\mathcal{R}}]^* \leftarrow sorted_{desc}([\mathcal{D}_{\mathcal{R}}])$
12:     $[\mathcal{D}_{\mathcal{F}}] \leftarrow top_{20}([\mathcal{D}_{\mathcal{R}}]^*)$
13:     Return $[\mathcal{D}_{\mathcal{F}}]$
14: end procedure
```

(2) source models are trained with many negative sentences (irrelevant to the considered permission), thus source model can successfully exclude negative sentences from target domain as well.

According to the state-of-the-art [79] transfer learning on the NLP domain, we do not need large-scale data to fine-tune the model. Small-scale data is enough to fine-tune the target model, which shows significant improvement over the target domain. After finishing the ranking process for all the applications in a domain, TKPerm sorts the applications based on their ranking score in descending order according to Equation 2.2 and thus, TKPerm gets the sorted list, $[\mathcal{D}_{\mathcal{R}}]^*$.

$$[\mathcal{D}_{\mathcal{F}}] = [sort_{desc}[\{\mathcal{A}_i, \mathcal{R}_{\mathcal{A}_i}\}_{i=1}^{m}]]_{j=1}^{n} \tag{2.2}$$

where sorting is based on the descending order of $\mathcal{R}_{\mathcal{A}_i}$, total number of applications in target domain, $m$ and the total number of considered applications, $n = 20$.

From the sorted list, TKPerm selects the top application list $[\mathcal{D}_{\mathcal{F}}]$, which has a higher rank than others, as mentioned in Line 12 of Algorithm 2. These top applications are divided into training and validation sets based on a hyperparameter.

### 2.5.4 Building Target Domain Model

We now describe how TKPerm builds the target domain model from two aspects: transferring and fine-tuning. First, TKPerm transfers two layers of the source domain model, i.e., one embedding layer and another pooling, from the source domains to the target. Second, TKPerm fine-tunes the transferred model based on the data selected in Section 2.5.3. Say, TKPerm selects data $D_T^t \cup D_T^v$ ($D_T^t$= training data, $D_T^v$= validation data) for the fine-tuning process: A human needs to label all $D_T^t \cup D_T^v$.

TKPerm then try different combinations of hyper-parameters in the target domain and select these hyper-parameters based on the performance of the target model on $D_T^v$. Specifically, these hyperparameters include the number of hidden layers, different optimizers (such as, Stochastic gradient descent, Adagrad, and Adam optimizer), dropout for regularization, learning rate, epoch, pooling type, and freezing model's lower layer. TKPerm enumerates all the different combinations of the hyperparameters and chooses the one with the best performance on $D_T^v$ for the selected target domain.

## 2.6 Implementation

In this section, we first describe all the datasets used in the implementation and evaluation of TKPerm in Section 2.6.1, and then present the models and hyperparameters built or selected by TKPerm in Section 2.6.2.

### 2.6.1 Datasets

We now describe the collection procedure, manual labeling, and then some statistics of the datasets.

#### Collection Procedure

We collect datasets from four different platforms, including Android, Chrome Extension, IFTTT, and SmartThings.

- *Android.* Android is a mobile platform that provides a range of applications available in the Play Store. We adopted the crawled data, provided by the authors of Autocog [163], in order to be consistent with prior works. This dataset contains the descriptions along with the permission information of 45,811 Android applications that were available in May 2014 in the Google Play Store.
- Chrome Extension. Chrome Extension is a platform that provides small programs integrated in Chrome browser and built-in HTML, JavaScript, and CSS to enhance the browsing experience. We collected 1,059 Chrome extension applications in November 2018. At that time, there were twelve different categories available in the Chrome store, including accessibility, blogging, by Google, news, shopping, web development, fun, photos, productivity, search tools, communication, and sports. We build a Chrome data crawler to get all the application's information. First, we extract all the IDs of Chrome extension applications by browsing webstore. Then, we made a query with the id in "chrome extension source viewer" [35]. Finally, with the help of our crawler, we were able to get the detailed information, such as, title, description, and required permissions. Then we stored that information and constructed our Chrome extension dataset.

- IFTTT. IFTTT is a platform that supports trigger-action programming (TAP) with IoT automation. We collected 259,523 IFTTT recipes in October 2017 using our crawler built with Python and beautiful soup. Here are the detailed procedures. We searched the recipes for each of the services by going through each service page in IFTTT. For each service, we collected the relative information including its description, title, trigger title, trigger service, action title, action service, and URL (i.e., link to the recipe). One recipe may appear both on the trigger service page and the action service page. For removing redundancy, we did not collect the recipe twice, and each recipe has two permissions, one for trigger service and the other for action service.
- SmartThings. SmartThings is another popular platform in the IoT domain provided by Samsung. We collected 243 SmartThings applications in August 2019. Here is how we collect them. We build a Python crawler to extract data from the official GitHub repository[1] of SmartThings. Here, we considered the capability of each application as its permission. At the time, there were 39 different capabilities, including lock, switch, motion sensor, acceleration sensor, presence sensor, and contact sensor. We select the top three capabilities based on the size of each dataset.

**Manual Labeling**

After data collection, we choose the top, common permissions, i.e., those with enough applications on each platform, and manually label sentences from descriptions of applications with these permissions. A detailed list of these top permissions is shown in the second column of Table 2.1. We select all the sentences on IFTTT, SmartThings, and Chrome extensions, i.e., our target platform, for human labeling; We only select these applications that are classified as positive by AutoCog for human labeling in the Android platform due to a large number of involved applications.

Our labeling process is as follows. We first present the official documents of our platforms and permissions to two students, who later become co-authors of the paper, as annotators and also ask them to get familiar with all the platforms under our study. We then ask them to label some datasets with our human-annotated ground truth to ensure that they understand these platforms and permissions. Next, we ask them to label whether a sentence in a particular application is related to certain permission requested by the application. If the student considers the sentence indicates the requirement of permission, they will label it as positive; otherwise, negative.

Once we have labels from two annotators, we will integrate all the labels from these two sources together. Interestingly, several sentences were vague enough to make confusion among the annotators, as a result,

[1]`https://github.com/SmartThingsCommunity/SmartThingsPublic/tree/master/smartapplications`

Table 2.1: Manually-labeled Data distribution of 20 different domains from Android, IFTTT, Chrome Extension, and SmartThings. Android is the source platform and the rest are target platforms.

| Platform | Permission | #Sent. | #Pos. Sent. | #Doc. | #Pos. Doc. |
|---|---|---|---|---|---|
| **Android** | Fine Loc. | 16,402 | 728 (4.44%) | 635 | 635 (100%) |
| | Coarse Loc. | 5,550 | 208 (3.75%) | 193 | 193 (100%) |
| | Camera | 498 | 166 (33.33%) | 11 | 11 (100%) |
| | Read Cal. | 802 | 401 (50.00%) | 16 | 16 (100%) |
| | Read Con. | 842 | 421 (50.00%) | 17 | 17 (100%) |
| | Record Au. | 366 | 183 (50.00%) | 10 | 10 (100%) |
| | Wr. Settings | 1,524 | 398 (26.12%) | 31 | 31 (100%) |
| | Send SMS | 8,398 | 407 (4.85%) | 286 | 286 (100%) |
| | Write APN | 1,811 | 92 (5.10%) | 35 | 35 (100%) |
| **IFTTT** | Evernote | 202 | 133 (65.84%) | 145 | 85 (58.6%) |
| | BMW Lab | 77 | 52 (67.53%) | 65 | 43 (66.2%) |
| | Facebook | 158 | 84 (53.16%) | 115 | 45 (39.1%) |
| | G. Cal. | 144 | 88 (61.11%) | 102 | 73 (71.6%) |
| | G. Con. | 85 | 50 (58.82%) | 49 | 43 (87.8%) |
| **Chrome Extension** | Geoloc. | 1,540 | 126 (8.18%) | 138 | 67 (48.6%) |
| | Proxy | 2,391 | 483 (20.20%) | 123 | 98 (79.7%) |
| | C. Settings | 774 | 92 (11.89%) | 58 | 28 (48.3%) |
| **Smart Things** | Lock | 34 | 10 (29.31%) | 30 | 8 (26.67%) |
| | Motion | 73 | 40 (54.79%) | 60 | 35 (58.33%) |
| | Switch | 185 | 118 (63.78%) | 153 | 111(72.55%) |

disagreement occurs. On average we have a very good consistent label between two annotators, i.e., we have an agreement rate of 97.89%, and kappa as 0.901. That is, we believe that our dataset is of high quality based on prior works [67, 204]. Furthermore, for these disagreements, we resolved the conflict issue by presenting the case to these two annotators again and asking them to reach a consensus. They reached a consensus on all the conflicts after a discussion.

We now describe a concrete example in which two annotators have a disagreement but reached a consensus after a thorough discussion. Here is the sentence: "When you have a meeting, auto create a note at Evernote", which belongs to an IFTTT recipe requiring access to Google Calendar. Two annotators have a disagreement because one thinks that this sentence has no relationship with Google Calendar, while the other thinks that a recipe can only know that you have a meeting based on access to Google Calendar. After a discussion, they agree that the latter annotator is correct as the IFTTT platform does not provide other permissions so that a recipe can know that the user has a meeting.

### Statistics

We now describe the statistics of our datasets after human labeling. In total, we labeled 36,193 sentences from 1,234 Android applications, 666 sentences from 476 IFTTT recipes, 4,705 sentences from 319 Chrome extensions and 292 sentences from 243 SmartThings applications. Table 2.1 shows the number of total and positive sentences and documents in each platform. Note that a document means all the descriptions from a certain application on a platform.

Now let us describe several observations made from our manually-labeled dataset. First, the number of positive sentences on Android and Chrome Extension is scarce when compared with negative ones. On the Android platform, only 8.3% of sentences, i.e., 3,004 out of 36,193 labeled ones, are positive, i.e., indicating the selected permission. The scenario on Chrome Extension is similar, with 13.42% positive sentences. Such a percentage is much higher on SmartThings and IFTTT, which are 49.29% and 61.29% respectively.

The reason for such a drastic difference on different platforms is the length of descriptions. Android applications usually have the longest descriptions on many functionalities that are unrelated to the app's permission request and the same applies to Chrome extensions. The descriptions of IFTTT recipes and SmartThings applications are usually short, which precisely presents the permission requirement. Therefore, the percentage of positive sentences on Android and Chrome extensions is scarce, but the one on IFTTT and SmartThings is abundant.

Second, many applications on IFTTT, SmartThings, and Chrome extensions are overprivileged as indicated in the percentage of positive documents on these platforms. Such an observation emphasizes our motivation in building a permission correlation system on these platforms. Note that all the documents on Android are positive because we only select applications that are labeled as positive by AutoCog, a prior permission correlations system.

Lastly, although the percentage of positive sentences on Android is small, the absolute number is still large. The reason is that the Android platform is with much more applications and users as compared with others. That is also the major reason that we adopt the Android platform as the source platform for TKPerm.

### 2.6.2 Models and Hyperparameters

In this section, we describe our source domain model that is used for transfer and also several hyperparameters. Note that we used the Amazon Elastic Compute Cloud (EC2) resources to run all of our experiments. The instance we used is called 'p3.2xlarge' with one NVIDIA Tesla V100 GPU, 16 Gibibyte GPU memory, 8 virtual central processing units (vCPUS), and 61 Gibibyte Main Memory. The operating system of this instance is the 'Deep Learning Amazon Linux Version 23.0'.

#### Source Domain Model

Our source domain model is built on our manually labeled sentences on the Android domain. One major challenge comes from the scarcity of positive sentences on the Android platform: if we randomly select data from all sentences for training, tuning, and evaluation, there is a chance of the source domain becoming fully biased to negative sentences [135, 14]. Therefore, we need to fix the positive and negative data ratio and train the source domain model. With a fixed ratio, we randomly sampled 70% data to train the model, 10% to tune the hyperparameters, and the rest 20% data to evaluate the model's performance. Finally, our results show that a 1:1 ratio will produce the best results on our evaluation set.

Table 2.2 shows the performance, including Accuracy, Precision, Recall, and F1 score, of the source model on each of the nine different source domains. The average source model performance in terms of F1 score is 84.2%. We calculate all the numbers based on the following criteria. The source domain model first extracts sentences from each of the documents and starts classification on those sentence data. After that, the model predicts whether an individual sentence belongs to a positive class (*i.e.,* 1) or a negative class (i.e. 0). We will consider the document, i.e., an application, as positive, if one of each sentence is considered as positive by the source domain model. If none of a document's sentences are positive, we will consider it as negative.

One interesting observation for our source domain model is that permissions with straightforward descriptions are the highest F1 score. For example, the F1 score for "Send SMS" is as high as 97%, because the descriptions of this permission usually involve a direct description of SMS. On the contrary, the F1 score for "Coarse Location" is the lowest, because the request for "Coarse Location" is often vague with the request for "Fine Location". Additionally, there are many ways to describe the request of locations, such as, local weather as we described in the motivating example.

**Hyperparameter Selection**

Hyperparameter selection in TKPerm is automatic because TKPerm reserves 10% of the total data as a validation set so that TKPerm can select the best hyperparameter. Specifically, TKPerm considers the following hyperparameters for training the source model, which are epoch, batch size, learning rate, dropout, pooling types, word vector dimension, gradient norm limit, number of selected fine-tuning data, and number of hidden layers. Many of these hyper-parameters change according to the different sources and target models.

We now describe some hyperparameters that are consistently chosen by TKPerm. Specifically, TKPerm selects $lr = 0.01, b = 256, e = 20$, where $lr$ = learning rate, $b$ = number of batch size, and $e$ = number of epoch, as these settings produce a good source model for all the domains. TKPerm also selects the dimension of this word embedding to 300 as this dimension works comparatively better than other dimensions. During the ranking algorithm, i.e., Algorithm 2, from the sorted list, TKPerm selects top 20 application list $[\mathcal{D}_{\mathcal{F}}]$, which have the higher rank than others, as mentioned in Line 12. Among the top 20 applications, 15 of them

Table 2.2: Performance, *i.e.,* accuracy, precision, recall, and F1 score, of source models on different source domains in Android platform.

| | Performance | | | |
|---|---|---|---|---|
| **Permission** | **Acc.** | **Prec.** | **Rec.** | **F1** |
| **Fine Location** | 85% | 73% | 84% | 78% |
| **Coarse Location** | 84% | 53% | 84% | 65% |
| **Camera** | 88% | 80% | 89% | 85% |
| **Read Calendar** | 89% | 87% | 89% | 88% |
| **Read Contact** | 92% | 92% | 90% | 91% |
| **Record Audio** | 84% | 83% | 83% | 83% |
| **Write Settings** | 87% | 69% | 86% | 77% |
| **Send SMS** | 93% | 93% | 100% | 97% |
| **Write APN** | 92% | 88% | 97% | 94% |
| **Total** | 88.22% | 79.78% | 89.11% | 84.20% |

are considered in the training set, while the rest five applications are selected as validation sets for tuning hyperparameters.

## 2.7 Evaluation

In the following section, we present the evaluations of TKPerm. To check if it is an effective and efficient system for detecting overprivileged issues, we evaluate both the performance and the computation overhead of TKPerm. For the performance evaluation, we report the end-to-end performance for detecting overprivileged applications, as well as the performance of different components in TKPerm. We show that TKPerm is effective for overprivileged detections on all the three platforms we test (average F1 score is 90.02% with 988 pieces of labeled data). Besides, we demonstrate that our greedy domain selection algorithm outperforms the popular source domain selection approaches such as, $\mathcal{H}$-divergence for transfer learning. TKPerm also reduces the labeling cost by selecting potentially useful data for the fine-tuning purpose. Our experiment results indicate the performance improvement of the data selection approach than randomly picking fine-tune data. In the end, we also evaluate the performance overhead of TKPerm, showing that TKPerm is scalable for a large number of applications.

### 2.7.1 Evaluation Questions and Metrics

To evaluate the performance of TKPerm, we want to answer the following questions:

- What is the end-to-end performance of TKPerm?
- What is the performance of each component in TKPerm?
- What is the computation overhead of TKPerm?

Table 2.3: Detailed Performance of TKPerm in different target domains.

| Platform | Permission | Performance | | | |
|---|---|---|---|---|---|
| | | **Acc.** | **Prec.** | **Rec.** | **F1** |
| **IFTTT** | **Evernote** | 84.6% | 77.53 % | 89.61% | 83.13% |
| | **BMW Lab** | 94.00% | 99.99% | 90.90% | 95.24% |
| | **Facebook** | 90.00% | 78.72% | 100% | 88.09% |
| | **G. Cal.** | 88.51% | 86.96% | 98.36% | 94.30% |
| | **G. Con.** | 94.11% | 93.33% | 100% | 98.41% |
| **Chrm.** | **Geoloc.** | 89.43% | 85.96% | 90.74% | 88.29% |
| | **Proxy** | 89.81% | 89.24% | 98.80% | 93.78% |
| | **C. Settings** | 76.74% | 68.97% | 95.24% | 85.31% |
| **Smart Things** | **Lock** | 93.33% | 75.00% | 100 % | 85.71% |
| | **Motion** | 82.22% | 77.14% | 100% | 87.10% |
| | **Switch** | 91.36% | 89.38% | 100% | 94.39% |

Answering the first two questions helps to show the effectiveness of TKPerm, as well as the effectiveness of design decisions in each component. Answering the third question helps to understand the scalability of our solution. For evaluating the performance of TKPerm and its components, we use the F1 score, which is the harmonic mean of precision (the fraction of relevant instances among the retrieved instances) and recall (the fraction of relevant instances that have been retrieved over the total amount of relevant instances). Higher precision will ensure a low FP (False Positive), whereas, a higher recall will ensure a low FN (False Negative). As a result, a higher F1 score will provide both low FP and FN. That's why for evaluating the model's performance, we are selecting the F1 score as the primary measurement criterion. Models with higher F1 scores will have a good chance of identifying overprivileged applications.

### 2.7.2 Overall performance of TKPerm

As mentioned in Section 2.6.1, we collect data from three popular platforms with third-party applications (IFTTT, Chrome extension, and SmartThings). To evaluate the overall performance of TKPerm, we experiment with 11 popular and sensitive target permission domains on the three different platforms. First, we use a small portion of data from the target domain to fine-tune the target model. According to the state-of-the-art approaches, we do not need to have a large amount of fine-tuned data [79] to get good performances in the target model. We extract the fine-tuned data systematically as described in Section 2.5.3 to reduce the labeling cost and ensure the quality of fine-tuning data. Table 2.1 depicts the data distribution of all 11 different target domains from three different platforms. While tuning the hyperparameter, we validated the model's performance on the validation dataset. We select hyperparameters automatically from the list of hyperparameters as described in Section 2.5.4.

**Comparison with baseline.** To get more insight into the performance of the transfer learning, we evaluate our approach compared with 'No Transfer' technique (considered as baseline). To have a fair comparison with 'No Transfer' scenario, we use the same amount of labeled data to train the model directly. Table 2.4

demonstrates the contrast of the performance between transfer with no transfer approach. By observing the improvement ratio for all target domains, we find that TKPerm outperforms the baseline approach in every target domain. On average, it exceeds the baseline by 12.62%.

**Measurement of overprivileged applications.** We find that overprivileged application is a pervasive issue. On average, we find 32.33% of applications are overprivileged. 135 applications (28.36%) from IFTTT, 114 applications (35.73%) from Chrome Extension, and 80 applications (32.9%) from SmartThings are overprivileged.

In the following, we report the component-level performance of TKPerm. In particular, we highlight the performance of the source domain selection module and data selection module because we make unique design choices for these two components.

Table 2.4: Performance improvement (based on F1 score) analysis of TKPerm compared with "No Transfer" (baseline) in 11 different target domains with the highest improvement of 33.77% using transfer learning.

| **Platform** | **Target Domain** | **Source Domain** | **Trans.** | **No Trans.** | **Improve.** |
|---|---|---|---|---|---|
| **IFTTT** | Evernote | Coarse Location + Fine Location + Camera | 83.13% | 79.78% | 3.35% |
| | BMW Lab | Send SMS + Record Audio | 95.24% | 85.71% | 9.53% |
| | Facebook | Camera | 88.09% | 75.00% | 13.09% |
| | Google Calendar | Read Calendar + Coarse Location | 94.30% | 83.54% | 10.76% |
| | Google Contact | Read Contacts | 98.41% | 97.22% | 1.19% |
| **Chrome Extension** | Geolocation | Fine Location + Coarse Location + Read Contact | 88.29% | 62.50% | 25.79% |
| | Proxy | Send SMS + Fine Location | 93.78% | 89.69% | 4.09% |
| | Content Settings | Fine Location + Read Contact | 85.31% | 59.61% | 25.7% |
| **Smart Things** | Lock | Write Setting | 85.71% | 75.00% | 10.71% |
| | Motion Sensor | Read Contact | 87.10% | 53.33% | 33.77% |
| | Switch | Send SMS + Read Calendar | 94.39% | 90.09% | 4.3% |

## 2.7.3 Source domain selection module performance

To successfully transfer knowledge to a target domain, we need to select the best source domain(s) for each given target domain. In transfer learning, selecting the best source domain is a challenging problem [3, 222]. In addition, in our problem settings, permissions in one platform cannot be easily mapped to permissions of other platforms intuitively due to the diversity of platforms. Moreover, if the source domain contains unnecessary data (*i.e.,* inappropriate for completing the targeted task), then our target model may experience "negative transfer" [167]. TKPerm overcomes these challenges by proposing the greedy selection-based domain selection algorithm (listed in Algorithm 1).

To compare our greedy selection algorithm's performance in selecting the best source domains, we also ran experiments with the state-of-art domain selection algorithm $\mathcal{H}$-divergence algorithm [15] to find the

Table 2.5: Performance comparison of $\mathcal{H}$-divergence with Greedy Selection algorithm for selecting best source domain for transferring learning using TKPerm in IFTTT platform.

| **Target Domain** | **Src Selection** | **Src Domain(s)** | **F1** |
|---|---|---|---|
| **Evernote** | $\mathcal{H}$-divergence | Read Calendar | 75.86% |
| | Greedy Select. | Coarse Location + Fine Location + Camera | 83.13% |
| **BMW Lab** | $\mathcal{H}$-divergence | Read Contact | 92.3% |
| | Greedy Select. | Send SMS + Record Audio | 95.24% |
| **Facebook** | $\mathcal{H}$-divergence | Read Calendar | 76.09% |
| | Greedy Select. | Camera | 88.09% |
| **Google Calendar** | $\mathcal{H}$-divergence | Read Calendar | 91.30% |
| | Greedy Select. | Read Calendar + Coarse Location | 92.30% |
| **Google Contact** | $\mathcal{H}$-divergence | Read Contacts | 99.20% |
| | Greedy Select. | Read Contacts | 99.20% |

most relevant source domain. The process of using $\mathcal{H}$-divergence algorithm to select the best source domain includes a binary classification problem, where the source and target domain has two different labels (*e.g.,* source data is labeled as 1, while the target data is labeled as 0). The intuition is that if the classifier can hardly distinguish two datasets, *e.g.,* making a lot of errors during the evaluation, then these two domains are very relevant. For source-target domain combinations with bigger errors, most likely those two domains are relevant to each other.

Table 2.5 shows the performance comparison of our greedy selection algorithm with $\mathcal{H}$-divergence algorithm to find the best source domain. We can observe that the greedy selection algorithm achieves 4.59% improvement of F1 score compared to $\mathcal{H}$-divergence algorithm. Thus, we can conclude that TKPerm can extract the best source domain for transferring knowledge to the target domain.

One interesting phenomenon we observe is source domains performing well in the source platform are likely to trigger good performance of their target domains in the target platform. For example, as is shown in Table 2.2 & 2.3, "Send SMS" achieves 97% F1 score in the source domain, then the performance at its target domain "BMW Lab" is also outstanding. Our greedy selection algorithm for domain selection can also identify these kinds of source domains at an early stage. In the first round of the Greedy Selection approach, "Send SMS" achieves 93.15% F1 score which is 7.44% improvement from "No Transfer".

In addition, we check if our domain selection algorithm will cause negative transfer issues. As is shown in Table 2.3, TKPerm provides the best target model for each of the domains. In IFTTT, we achieve the best performance with 98.41% F1 score in the Google Contact domain whereas, the lowest performance is 83.13% in IFTTT Evernote domain. Interestingly, we can notice that all of the target domains achieve more than 80% F1 score; the average performance is 90.02%. Therefore, TKPerm's domain selection algorithm avoids "negative transfer" issue.

### 2.7.4 Data selection module performance

To further reduce human labeling effort, we design our data selection module to select high-quality data from the target domain. Here, by "high-quality data" we meant the data that is helpful for fine-tuning the target model. We described our data selection module in Section 2.5.3, which ranked documents from the target domain. Through our data selection process, we select the top 20 applications from the target domain. After extracting those highly ranked applications, we labeled sentences from each of the applications manually. Section 2.6.1 describes our detailed methodology of data labeling. Table 2.7 illustrates the effectiveness of the data selection module. We can observe the comparison of performance between with and without data selection techniques in the Chrome Extension platform. While experimenting without a data selection approach, we select randomly 20 fine-tuning applications from the target domain. To have a fair comparison, we keep the rest of the techniques the same and only change the data selection process.

We find that the data selection algorithm we introduce is effective for building models for target domains. For example, we show the comparisons of performance with our proposed data selection algorithm, and with the random data selection approach in Table 2.7. In the Chrome platform, we achieve significant improvement by applying our proposed data selection approach. On average, we achieve 89.13% F1 score using the data selection approach while only 84.36% F1 score without data selection approach. This indicates 4.77% improvement using data selection approach in the Chrome platform. In addition, we can observe similar improvements (0.75% on IFTTT and 4.45% on SmartThings), as is shown in Table 2.6.

Table 2.6: Performance (F1 Score) comparison among three settings- no transfer, without data selection, and data selection in three different target platforms and comparison of performance improvement compared to "No Transfer" (baseline).

| **Platform** | **Perform.** | **Configuration** | | |
|---|---|---|---|---|
| | | **No Trans.** | **W/ DS** | **With DS** |
| **IFTTT** | F1 score | 84.25% | 91.08% | 91.834% |
| | Improv. | - | 6.83% | 7.584% |
| **Chrome** | F1 score | 70.6% | 84.36% | 89.13 % |
| | Improv. | - | 13.76% | 18.53% |
| **Smart Things** | F1 score | 72.80% | 84.65% | 89.1% |
| | Improv. | - | 11.85% | 16.3% |

### 2.7.5 Computation Overhead

To evaluate the scalability of the TKPerm, we measure the computation overhead for the system. As mentioned in Section 2.6.2, we use Amazon Elastic Compute Cloud (EC2) resources and NVIDIA Tesla V100 GPU to train our model.

Table 2.7: Performance improvement analysis of fine-tune data selection process of two different techniques-with and without data selection (DS) module in Chrome extension platform. The results clearly show that DS improves the F1 score of transfer learning on all three target domains.

| **Target** | **Source** | **Perf.** | **W/DS** | **DS** |
|---|---|---|---|---|
| **Geoloc.** | Fine Location + Coarse Location + Read Contact | F1 | 83.10% | 88.29% |
| | | Improv. | - | 5.19% |
| **Proxy** | Send SMS + Fine Location | F1 | 93.61% | 93.78% |
| | | Improv. | - | 0.17% |
| **C. Sett.** | Fine Location + Read Contact | F1 | 76.36% | 85.31% |
| | | Improv. | - | 8.95% |

We run each experiment 11 times to compute the average computation overhead. First, we run the experiment on nine different source domains for any given target domain, then use the greedy algorithm we have shown in Algorithm 1 to get the best combination of the source domain. We have reached an average of 90.02% F1 score on the target domain with an average size of 94 documents.

Table 2.8 shows the performance overhead of running experiments on different permissions and platforms, which includes the information about the size of the data (source data + target data) and the time cost of TKPerm (getting the result on the best source combination).

Note that, the performance overhead of TKPerm is a one-time cost, once we train a model for one target permission, we do not need to retrain the model for new applications. Therefore, based on the data from Table 2.8, TKPerm is scalable.

Table 2.8: End-To-End evaluation for the overall computation cost of TKPERM. Note that the computation cost includes source model training time and 29 iterations of transfer learning. All the times are in (hh:mm:ss) format.

| **Platform** | **Target** | **Source** | **#Doc (target)** | **#Doc (source)** | **Time (hh:mm:ss)** |
|---|---|---|---|---|---|
| **IFTTT** | Evernote | Coarse Location + Fine Location + Camera | 145 | 839 | 33:27:03 |
| | BMW Lab | Send SMS + Record Audio | 65 | 296 | 14:08:40 |
| | Facebook | Camera | 115 | 11 | 22:57:20 |
| | Google Calendar | Read Calendar + Coarse Location | 102 | 207 | 15:15:18 |
| | Google Contact | Read Contacts | 49 | 17 | 18:40:17 |
| **Chrome Extension** | Geolocation | Fine Location + Coarse Location + Read Contact | 138 | 845 | 07:37:28 |
| | Proxy | Send SMS + Fine Location | 123 | 921 | 06:54:01 |
| | Content Settings | Fine Location + Read Contact | 58 | 652 | 09:42:45 |
| **Smart Things** | Lock | Write Setting | 30 | 31 | 03:47:59 |
| | Motion Sensor | Read Contact | 60 | 17 | 04:09:44 |
| | Switch | Send SMS + Read Calendar | 153 | 302 | 14:11:08 |

### 2.7.6 Factors impacting transfer learning

By looking into the data sets in original domains and target domains, we have the impression that the word embedding and permission knowledge might be the major knowledge that gets transferred for TKPerm. We design some experiments to control these two factors to give more insights into why transfer learning works in

TKPerm. These experiment results match our intuition. Indeed, word embedding and permission knowledge have significant impacts on the target model performance.

**Effect of word Embedding**

Table 2.9: Performance comparison of two different word embeddings (one is built from source platform, while the other one is from target platform) while training the target model.

| Platform | Perform | Word Embedding | |
|---|---|---|---|
| | | Target | Source |
| **IFTTT** | F1 score | 86% | 91.83% |
| | Improv. | - | 5.83% |
| **Chrome Extension** | F1 score | 88% | 89.13% |
| | Improv. | - | 1.13% |
| **Smart Things** | F1 score | 85% | 89.1% |
| | Improv. | - | 4.1% |

TKPerm uses the source domain's (Android) word embedding and transfers it to the target domain. As the description in Android is both rich and well structured, our source model is initialized with useful features. Transferring the pre-trained word embedding to the target domain also shows improvement over the target model.

To investigate the impact of word embedding, we ran several experiments by using word embedding built from the source platform and target platform. Our intention was to check whether the model initialization phrase is getting any help from the source data or not. Table 2.9 illustrates the performance comparison between word embedding from source data and target data. The key difference between these two settings is the selection of the platform while building the word embedding. For checking the performance of source word embedding, we directly use the embedding built from source data (i.e. Android). Whereas, to test the performance of word embedding of the target data, we use the corresponding target platform's full data to train the word embedding. An important thing to note is that training word embedding is an unsupervised approach. We do not need the labeled data for that. So, we use the large corpus data from the target platform to build the word embedding for all the target domains of that platform. For example, to experiment with "Geolocation" permission in the Chrome Extension platform, we used all the descriptions (from 1,059 applications) to train Chrome word embedding, and then we used that in the target model's embedding layer.

Results in Table 2.9 highlight the impact of the source platform's word embedding. Indeed, TKPerm gets benefits by using the source platform's word embedding with an improvement of 3.69% F1 score (average) in all three target platforms. Finally, we can conclude that we do not need to train different word embedding for different target platforms. It reduces the training cost for target platforms.

**Knowledge of Permissions**

TKPerm learns specific features from the textual data for a particular permission. Then, it maps such features to the corresponding permission. In particular, TKPerm correlates requested permission with the description and thus resulted in output indicating whether a particular application is overprivileged or not. In our investigation, we observed that several permissions (though the applicability may differ from each other) overlap with each other in the source and target platform. For example, Geolocation from Chrome, and Fine Location & Coarse Location from Android; Google Calendar from IFTTT, and Read Calendar from Android, Google Contact from IFTTT, and Read Contact from Android— all of these pair of permissions from two different platforms, descriptions of which have some common characteristics because of the similar functionality. In our experiment (as illustrated in Table 2.4), we noticed the improved performance of such correlated domains. Finally, we can conclude that permission knowledge from the source domain is also driving the performance of transfer learning.

## 2.8 Case Study

TKPerm identifies 329 overprivileged applications from all the different platforms. We have responsibly reported these applications to their developers. In the following, we show a few examples of overprivileged applications from each of the three platforms, such as, *Tomorrow's Forecast on the way home* and *YSA email automation* from IFTTT, *GamingHub* and *Private Internet Access* from Chrome Extension, *Button Controller* and *Hue Mood Lighting* from SmartThings.

### 2.8.1 IFTTT

"Tomorrow's Forecast on the way home" is an IFTTT recipe that helps a driver to get the upcoming weather forecast while driving back home from work. To install this recipe, the user needs to give access to her BMW car. Then, she will be notified about the upcoming weather forecast during a particular time of the day. They describe their recipe's functionality as "Connect the weather service before running." Just by reading the title or description, the user will not expect that this recipe would access her BMW car because the recipe does not mention anything relevant in its description and title.

"YSA email automation" is another overprivileged recipe from the IFTTT platform. This recipe accesses the sensitive Google contact data, which is not relevant to its description "YSA email automation New member email tree". After installing this recipe, whenever a user adds a new Google contact, the recipe will send out an email notification. Users would not have expected the application to access their Google contact information when they install the recipe.

### 2.8.2 Chrome Extension

"Private Internet Access" is a popular extension on the Chrome platform with more than 170,000 users. It helps to encrypt user network traffic and keeps them protected while connected to the internet. They protect the user by webRTC (real-time web communication) blocking, and sorting the gateways by latency. To install this extension in the Chrome browser, the user needs to allow the Content Settings permission, which is the ability to "Change your settings that control websites' access to features such as, cookies, JavaScript, plugins, geolocation, microphone, camera, etc." according to the official document. TKPerm detects this extension as overprivileged as its access to sensitive data like cookies, geolocation, microphone, and camera without clear justification. We have received some initial, relatively positive feedback from this extension's developers—they partially acknowledged our issue and we are still in the process of talking with them regarding a possible solution.

"GamingHub" is another popular extension on Chrome platform with over 20,000 users. It enables users quick & elegant access to some of the most popular web games to date. It does so by displaying them as quick access links on the browser's *New Tab* Page. TKPerm detects this application as overprivileged because of its access to the user's physical location without any links to its described functionality of displaying web games. It is worth noting that many other permissions, such as "Read and change all your data on the websites you visit" and "Read and change your browsing history", of the extension, are also overprivileged.

### 2.8.3 SmartThings

"Button Controller" is an application from the SmartThings platform. The functionality of this application is to "control devices with buttons like the Aeon Labs Minimote". This description reflects that it will operate like a switch that can control devices by turning them on/off. Unfortunately, there does not exist anything, neither in their title nor in the description, mentioning that they will also access the "Lock" permission. Interestingly, if a user allows all the permissions of the app, which are switch, lock, music player, and alarm, she grants access to the application in both locking and unlocking her door, which is fearsome.

"Hue Mood Lighting" is another overprivileged application from SmartThings platform. From the description listed as "Sets the colors and brightness level of your Philips Hue lights to match your mood", it is clear that this application will change the color and brightness of lights. TKPerm detects the application as overprivileged as it is requesting permissions for accessing motion sensors without any clear justification. It is worth noting that the application also requests many other unnecessary permissions, such as, contact sensor, acceleration sensor, switch, presence sensor, smoke detector, water sensor, and button, posing an even greater threat.

## 2.9 Summary

In this chapter, we present a framework, TKPerm to detect overprivileged third-party applications, which have access to sensitive user data. Our experimental results indicate that we are successfully able to transfer such permission knowledge from the Android platform to three different platforms, which are IFTTT, Chrome Extension, and Samsung SmartThings. Our transferred model can achieve an average F1 score of 90.02% with only a handful of pieces of labeled data. The transferred model by TKPerm detects 329 different overprivileged applications, such as Tomorrow's Forecast on the way home and YSA email automation from IFTTT, GamingHub and Private Internet Access from Chrome Extension, Button Controller and Hue Mood Lighting from SmartThings.

# Chapter 3

# Identifying Vulnerable Code in Large-Scale Software by Combining Machine Learning with Symbolic Execution

The first key challenge highlighted in Chapter 1, the lack of labeled data, is revisited in this chapter. This challenge is examined in the context of detecting vulnerable code in practical software applications. Although existing detection tools have made progress in identifying vulnerabilities, it is still challenging to identify vulnerable code in large, real-world software [10, 169, 114, 112, 111]. This issue is primarily attributed to the limitations of data availability and data quality. Relying purely on an ML-based approach (similar to the one developed in Chapter 2) proves ineffective here. As only a data-driven approach is not sufficient. Hence, we introduce VulDetector, a synergistic solution that combines ML with a program analysis technique. This tool enables the successful detection of vulnerable code in diverse large-scale real-world software.

## 3.1 Introduction

Vulnerable code is a major threat to the security and privacy of real-world software. They enable attackers to launch payloads that cause damages including but not limited to – attackers taking control over the entire system, leaking sensitive data, and altering user information. In 2016, the total number of vulnerabilities registered in the Common Vulnerabilities and Exposures (CVE) index was 6,454, and it increased largely to 23,382 in 2022 [207]. Finding vulnerabilities manually requires lots of domain expert efforts.

To reduce the burden of human effort and make the process faster, researchers started building tools to detect vulnerable codes automatically. However, despite the progress of vulnerability detection in recent years with program analysis techniques[22, 194, 154, 10, 169], and machine learning techniques [114, 112, 111], it is still challenging to accurately detect vulnerabilities in large-scale real-world software. First, approaches based on program analysis such as symbolic execution are precise and can capture vulnerabilities with high confidence. However, it also has the well-known *state explosion* problem and scales poorly for large projects. Static analysis mitigates the scalability problem by sacrificing precision.

Second, machine learning-based approaches scale well but their performance is limited by the quality of data. The proposed solutions usually only have reasonable performance when working on the same software database with lots of labeled data. Unfortunately, unlike classic ML applications (*e.g.,* NLP, computer vision, robotics, etc.), labeling data on new software for vulnerability is very expensive. It requires specific security domain knowledge and time-consuming investigations. Finally, there is a huge semantic gap between codebases from different domains and applications. This makes it difficult to reuse machine learning models trained from existing labeled software datasets to other software. For example, these days widely available popular vulnerability detection dataset, CVE dataset [189], mostly contains code snippets with simple data types (*e.g.,* boolean, int, string), whereas real-world projects use complex data structures, such as, Tensor in TensorFlow.

To address these challenges, we design and implement a vulnerability detection tool VulDetector to achieve the following goals: (1) Accurate: the tool should be able to accurately detect vulnerabilities with low false positive and low false negative rates. (2) Scalable: the tool should be able to analyze large-scale and complex software within a reasonable amount of time.

**Our Approach.** To achieve the above-mentioned goals, our tool VulDetector has a novel co-design of machine learning and a symbolic engine to detect vulnerable code in real-world software. The key idea is to use machine learning to detect potentially vulnerable code slices in a scalable fashion so that our symbolic engine can analyze these relatively small code snippets more precisely to find vulnerabilities. This is non-trivial and we face several challenges. First, the code for training and testing may include lots of information that is not relevant to vulnerability, which can introduce noise for the machine learning models. To address this challenge, we propose a novel *K-bounded program slicing* based on program dependency graphs (PDG) to capture *contextual program slices* that are most relevant to potential vulnerabilities. On the machine learning side, we use bidirectional gated recurrent units (BGRU) to classify vulnerable code slices.

In order not to miss crucial vulnerabilities, we adjust the hyperparameters of our model to bias toward lower false negative rates. This design choice, however, can flag benign code snippets as malicious. A natural way to mitigate false positive cases, as the second step of our approach, is to perform a more precise analysis via symbolic reasoning. Because code flagged by the BGRU model is partial and may not be compilable, we can not directly leverage off-the-shelf symbolic engines such as KLEE [22]. To address this challenge, we design a customized symbolic engine that is able to reason about partial code snippets and handle abstract domains for complex data structures widely used in open-source libraries such as tensors and vectors.

To evaluate the effectiveness of our approach, we implement the proposed ideas in a tool called VulDetector, and evaluate it on seven widely used real-world software (Android, FreeRDP, GPAC, Linux, Pillow, TCPDump, and TensorFlow) from a wide range of different domains. Through the comparison against existing approaches, namely, Devign and Vuddy, our experiment results indicate that VulDeePecker [114], Devign [238], and Vuddy [100] achieve 14%, 32.7%, and 4.43% F1-Score, respectively. Among all the baselines, Devign [238] achieves the best performance with 32.71% F1-Score among all these software. VulDetector outperforms the baseline by having a 62.86% F1-Score (average), an improvement of 30.15% F1-Score in all the seven real-world software. Next, we use our tool to find zero-day vulnerabilities in the latest version of the software. Interestingly, we are able to detect 76 confirmed zero-day vulnerabilities. And our findings result in the acceptance of 12 CVEs. We have responsibly disclosed all the vulnerabilities to the corresponding company.

## 3.2 Motivating Example

In this section, we provide an example of vulnerable code which is detected by VulDetector.

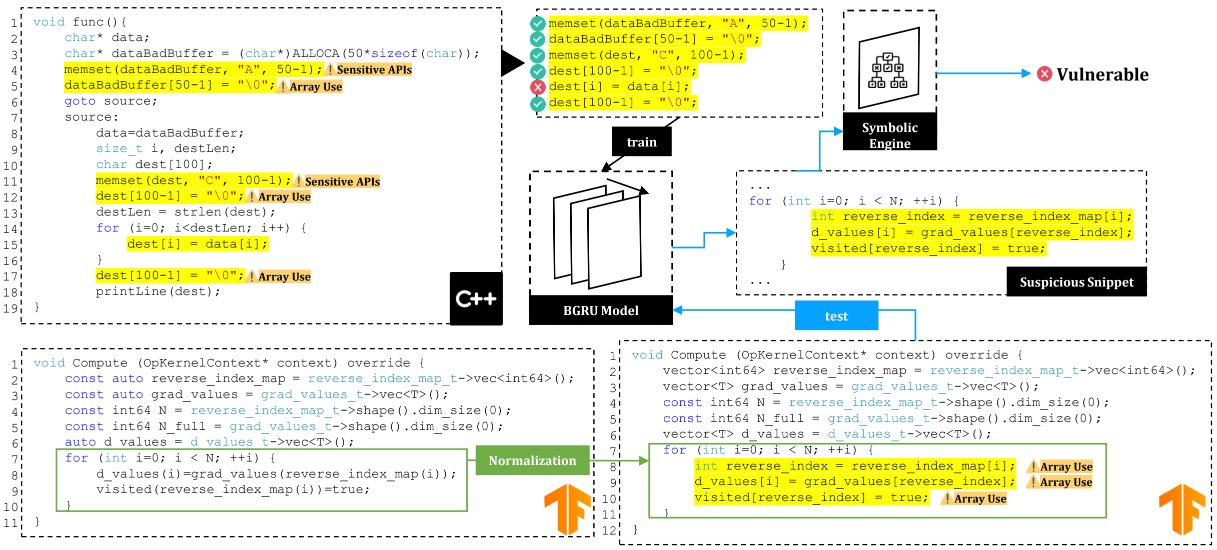
Listing 1 shows a vulnerability found in TensorFlow, a popular ML framework. This vulnerability happens due to incorrect access to the memory elements. In particular, there is a double (nested) indexing of *grad_values* at line 24 (of Listing 1). The program takes input from the user and assigns the corresponding value in *grad_values_t* in Line 6, and *reverse_index_map_t* in Line 7. As there is no input validation check on these variables, it is possible that the value of *reverse_index_map* is an index that will be outside of the bounds of *grad_values*. In such a scenario, it will cause a heap buffer overflow.

Finding this vulnerability with traditional symbolic reasoning techniques is hard due to the well-known path-explosion problem: the vulnerability lies inside `Compute` function that contains around 200 lines of code. But the function also transitively invokes other twenty functions from different directories which contain thousands of lines of C++ code in total.

An alternative way is to use machine learning, which is more scalable. However, a reliable ML model is contingent on the quality and quantity of the training data, whose labeled vulnerabilities are labor-intensive.

```
void Compute (OpKernelContext* context) override {
  const Tensor* reverse_index_map_t;
  const Tensor* grad_values_t;
  ...
  // OP_REQUIRES_OK takes user input and assigned to the variable passed as a parameter
  OP_REQUIRES_OK(context, context->input( "grad_values", &grad_values_t ));
  OP_REQUIRES_OK(context, context->input( "reverse_index_map", &reverse_index_map_t ));
  // Declaration of automatic variables
  const auto reverse_index_map = reverse_index_map_t->vec<int64>();
  const auto grad_values = grad_values_t->vec<T>();
  ...
  // Getting the size of the tensor
  const int64 N = reverse_index_map_t->shape().dim_size(0);
  const int64 N_full = grad_values_t->shape().dim_size(0);
  const Tensor& indices_t = context->input(kIndicesInput);
  ...
  Tensor* d_values_t;
  OP_REQUIRES_OK(context, context->allocate_output ("d_values", TensorShape({N}), &d_values_t));
  ...
  auto d_values = d_values_t->vec<T>();
  ...
  // Double indexing inside the for loop where write may happen outside the bound of grad_values causing
  ↪  heap-based buffer overflow
  for (int i=0; i < N; ++i) {
    d_values(i) = grad_values( reverse_index_map(i));
    visited(reverse_index_map(i))= true;
  }
}
```

Listing 1: Heap-based buffer overflow in TensorFlow.



```
void func(){
    char* data;
    char* dataBadBuffer = (char*)ALLOCA(50*sizeof(char));
    memset(dataBadBuffer, "A", 50-1);
    dataBadBuffer[50-1] = "\0";
    goto source;
    source:
        data=dataBadBuffer;
        size_t i, destLen;
        char dest[100];
        memset(dest, "C", 100-1);
        dest[100-1] = "\0";
        destLen = strlen(dest);
        for (i=0; i<destLen; i++) {
            dest[i] = data[i];
        }
        dest[100-1] = "\0";
        printLine(dest);
}
```

```
✓ memset(dataBadBuffer, "A", 50-1);
✓ dataBadBuffer[50-1] = "\0";
✓ memset(dest, "C", 100-1);
✓ dest[100-1] = "\0";
✗ dest[i] = data[i];
✓ dest[100-1] = "\0";
```

```
...
for (int i=0; i < N; ++i) {
        int reverse_index = reverse_index_map[i];
        d_values[i] = grad_values[reverse_index];
        visited[reverse_index] = true;
    }
...
```

```
void Compute (OpKernelContext* context) override {
    const auto reverse_index_map = reverse_index_map_t->vec<int64>();
    const auto grad_values = grad_values_t->vec<T>();
    const int64 N = reverse_index_map_t->shape().dim_size(0);
    const int64 N_full = grad_values_t->shape().dim_size(0);
    auto d_values = d_values_t->vec<T>();
    for (int i=0; i < N; ++i) {
        d_values(i)=grad_values(reverse_index_map(i));
        visited(reverse_index_map(i))=true;
    }
}
```

```
void Compute (OpKernelContext* context) override {
    vector<int64> reverse_index_map = reverse_index_map_t->vec<int64>();
    vector<T> grad_values = grad_values_t->vec<T>();
    const int64 N = reverse_index_map_t->shape().dim_size(0);
    const int64 N_full = grad_values_t->shape().dim_size(0);
    vector<T> d_values = d_values_t->vec<T>();
    for (int i=0; i < N; ++i) {
        int reverse_index = reverse_index_map[i];
        d_values[i] = grad_values[reverse_index];
        visited[reverse_index] = true;
    }
}
```

Figure 3.1: A motivating example.

**Key Insight.** To mitigate the above-mentioned challenges, we can co-design these two approaches to embrace the advantages from both sides. Specifically, we can improve the efficiency via machine learning whose outputs will be refined by a more precise symbolic engine. Figure 3.1 shows the overview of our approach with the same running example.

While we have a rich dataset available on CVE in basic C++ dataset [189], it is challenging to use the dataset to train a good model for identifying vulnerable patterns in real-world software due to the differences in data structures and code semantics. To reduce the gap, we need a compact semantic representation for vulnerable code.

From Listing 1, observe that it requires contextual information to identify the vulnerability (*i.e.,* line 24 in Listing 1), we need to understand how *reverse_index_map* is initialized and how the other parts of the program modify this variable. Meanwhile, most other information is irrelevant with respect to the vulnerability. To address this issue, we design a *k-bounded* program slicing technique to maintain a small amount of context relevant to vulnerability and use this to perform our prediction. To capture as many vulnerabilities as possible, we finetune the hyperparameters of the machine learning model towards a low false negative rate, which could lead to many false alarms.

Next, we refine the output of the ML model with a novel symbolic execution framework. Unlike traditional symbolic execution engines, our engine has to deal with two major challenges: first, it has to reason about partial code snippets instead of fully compilable code, and second, it needs to reason about programs that manipulate complex data structures without generating complex constraints that are hard to handle by off-the-shelf solvers. For instance, capturing the bug at line 24 requires precise reasoning over containers (*i.e.,* maps and tensors), which is hard to scale. To address this challenge, our symbolic engine can handle customized abstract domains that are relevant to the vulnerabilities. In this case, although the elements of the tensor could be arbitrarily complex, it is sufficient to capture the bug by reasoning about the size of the tensor. In this case, our tool will convert the code snippet into its *projected program* $P$ in our domain-specific language (DSL) where $P$ operates on the abstract properties of the data structures instead of its actual elements. This design choice leads to constraints that are much easier to solve while preserving the semantics of the vulnerable code.

## 3.3 System Design

In this section, we elaborate on the key components of VulDetector.

### 3.3.1 System Architecture of VulDetector

Figure 3.2 shows the overall architecture of VulDetector. Its core contains a co-design of a machine-learning model and a customized symbolic execution engine. In particular, VulDetector has four major components: (1) slice generator, (2) normalization, (3) BGRU-based neural architecture, and (4) symbolic engine.

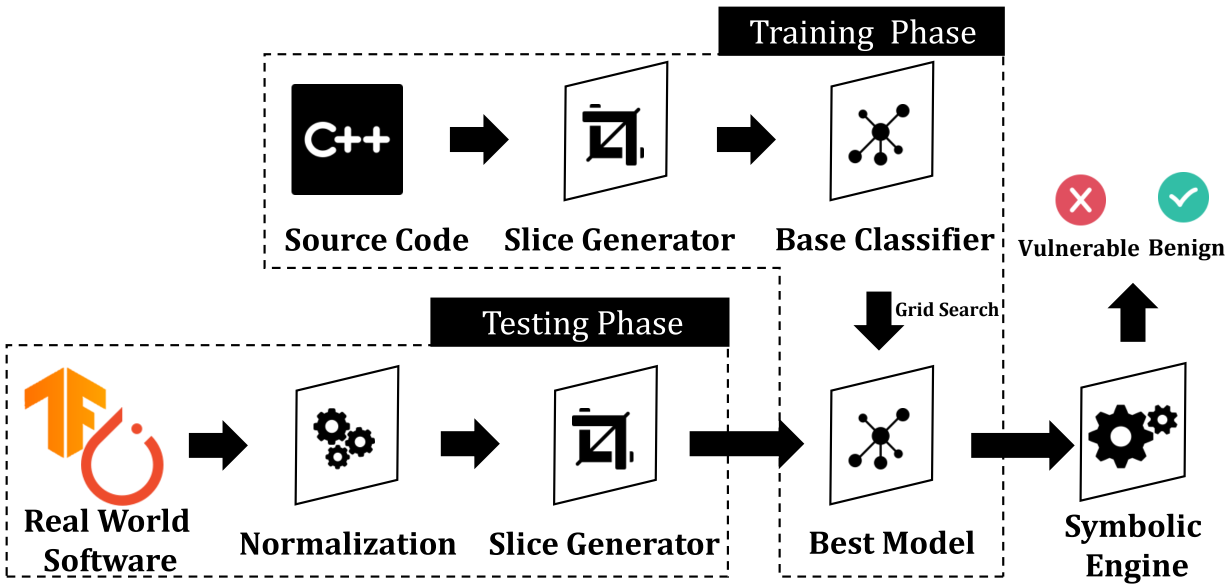


Figure 3.2: System overview of VulDetector.

First, to make sure the machine learning models get accurate signals from the dataset, we propose a novel design of a slice generator to extract program slices that are relevant to vulnerabilities. Second, to reduce the semantic gap in different software (the vulnerability dataset and the real-world software), we propose the normalization component. Third, we train a machine learning model to classify the vulnerable code snippets with a high recall. Finally, we design a novel symbolic engine to refine the output from the machine learning model.

### 3.3.2 Slice Generator

Given a program location in which a bug may be triggered (*e.g.,* memory access, buffer read/write), VulDetector's program slice generator is used to extract the program segments that are relevant to vulnerabilities. Since a naive backward slicing (*i.e.,* goes all the way to the entry) may contain too much noisy information that may confuse the ML model, we only maintain a bounded number of contexts.

#### Program Dependence Graph

To understand whether a particular operation may result in vulnerability, we need to track how different variables are initialized, modified, and used. One way to unfold those is to leverage a program dependence graph (PDG), which captures semantic information of a program via data- and control dependencies. VulDetector creates the PDG by leveraging data dependencies and control dependencies among all the statements and predicates in the program. Formally, we define the program dependence graph as (V, C, D) where:

1. V is a set of interesting vertices, where each vertex $v \in V$ represents an interesting statement in the program. We define statements as interesting if they match vulnerability syntax characteristics, including $PointerUse$, $ArrayUse$, $ArithmeticExpression$, and $SensitiveAPI$ [112, 114].

2. C indicates the control-dependency edges. Given, $(v, v') \in C$, execution of $v'$ is dependent on $v$. $v'$ will only get executed if the previous instruction (which is $v$) executes in a way that allows $v'$ to be executed.

3. D represents the data-dependency edges. Let's consider the following relation $(v, v', d) \in D$. Here, both $v$ and $v'$ vertices are connected by the data, $d$. It indicates $v$ is data dependent on $v'$ if: a) $v'$ is an assignment, and b) the value assigned in $v'$ can be referenced from $v$

**Example 1.** *We have demonstrated an example of creating PDG from the code in Figure 3.3. Here, all the nodes indicate the statements of the program. For example, Line 9 illustrates an assignment statement, where it assigns values to a tensor. We mark it as an interesting point. And in the PDG, we can observe data from Line 1,2,6,7,8 are flowing to this node. And in Line 8, there is a for loop which iterates up to 'N' (size of 'reverse_index_shape'). It controls whether Lines 8 and 9 will be executed or not. If the value of 'i' becomes equal to N it terminates the loop. That's why we notice a control edge from Line 8 to 9.*

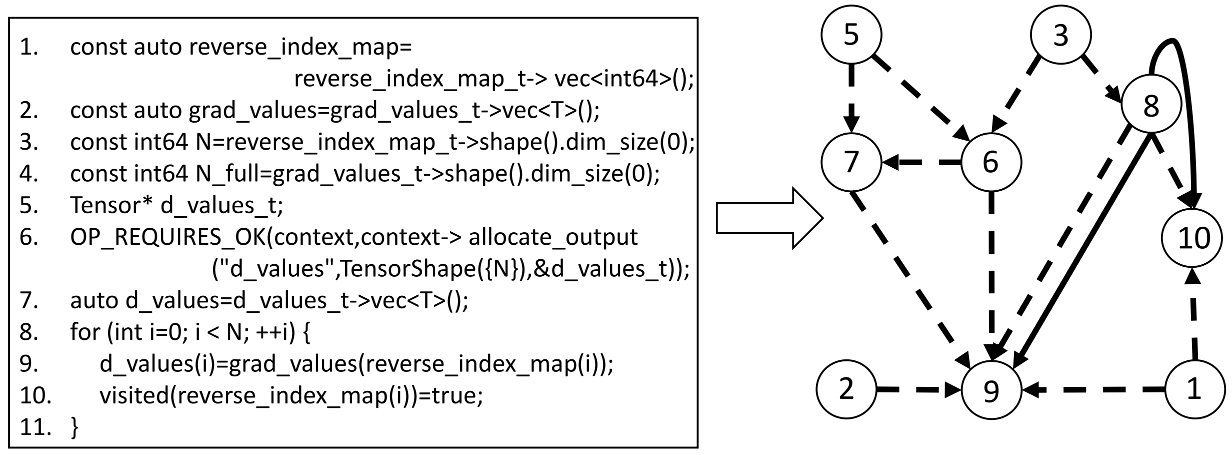

```
1.   const auto reverse_index_map=
                              reverse_index_map_t-> vec<int64>();
2.   const auto grad_values=grad_values_t->vec<T>();
3.   const int64 N=reverse_index_map_t->shape().dim_size(0);
4.   const int64 N_full=grad_values_t->shape().dim_size(0);
5.   Tensor* d_values_t;
6.   OP_REQUIRES_OK(context,context-> allocate_output
                         ("d_values",TensorShape({N}),&d_values_t));
7.   auto d_values=d_values_t->vec<T>();
8.   for (int i=0; i < N; ++i) {
9.       d_values(i)=grad_values(reverse_index_map(i));
10.      visited(reverse_index_map(i))=true;
11.  }
```

Figure 3.3: Generating PDG diagram. $\dashrightarrow$ indicates data dependency edge and $\rightarrow$ indicates control dependency edge.

**PDG Predicates** VulDetector provides built-in predicates to encode the generated PDG. In particular, the `flowTo`$(y, x)$ predicate indicates that the value of $x$ has data dependence on $y$. Similarly, the `followBy` predicate is inferred from the contract's CFG (Control Flow Graph). Intuitively, `followBy`$(L_1, L_2)$ holds for $L_1$ and $L_2$ if both are in the same basic block and $L_2$ follows $L_1$, or there is a path from the basic block of

$L_1$ to the basic block of $L_2$. The PDG predicates are computed using the following datalog rules:

$$\begin{aligned}
\mathsf{flowTo}(x,y) &\quad :- \quad \mathsf{alloc}(\_,y,x)\\
\mathsf{flowTo}(x,y) &\quad :- \quad \mathsf{assign}(\_,y,x)\\
\mathsf{flowTo}(x,z) &\quad :- \quad \mathsf{assign}(\_,y,x),\mathsf{flowTo}(y,z)\\
\mathsf{flowTo}(x,z) &\quad :- \quad \mathsf{alias}(y,z),\mathsf{flowTo}(x,y)\\
\mathsf{followBy}(x,y) &\quad :- \quad \mathsf{follow}(x,y)\\
\mathsf{followBy}(x,z) &\quad :- \quad \mathsf{followBy}(y,z),\mathsf{follow}(x,y)
\end{aligned}$$

Here, we use the `follow`$(L_1, L_2)$ as the base case which holds if $L_2$ immediately follows $L_1$ in the CFG.

**K-bounded Slices**

PDG is a general representation of a program but contains too much irrelevant information for vulnerability detection. A common approach is to compute a slice of the PDG for a specific statement of interest, referred to as a *hotspot*. However, a naive slice from a hotspot to the program's entry can result in a large graph. To address this, we propose a K-bounded slice that keeps the hotspot and limits the context.

Before we introduce the rule for the K-bounded slice, we first introduce a couple of auxiliary notations. First, the `hotspot`$(x)$ predicate denotes that the instruction at location $x$ is a sensitive statement regarding a particular type of vulnerability. For instance, Lines 26-27 in Listing 1 contains an array usage operation. It matches the vulnerable syntax characteristics, that's why we mark this statement as a hotspot.

Second, we inductively define a distance function that measures the number of hops between two statements:

$$\begin{cases}\mathsf{len}(x,x)=0\\ \mathsf{len}(x,z)=K+1 \quad \text{if } \mathsf{len}(x,y)=K \wedge \mathsf{follow}(y,z)\end{cases}$$

For example, in Listing 1, the distance function between hotspot (*i.e.,* $d\_values$) and 1-hop distance function is in Line 24, where they initialized a $d\_values\_t$ type variable. And 2-hop distance function is in Line 19, which is the declaration of that tensor variable (*i.e.,* $Tensor * d\_values\_t$).

Finally, the K-bounded slice can be summarized using the following rule:

$$\mathsf{flowToK}(x,y) \quad :- \quad \mathsf{hotspot}(x),\mathsf{flowTo}(x,y),\mathsf{len}(x,y)<K$$

In other words, the `flowToK`$(x, y)$ predicate computes a subgraph whose nodes' distance to the hotspot is not greater than $K$.

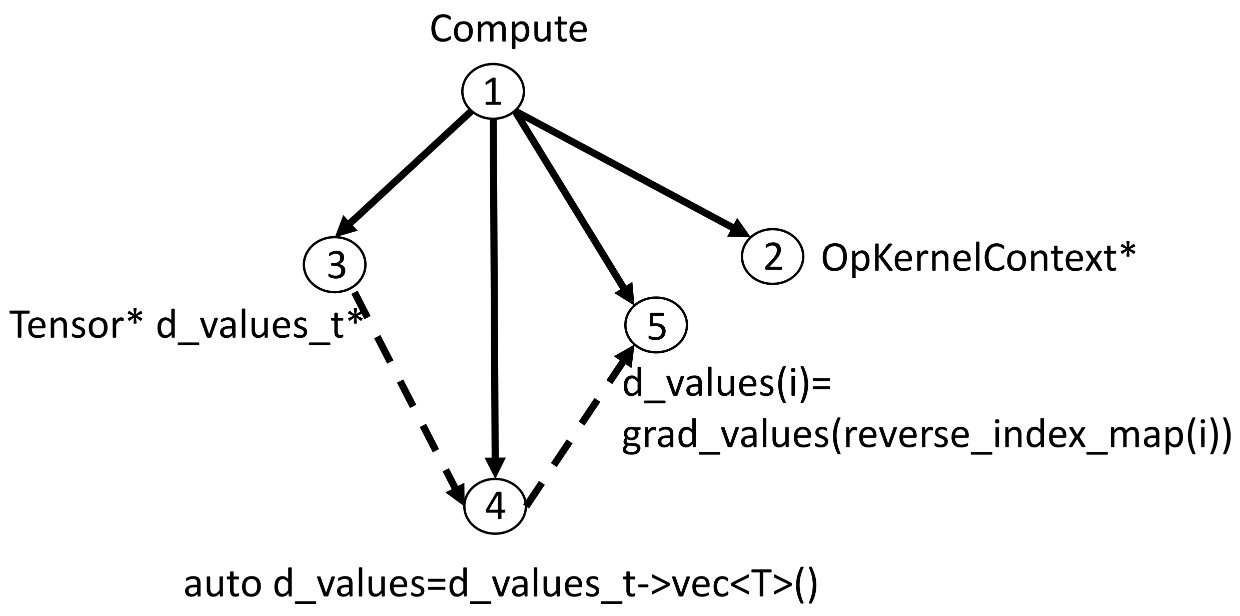

Compute
1
2
OpKernelContext*
3
Tensor* d_values_t*
5
d_values(i)=
grad_values(reverse_index_map(i))
4
auto d_values=d_values_t->vec<T>()

Figure 3.4: K-bounded slice, where K=2. $\dashrightarrow$ indicates data dependency edge and $\rightarrow$ indicates control dependency edge.

**Example 2.** *In Figure 3.4, we show a 2-bounded slice, where the hotspot is in Node 5. We generate this slice by doing reachability slicing. Here, we can observe the data-dependence edge between Node 3 & 4 (for d_values_t) and Node 4 & 5 (for d_values).* '

### 3.3.3 Normalization

Directly applying the trained model to real-world software will not perform well due to the semantic and syntactic gaps across different applications. We propose our normalization approach to overcome this challenge. We build this heuristic by going through the API documentation of each of the software. In what follows, we elaborate on the details of the normalization component.

**Conditional statement.** In many cases, condition check operations are done with some specific macros or in-line functions (*e.g.,* OP_REQUIRES, OP_REQUIRES_OK, TF_LITE_ENSURE_EQ, TF_LITE_ENSURE) which is different from commonly used conditional statements (*e.g.,*'if-else', 'while'). Control logic might hide behind these macro uses, which might lead to an incomplete PDG. So we transform them into traditional *if* statements.

**Data type inference.** We face a challenge in recognizing array/pointer type variables in programs with inferred data types at compile time(*e.g.,* auto, AutoType). To resolve this, we infer their data type by analyzing the values assigned to them.

**Macro replacement.** Function-like macros accept arguments like function calls. The benefit of using macros is they do not generate actual function calls which helps the programs to run faster by replacing function calls with macros. When the compiler comes across a macro name, it substitutes the name with the macro definition. However, most of the time developers define those macros in a separate file or require many in-depth traversals to unfold the actual values. To overcome this problem, we replace those macros with the actual value. For example, whenever we encounter *'ND_TTEST2'*, we replace it with '$(ndo->ndo\_snapend - (l) <= ndo->ndo\_snapend \&\& (constu\_char*) \& (var) <= ndo->ndo\_snapend - (l))'$.

**Removing log/debug messages.** Developers use various log messages for debugging purposes. This extraneous information will add new nodes and edges in PDG which is of no use. For example, in TensorFlow, they use 'VLOG', in Pillow they use 'TRACE, TIFFError' to show messages. VulDetector removes those relevant statements before generating the PDGs.

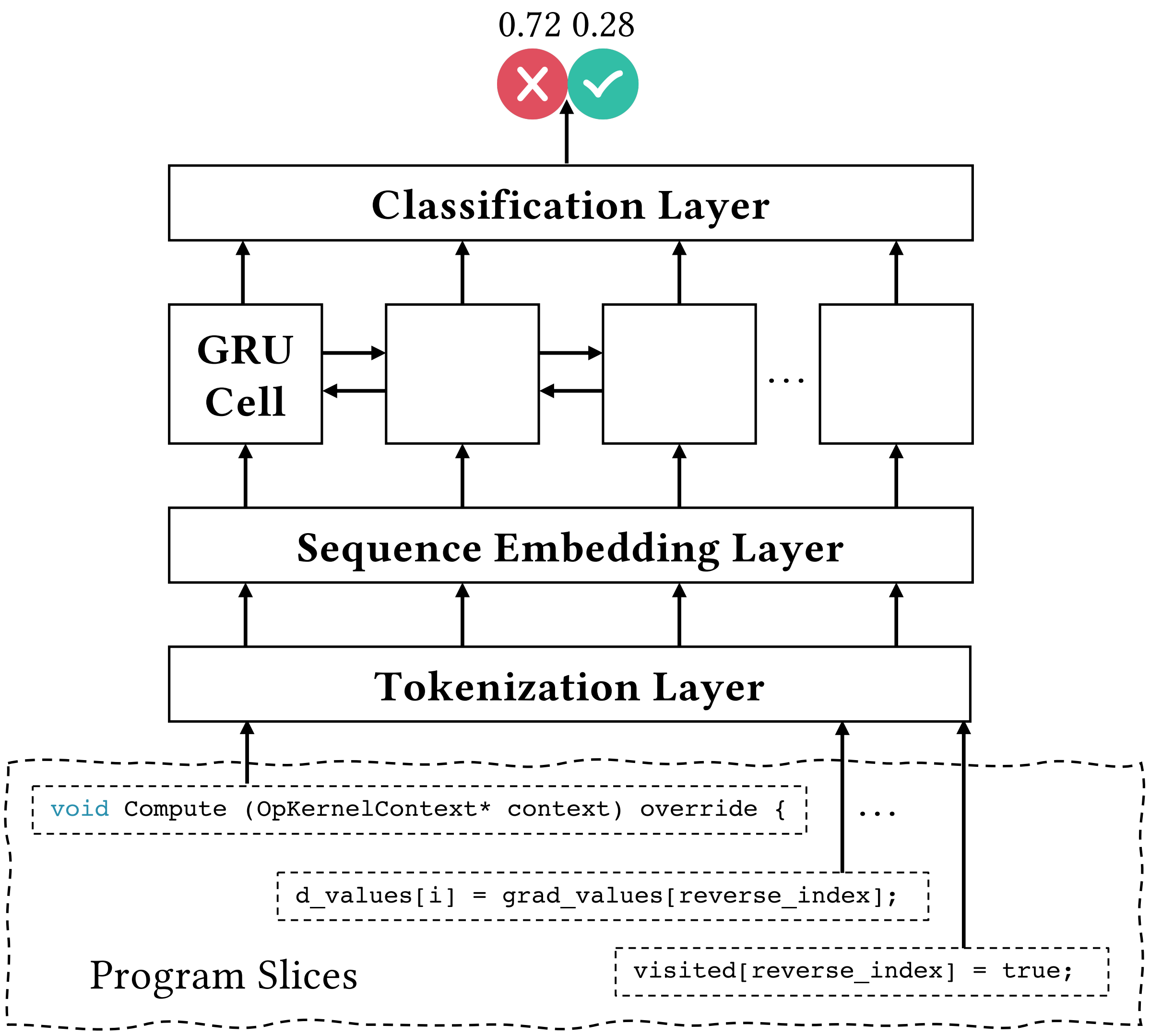
**Data structure.** The way of data structure representation varies across different open-source libraries. Even though their underlying functionality is the same. We normalize those to the same type. For example, we convert $var\_0(idx)$ to $var\_0[idx]$ after analyzing the operation. We only apply this normalization once we detect it is an array-like variable. And if it's not an array-like variable or it is just a function, we will not change the symbolic representation.

**Memory usage API.** Software implements its own API to better optimize memory usage. But the underlying logic remains the same as traditional memory usage APIs. For example, *kmalloc* does the same operation as *malloc* [131]. However, if we do not transform it to *malloc*, then our neural network will treat *kmalloc* as an unrecognized token which will negatively impact the performance of the model. As we show later in our evaluation, we investigate all seven software API documents to better convert the memory operation-related APIs to traditional ones.

In Table 3.1 we have illustrated the complete list of semantics for each of the six different types of normalization approach that we adapted to reduce the semantic gaps across different software.

| Type | Semantics |
|---|---|
| Conditional statement | ND_TCHECK, ND_TCHECK2, ND_TTEST2, OP_REQUIRES, OP_REQUIRES_OK, TF_LITE_ENSURE_EQ, TF_LITE_ENSURE |
| Data type inference | auto, AUTOTYPE |
| Macro replacement | WCHAR, UINT32, UINT16, UINT, ISC_REQ_*, ISC_RET_*, ASC_REQ_*, ASC_RET_*, SEC_WINNT_AUTH_*, SECPKG_FLAG_*, SEC_E_*, SEC_I_*, EXTRACT_LE_{*}BITS |
| Removing log/debug | GF_LOG, ND_PRINT, VLOG, TRACE, TIFFError |
| Data structure | convert array-like variables |
| Memory usage API | kvmalloc, kzalloc, kzalloc_node, kmalloc, kcalloc, kfree, ksize |

Table 3.1: Normalization.



```
void Compute (OpKernelContext* context) override {
d_values[i] = grad_values[reverse_index];
visited[reverse_index] = true;
```

Figure 3.5: Bidirectional Gated Recurrent Unit architecture used in VulDetector.

### 3.3.4 Bidirectional Gated Recurrent Unit-based Neural Architecture

The neural constituent of our machine learning model, as illustrated in Figure 3.5, is a Bidirectional Gated Recurrent Unit (BGRU) constructed upon a tokenization layer. We opted for this particular architecture due to its best performance relative to the other models we experimented with, such as, Convolutional Neural Networks and Bidirectional Long Short-Term Memory.

**Tokenization.** To capture the semantic information, we first tokenize the code and transform program-specific tokens into symbolic forms for the neural component to learn a program-agnostic representation and generalize to unseen samples. The tokenization layer runs a standard tokenization process that decouples every line of code into a sequence of basic units — programming language keywords, variables, operators, and other language terminal and non-terminal symbols. It captures semantic information to train the neural network. It removes all the non-ASCII characters and comments. As these are not useful to determine the vulnerability of the code, rather they would introduce noise that will make the model confuse. Post-processing is applied to anonymize program-specific tokens — variables, user-defined identifiers, string literals, macros, etc. — into their symbolic forms. For example, variables such as, `d_values` and `reverse_index` will become `var0` and `var1`, respectively. We also replaced the name of the functions with `funcX`, where X indicates the function number in the program (chronologically). This helps the neural component to learn a program-agnostic representation, focus on the commonly shared features and generalize to unseen samples. The tokenization layer then concatenates the sequences of tokens from lines of codes according to their topological orderings from PDG.

**Vectorization.** ML model recognizes only the numerical form. That's why we need to convert the tokens into representative vectors. To achieve this, we use lexical analysis to separate the symbolic representation into a series of symbols. For example, we encode `var0[9]=var1;` to `'var0', '[', '9', ']', '=', 'var1', ';'`. Then, We convert each of those symbols into a vector with the help of a Word2Vec model [134]. We train the Word2Vec model using the C/C++ corpus. Note that, we use the CVE dataset [189] to train the Word2Vec model. With the help of this model, we generate the corresponding vector for each of the symbolic forms. To obtain the full vectorized representation of the line of code, we concatenate those vectors. However, the number of symbols in each line may vary. But the neural network takes the vectors of the same length as input. Otherwise, it cannot process the inputted vectors. That's why we set the dimension of each vector $\mathcal{V}_{dim}$ beforehand. We use this value as the length of vectors for the input to the neural network. Whenever a vector becomes shorter than $\mathcal{V}_{dim}$ then we apply zero padding at the end of the vector. For vectors larger than $\mathcal{V}_{dim}$, we truncate the trailing part to keep the vector length consistent. In this way, we construct the vectors corresponding to each line of the code in the slice.

Again, the lines of code in each slice may vary from others. To keep the corresponding vectors for each slice consistent, we set the maximum line of the code handled by our neural network to be $\mathcal{M}_{len}$. Similar to $\mathcal{V}_{dim}$, we zero pad the vectors shorter than $\mathcal{M}_{len}$ whereas truncating the trailing vectors when it is larger than $\mathcal{M}_{len}$.

**Layers.** We introduce $\mathcal{L}_n$, n numbers of layers containing GRU cells. Here, n depends on the number of layers based on the experiment settings. To avoid the overfitting issue, we set the dropout after this layer. Again, the value of dropout depends on the experiment settings. We select the following hyperparameters through a series of experiments in a grid search manner. They are– vector dimension, batch size, maximum vector length, number of layers, dropout, optimizer, learning rate, and epoch. We set the mask value to 0 in our model's network. Mask value refers to every timestep in the input, if all input values at that timestep are equal to 0, then the timestep will be skipped in all the downstream layers.

**GRU-cell.** We then encode the sequence of token representation into $V$.

$$E(Q) = (E(v_0), E(v_1), ..., E(v_n))$$

Internally at every time step $t$, a token (representation) $E(v_t)$ is fed to the GRU cell and computes the following functions as given by the GRU mechanism:

$$\begin{aligned} z_t &= \sigma(W_z E(v_t) + U_z h_{t-1}) \\ r_t &= \sigma(W_r E(v_t) + U_r h_{t-1}) \\ h'_t &= \tanh(W E(v_t) + r_t \odot U h_{t-1}) \\ h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot h'_t, \end{aligned}$$

where $W$ refers to *learnable* network parameters, $U$ corresponds to intermediate results $z_t$ denotes an *update gate*, $r_t$ indicates a *forget gate*, $h'_t$ and $h_t$ are the vector representation of the tokens *up to* the current time step. $\sigma$ and tanh are two kinds of commonly used activation functions, and $\odot$ denotes the Hadamard product (element-wise product). Essentially GRU outputs the hidden state of the last time step $h_n$ as the representation of the whole sequence: $GRU(E(Q)) = h_n$. In addition, a bidirectional GRU concatenates the representations of the original sequence and its reverse $Q^{-1} = (v_n, v_{n-1}, ..., v_0)$ into:

$$V = [h_n : h_n^{-1}],$$

where $h_n^{-1} = GRU(Q^{-1})$.

The BGRU network takes as input the sequence of tokens generated by the tokenization layer, encodes it in a recurrent way that respects the linear dependency between consecutive tokens, and performs a binary classification (benign/vulnerable). In particular, given a sequence of programming language tokens $Q$:

$$Q = (v_0, v_1, ..., v_n),$$

the BGRU network models its probability of being vulnerable $P(\texttt{vul}|Q)$, which can further be decomposed conditionally across time in BGRU:

$$\begin{aligned} P(\texttt{vul}|Q) &= P(\texttt{vul}|v_1, v_2, ..., v_n) \\ &= P(\texttt{vul}|E(v_0), E(v_1), ..., E(v_n)) \\ &= P(\texttt{vul}|V) \end{aligned}$$

where $E(v_t)$ corresponds to *learnable* vector representation of every token given by the embedding layer, and $V$ corresponds to the vector representation of the program slice.

**Classification.** We label each of the input vectors as '1' (representing vulnerable code) and '0' (representing benign code). A classification layer then estimates $P(\texttt{vul}|Q)$ given the above outputs:

$$P(\texttt{vul}|Q) = \sigma(W_u V + b_u),$$

where $W_u$ and $b_u$ are both *learnable* parameters.

**Grid Search.** We build BGRU model using Keras 2.6.0. We select the following set of hyperparameters while performing grid search experiment: vector dimension $\mathcal{V}_{dim}$ = {16, 32, 64}, batch size $\mathcal{B}$ = {8, 16, 32}, maximum vector length $\mathcal{M}_{len}$ = {150, 200, 250,300}, layers $\mathcal{L}_n$ = {2,3,4}, dropouts $\mathcal{D}$ = {0, 0.2, 0.5}, optimizers $\mathcal{O}$ = {adam, adamax}, learning rate $\alpha$ = {0.1, 0.01, 0.001, 0.0001}, *epoch* = 50. In each step of our grid search algorithm, we take a single value for setting each of the hyperparameters listed above and compute the model performance on the validation data. After training the model with all the combinations, we select the best model based on the performance of the validation data.

Next, whenever our ML model is having more than 50% of confidence that one slice exhibits vulnerable characteristics, then we provide that slice to the symbolic engine for further refinements.

### 3.3.5 Symbolic Engine

Recall that VulDetector generates a potentially malicious code snippet $P$ from the output of the BGRU module. The output could lead to false positives due to the limitation of ML models. Then, VulDetector invokes its symbolic execution (SE) module to further refine the result. In this section, we first elaborate on the way we abstract the program semantics for modeling vulnerabilities in a given code snippet, followed by an overview of the workflow of the SE module. We then elaborate on more of the inference rules of the SE module and give an example of how it works.

$$
\begin{array}{lcl}
\langle \mathit{proc} \rangle & ::= & (\text{proc } (\langle \mathit{var} \rangle\ \langle \mathit{var} \rangle^*)\ \langle \mathit{block} \rangle) \\
\langle \mathit{block} \rangle & ::= & (\text{begin } \langle \mathit{stmt} \rangle^*) \\
\langle \mathit{stmt} \rangle & ::= & \langle \mathit{def} \rangle \mid \langle \mathit{assign} \rangle \mid \langle \mathit{ret} \rangle \mid \langle \mathit{cond} \rangle \\
 & \mid & \langle \mathit{loop} \rangle \mid \langle \mathit{call} \rangle \mid \varnothing \\
\langle \mathit{def} \rangle & ::= & (\text{define } \langle \mathit{var} \rangle\ \langle \mathit{expr} \rangle) \\
\langle \mathit{assign} \rangle & ::= & (\text{set! } \langle \mathit{var} \rangle\ \langle \mathit{expr} \rangle) \\
\langle \mathit{ret} \rangle & ::= & (\text{return } \langle \mathit{expr} \rangle) \\
\langle \mathit{cond} \rangle & ::= & (\text{if } (\langle \mathit{expr} \rangle\ \langle \mathit{block} \rangle\ \langle \mathit{block} \rangle)) \\
\langle \mathit{loop} \rangle & ::= & (\text{loop } (\langle \mathit{var} \rangle\ \langle \mathit{var} \rangle)\ \langle \mathit{block} \rangle) \\
\langle \mathit{call} \rangle & ::= & (\text{call } \langle \mathit{func} \rangle\ \langle \mathit{var} \rangle^*) \\
\langle \mathit{expr} \rangle & ::= & (\langle \mathit{binop} \rangle\ \langle \mathit{var} \rangle\ \langle \mathit{var} \rangle) \\
 & \mid & (\langle \mathit{unop} \rangle\ \langle \mathit{var} \rangle\ \langle \mathit{var} \rangle) \\
 & \mid & \langle \mathit{var} \rangle \\
\langle \mathit{binop} \rangle & ::= & + \mid - \mid * \mid < \mid <= \mid == \mid != \mid >= \mid > \\
 & \mid & \&\& \mid || \mid \text{index} \mid \text{range} \\
\langle \mathit{unop} \rangle & ::= & ! \mid \text{size} \mid \text{shape} \mid \text{flat} \mid \text{dimsize} \mid \text{tensor?}
\end{array}
$$

$$\langle \mathit{var} \rangle \in \textbf{symbols} \quad \langle \mathit{func} \rangle \in \textbf{functions}$$

Figure 3.6: Syntax for Toy Symbolic Execution Language.

**Program Abstraction** Since faithfully executing the given code snippet would result in scalability issues, VulDetector incorporates domain-specific languages for modeling the aforementioned types of vulnerabilities, which provide faster and more precise reasoning by abstracting out the related semantics. To see how, we first introduce an example DSL for modeling potential vulnerability in array use, as shown in Figure 3.6. The DSL includes language constructs that model common program behavior such as statements, conditions, and loops, which are shown below:

- $\langle \mathit{proc} \rangle$ denotes the scope of a program and its given parameters.

- $\langle \mathit{block} \rangle$ defines the scope of a program.

$$\frac{\begin{array}{c}\langle\delta,\vec{s}\rangle \qquad \vec{s}=(s_0,s_1,...,s_n)\\ \langle\delta,s_0\rangle\rightsquigarrow\langle\delta_1,\varnothing\rangle\\ ...\\ \langle\delta_n,s_n\rangle\rightsquigarrow\langle\delta',\varnothing\rangle\end{array}}{\langle\delta,\texttt{(begin } s_0...s_n\texttt{)}\rangle\rightsquigarrow\langle\delta',\varnothing\rangle}\ \text{(block)} \qquad \frac{\begin{array}{c}\langle\delta,e\rangle\rightsquigarrow\langle\delta,R\rangle\\ x\notin\mathrm{dom}(\delta) \qquad \delta'=\delta\cup\{x\mapsto R\}\end{array}}{\langle\delta,\texttt{(define x e)}\rangle\rightsquigarrow\langle\delta',\varnothing\rangle}\ \text{(def)}$$

$$\frac{\begin{array}{c}\langle\delta,e\rangle\rightsquigarrow\langle\delta,R\rangle\\ x\in\mathrm{dom}(\delta) \qquad \delta'=\{y\mapsto\delta(y)\mid y\in\mathrm{dom}(\delta)\wedge y\neq x\}\cup\{x\mapsto R\}\end{array}}{\langle\delta,\texttt{(set! x e)}\rangle\rightsquigarrow\langle\delta',\varnothing\rangle}\ \text{(assign)}$$

$$\frac{\langle\delta,e\rangle\rightsquigarrow\langle\delta,R\rangle}{\langle\delta,\texttt{(return e)}\rangle\rightsquigarrow\langle\delta',R\rangle}\ \text{(ret)} \qquad \frac{x\in\mathrm{dom}(\delta) \qquad \delta(x)=R}{\langle\delta,x\rangle\rightsquigarrow\langle\delta,R\rangle}\ \text{(var)}$$

$$\frac{\begin{array}{c}\langle\delta,c\rangle\rightsquigarrow\langle\delta,R_b\rangle \qquad R_b=\{\langle\pi_t,\texttt{true}\rangle,\langle\pi_f,\texttt{false}\rangle\}\\ \langle\delta,b_0\rangle\rightsquigarrow\langle\delta_0,R_0\rangle \qquad \langle\delta,b_1\rangle\rightsquigarrow\langle\delta_1,R_1\rangle\end{array}}{\langle\delta,\texttt{(if } c\ b_0\ b_1\texttt{)}\rangle\rightsquigarrow\langle\delta\uplus\delta_0\uplus\delta_1,\phi(R_b,R_0\odot\pi_t,R_1\odot\pi_f)\rangle}\ \text{(cond)}$$

$$\frac{\begin{array}{c}\delta(r)=\langle\pi_r,v_r\rangle \qquad v_r\neq\varnothing\\ \langle\delta,x\rangle\rightsquigarrow\langle\delta_0,\langle\pi_r,v_r[0]\rangle\rangle \qquad \delta_0=\delta\uplus\{x\mapsto\langle\pi_r,v_r[0]\rangle\}\\ \langle\delta_0,b\rangle\rightsquigarrow\langle\delta',\varnothing\rangle\end{array}}{\langle\delta,\texttt{(loop (}x\ r\texttt{) }b\texttt{)}\rangle\rightsquigarrow\langle\delta',\varnothing\rangle}$$

(loop-body)

$$\frac{\delta(r)=\langle\pi_r,v_r\rangle \qquad v_r=\varnothing}{\langle\delta,\texttt{(loop (}x\ r\texttt{) }b\texttt{)}\rangle\rightsquigarrow\langle\delta,\varnothing\rangle}\ \text{(loop-end)} \qquad \frac{\begin{array}{c}\vec{x}=(x_0,...,x_n) \qquad \delta(x_0)=R_0 \qquad ... \qquad \delta(x_n)=R_n\\ \langle\delta,\vec{x}\rangle\rightsquigarrow\langle\delta_0,\varnothing\rangle \qquad \delta_0=\{x_i\mapsto R_i|x_i\in\vec{x}\}\\ \langle\delta_0,f\rangle\rightsquigarrow\langle\delta',R'\rangle\end{array}}{\langle\delta,\texttt{(call }f\ \vec{x}\texttt{)}\rangle\rightsquigarrow\langle\delta',R'\rangle}\ \text{(call)}$$

$$\frac{\langle\delta,x\rangle\rightsquigarrow\langle\delta,R\rangle}{\langle\delta,\texttt{(}\dagger\ x\texttt{)}\rangle\rightsquigarrow\langle\delta,\{\langle\pi_i,\dagger v_i\rangle\mid\forall\langle\pi_i,v_i\rangle\in R\}\rangle}\ \text{(unop)}$$

$$\frac{\langle\delta,x_0\rangle\rightsquigarrow\langle\delta,R_0\rangle \qquad \langle\delta,x_1\rangle\rightsquigarrow\langle\delta,R_1\rangle}{\begin{array}{c}\langle\delta,\texttt{(}\oplus\ x_0\ x_1\texttt{)}\rangle\rightsquigarrow\\ \langle\delta,\{\langle\pi_i\wedge\pi_j,v_i\oplus v_j\rangle\mid\forall\langle\pi_i,v_i\rangle\in R_0,\forall\langle\pi_j,v_j\rangle\in R_1\}\rangle\end{array}}\ \text{(binop)}$$

Figure 3.7: Major Inference Rules for Symbolic Execution. † and ⊕ represent unary and binary operators, respectively.

- ⟨*stmt*⟩ defines 7 kinds of common statements, namely definition, assignment, return, condition, loop, function call, and empty statement.
  - ⟨*def*⟩ declares a variable and initializes it with the given value.
  - ⟨*assign*⟩ assigns a given value to an existing variable.
  - ⟨*ret*⟩ ends a ⟨*proc*⟩ block and returns a given expression.

  - $\langle cond \rangle$ evaluates a boolean expression and executes one of the two given branches depending on the truthfulness of the expression.
  - $\langle loop \rangle$ defines and executes a loop with a given initial expression, condition, and loop body.
  - $\langle call \rangle$ delegates the program execution to a previously defined program with corresponding expressions provided as arguments.
  - $\varnothing$ denotes an empty statement that does nothing when executed.

- $\langle expr \rangle$ defines 3 kinds of expressions, namely binary expression $\langle binop \rangle$, unary expression $\langle unop \rangle$ and single variable $\langle var \rangle$.
  - $\langle binop \rangle$ denotes binary expressions, including arithmetic operators such as $+$, $-$ and $*$, boolean operators such as $<$, $\leq$, etc. and array use related operators index and range which will be elaborated below.
  - $\langle unop \rangle$ denotes unary expressions, including a negation operator ! and array use related operators that are elaborated below.
  - $\langle var \rangle$ denotes a variable name.

Besides the common part that is shared by DSLs for different vulnerabilities, the example DSL also defines specific operators for array use. We elaborate on them below.

- index(arr, $i$) takes as input an array arr and returns its $i$th element.
- range($s, e$) returns an array of numbers from $s$ to $e$ with an interval of 1.
- size(arr) returns the total number of elements of array arr.
- shape(arr) returns a tuple representing shape of array arr.
- flat(arr) flattens array arr into its one dimensional version.
- dimsize(arr) returns the number of dimensions.
- tensor?(v) returns a boolean indicating whether the given variable is an array or not.

With details of the DSL, we then introduce the workflow of our symbolic engine and inference rules.

**Workflow** The overall workflow of the SE module contains three parts, as stated below:

$$\frac{\delta_0 \quad \delta_1 \quad \delta = \{\langle \pi_i \wedge \pi_j, x_i \rangle \mid \forall \langle \pi_i, x_i \rangle \in \delta_0, \\ \forall \langle \pi_j, x_j \rangle \in \delta_1, (x_i = x_j) \wedge (\pi_i \wedge \pi_j)\}}{\langle \varnothing, \delta_0 \uplus \delta_1 \rangle \rightsquigarrow \langle \varnothing, \delta \rangle} \text{ (E-merge } \uplus\text{)}$$

$$\frac{\pi \quad R = \{\langle \pi_i, v_i \rangle \mid i = 1...n\} \\ R' = \{\langle \pi_i \wedge \pi, v_i \rangle \mid i = 1...n, \pi_i \wedge \pi\}}{\langle \varnothing, R \odot \pi \rangle \rightsquigarrow \langle \varnothing, R' \rangle} \text{ (C-merge } \odot\text{)}$$

Figure 3.8: Symbolic Merging Rules.

- **Code Parsing and Mapping** Since the given code snippet can be in different languages, VulDetector first incorporates a transpilation procedure to parse them into programs of DSL shown in Figure 3.6. Such a procedure identifies and translates program constructs into their corresponding DSL components.

- **Code Variable Initialization** VulDetector then identifies the variables and their types and initialized values in the program.

- **Symbolic Reasoning** The SE module then executes and checks for vulnerability according to the operational semantics defined in Figure 3.7 and Figure 3.8.

**Inference Rules** Figure 3.7 shows a representative subset of the major inference rules that the SE module uses to execute the program and check for vulnerabilities. A program state corresponds to the value environment $\delta$ that stores a mapping from variables to their corresponding *guarded values*. Here, we refer to a guarded value $R$ as a set of tuples $\langle \pi, v \rangle$ where $\pi$ corresponds to a path condition and $v$ corresponds to a value under path condition $\pi$. A rule $\langle \delta, e \rangle \rightsquigarrow \langle \delta', R \rangle$ corresponds to a successful execution of $e$ in the program state $\delta$, which results in guarded value $R$ and state $\delta'$. We now elaborate the inference rules in more detail as below.

- **Block execution**. The block rule visits and executes the statements $\vec{s}$ sequentially and meanwhile updating the value environment $\delta$.

- **Variables**. The declaration and assignment of a variable using rules def and assign update the corresponding guarded value. Specifically, assign rule updates the variable by merging two guarded values.

- **Condition**. The condition rule splits the execution path into two paths where the condition evaluates to true and false respectively, which are merged into the new environments when returned.

```
 1 void Compute(OpKernelContext* ctx){
 2     if (!(TensorShapeUtils::IsVector(shape_tensor.shape()))) {
 3         return;
 4     }
 5     int32 num_batches = shape_tensor.flat()(0);
 6     int32 samples_per_batch = 1;
 7     int32 num_dims = shape_tensor.dim_size(0);
 8     for (int32 i=1; i<num_dims; i++) {
 9         samples_per_batch *= shape_tensor.flat()(i);
10     }
11     int32 num_elements = num_batches * samples_per_batch;
12 }
```

Listing 2: Code snippet of an array-out-of-bound bug detected by symbolic execution.

- **Loop**. A loop is unrolled and executed by a bounded number of times. The $\mathsf{loop-body}$ rule executes the loop body while keeping track of the loop variables. The $\mathsf{loop-end}$ rule finalizes the results of loop execution and performs merging of the returned environments.

- **Expression**. Binary and unary expressions are modeled by binop and unop rules respectively. In particular, for a binary expression whose operands have multiple path conditions, the binop rules enumerate each possible combination and return a new guarded value with merged paths recorded.

- **Call**. The call creates a new environment that loads the values corresponding to the arguments to the callee before transferring before calling the target function $f$. It then updates the current environment with the returned value by executing $f$.

**Symbolic Execution for Bug Detection** Listing 2 shows a code snippet that contains an array-out-of-bound bug. In particular, given a tensor `shape_tensor`, the code snippet computes the sum of all values in this tensor, by looping and accumulating on every element. Since the loop starts from the 1st position (line 8), it by default assumes that the given tensor is non-empty, which is not the fact and becomes the root cause of the array-out-of-bound bug.

We elaborate on how VulDetector reasons and detects the bug using the symbolic execution engine. VulDetector starts by performing a syntax-guided translation to map the target code snippet into a program written in the DSL for symbolic execution, as shown in Listing 3.

Then VulDetector invokes the symbolic execution engine to interpret the code snippet. As `shape_tensor` is modeled as an array by default, the `if` statement in line 3 does not always return `true` and keeps one execution path. However, in the assignment of the `samples_per_batch` variable in line 11, where `index` operator accesses the `i`-th element of the tensor, in the first loop `i` is equal to 1. This splits the execution into two paths: 1) if the size of `shape_tensor` is no less than 1, `index` returns the element of the corresponding position, otherwise, 2) `index` ends up with an exception since access to position 1 is invalid.

```
(proc (snippet shape_tensor)
  (begin
     (if (! (tensor? (shape shape_tensor)))
         (return void)
         (void)
     )
     (define num_batches (index (flat shape_tensor) 0))
     (define samples_per_batch 1)
     (define num_dims (dimsize shape_tensor))
     (loop ([i (range 1 (- num_dims 1))])
       (set! samples_per_batch (* samples_per_batch (index (flat shape_tensor) i))))
     (define num_elements (* num_batches samples_per_batch))
 )
)
```

Listing 3: DSL program translated from Listing 3.

The symbolic execution identifies the bug by formalizing and capturing the above exception, thus exposing the potential array-out-of-bound bug, together with its root cause from the path with the exception.

## 3.4 Evaluation

In this section, we present the detailed process of evaluating VulDetector in detecting vulnerable code in the large-scale code base. We design our evaluation in a way to seek answers to the following four research questions:

*RQ*1 How does VulDetector perform in detecting vulnerabilities in open-source large-scale software?

*RQ*2 What is the performance comparison of VulDetector with existing tools in identifying vulnerabilities?

*RQ*3 How significant is the benefit of machine learning and symbolic engine in VulDetector?

*RQ*4 Is VulDetector effective for detecting zero-day vulnerabilities from real-world applications?

In the remainder of this section, we describe a series of experiments to answer the above questions. We used Python 3.8.11 to implement and build VulDetector, which consists of 9,232 lines of code (including data cleaning, data processing, model building, and modeling testing). We performed all the experiments on a Desktop PC running Debian OS with a 4.90GHz Intel Core i5 processor, 32GB RAM, and 8GB graphics card NVidia GTX-1060.

### 3.4.1 Data Collection

We evaluate our tool by collecting data from real-world software. To train the machine learning model, we use the CVE dataset [189]. Later, we evaluate it on seven different software. These seven datasets are collected from three different sources: (1) GitHub Security Advisories [70], (2) Android [8], and (3) CVE Fixes Dataset [16]. Next, we describe the procedure for collecting these datasets.

**CVE dataset.** First, to train and evaluate our base classifier, we collect C/C++ source code from CVE dataset [189]. Here, the data is categorized as 'good' (*i.e.,* having no vulnerabilities), and 'bad' (*i.e.,* having vulnerabilities). Later, we extract code slices using our tool. As **Table 3.2** shows, there are 40,000 slices in total, and 24.2% (9,694) of them are marked as vulnerable. We use this data to train our base classifier for the ML model.

| **Train** | | **Validation** | | **Total** |
|---|---|---|---|---|
| **#Vul** | **#Ben** | **#Vul** | **#Ben** | |
| 6,504 | 23,496 | 3,190 | 6,810 | 40,000 |

Table 3.2: CVE dataset used for training the model.

**Real world software.** In the following, we discuss our data collection procedure for seven different software.

**1. TensorFlow.** We investigate all the security advisories of TensorFlow listed in GitHub Security Advisory [71]. From there, we collect 37 files that contain vulnerabilities. All of these files are from the 'core/kernels' module. Later we use our tool to generate slices from these code files. In total, we find 49 vulnerable slices. For finding the benign slices we select the 'core/ops' module because, from our manual investigation, we find that this module tends to have many benign files. In total, we find 930 benign slices from this module. We use this data for evaluating our tool on the TensorFlow dataset.

**2. Pillow.** Similar to TensorFlow, we investigate all the advisories related to Pillow in GitHub Security Advisory [70]. we collect 11 vulnerable code files from the 'libImaging' module. For benign slices, we select all the files from '/Pillow/src/' module. We identify 23 vulnerable and 387 benign slices from Pillow.

**3. Android.** We investigate the vulnerabilities published in [8] from 2018 to 2022. From reading the description listed the description, we identify 11 vulnerable code files related to memory/buffer overflow and generated 30 vulnerable slices. For the benign code, we select all the files from 'kernel/common'. From manual investigation, we find that this module does not contain any vulnerable code. We collect 367 benign slices from this module.

Next, we collect the rest of the most popular software from CVEFixes [16]. CVEFixes provides the repository name, star, programming language, commit information, vulnerability type, CVE, CWE, vulnerable code, and patch (if available). We investigate the most popular 170 software from this database. Out of those, we select four software for our analysis based on the popularity of the software and amount of vulnerabilities. In the following, we describe our data collection process from CVEFixes:

| **Software** | **#Vul** | **#Ben** | **#Total** |
|---|---|---|---|
| Android | 39 | 367 | 406 |
| FreeRDP | 11 | 413 | 424 |
| GPAC | 52 | 1,239 | 1,291 |
| Linux | 14 | 649 | 663 |
| Pillow | 23 | 387 | 410 |
| TCPDump | 29 | 288 | 317 |
| TensorFlow | 49 | 930 | 979 |
| **Total** | **217** | **4,273** | **4,490** |

Table 3.3: Seven large real-world software for evaluating VulDetector.

**4. Linux.** We collect eight vulnerable files from which we have extracted 14 vulnerable slices. Then, we select the 'kernel' module for collecting benign files, because we do not find any vulnerable code from this module during the manual investigation. In total, we have 663 slices (14 vulnerable and 649 benign) from this software.

**5. FreeRDP.** We collect 11 vulnerable slices out of six files that contain vulnerable code. For collecting the benign slices, we select 'libfreerdp/core' module, because it contains the most number of C/C++ files and none of the files contain any vulnerable code. In total, we have 424 slices (11 vulnerable and 413 benign) from this software.

**6. TCPDump.** We extract 29 vulnerable slices from all 11 vulnerable files. We select 'the-tcpdump-group/tcpdump' module for collecting the benign slices. In total, we have 317 slices (29 vulnerable and 288 benign) from this software.

**7. GPAC.** We extract 52 vulnerable slices from all 14 vulnerable files. And for collecting the benign slices, we select 'gpac/src' module. In total, we have 1,291 slices (52 vulnerable and 1,239 benign) from this software.

### 3.4.2 Performance Evaluation of VulDetector

To evaluate the performance of VulDetector, we run a series of experiments with all seven software. In Table 3.4, we can observe that our tool achieves good Recall (*i.e.,* low false negative ratio), ranging from 74% to 90%, and good average precision of 49% VulDetector performs best in Android with an F1-Score of 61%. We have also investigated the mispredictions of our tool. It fails to detect vulnerable code when the slice contains only a few lines of code (e.g. 2-3 lines of code). Also, the definition of the vulnerability types, e.g., sensitive API, would potentially introduce false positives (since it overapproximates the semantics).

**RQ1**: VulDetector achieves a good performance with an F1-Score of 57% (49% precision, 74.28% recall, 94.71% accuracy) in detecting vulnerable code.

| | Model | Prec. | Rec. | F1 | Acc. |
|---|---|---|---|---|---|
| Android | VulDeePecker | 8% | 93% | 15% | 23% |
| | Devign | 20% | 25% | 22% | 53% |
| | Vuddy | 0% | 0% | 0% | 73% |
| | Our Tool (ML) | 45% | 90% | 60% | 91% |
| | **Our Tool (ML+SE)** | **47%** | **87%** | **61%** | **92%** |
| FreeRDP | VulDeePecker | 3% | 100% | 7% | 21% |
| | Devign | 100% | 50% | 66% | 75% |
| | Vuddy | 0% | 0% | 0% | 50% |
| | Our Tool (ML) | 41% | 82% | 55% | 96% |
| | **Our Tool (ML+SE)** | **44%** | **82%** | **57%** | **97%** |
| GPAC | VulDeePecker | 6% | 98% | 11% | 21% |
| | Devign | 100% | 14% | 25% | 33% |
| | Vuddy | 0% | 0% | 0% | 22% |
| | Our Tool (ML) | 62% | 88% | 73% | 97% |
| | **Our Tool (ML+SE)** | **63%** | **88%** | **73%** | **98%** |
| Linux | VulDeePecker | 3% | 94% | 6% | 20% |
| | Devign | 33% | 60% | 43% | 55% |
| | Vuddy | 0% | 0% | 0% | 72% |
| | Our Tool (ML) | 40% | 86% | 55% | 97% |
| | **Our Tool (ML+SE)** | **42%** | **86%** | **56%** | **98%** |
| Pillow | VulDeePecker | 16% | 93% | 28% | 27% |
| | Devign | 55% | 29% | 38% | 51% |
| | Vuddy | 0% | 0% | 0% | 48% |
| | Our Tool (ML) | 39% | 74% | 51% | 92% |
| | **Our Tool (ML+SE)** | **43%** | **74%** | **54%** | **93%** |
| TCPDump | VulDeePecker | 16% | 93% | 28% | 27% |
| | Devign | 100% | 1% | 2% | 23% |
| | Vuddy | 0% | 0% | 0% | 15% |
| | Our Tool (ML) | 63% | 83% | 72% | 94% |
| | **Our Tool (ML+SE)** | **63%** | **83%** | **72%** | **94%** |
| TensorFlow | VulDeePecker | 18% | 100% | 31% | 35% |
| | Devign | 56% | 24% | 33% | 53% |
| | Vuddy | 100% | 18% | 31% | 60% |
| | Our Tool (ML) | 43% | 86% | 57% | 94% |
| | **Our Tool (ML+SE)** | **55%** | **86%** | **67%** | **96%** |

Table 3.4: Performance comparison with Baseline.

### 3.4.3 Comparison against Baselines

For selecting baselines, we mainly focus on research works published in well-known security conferences (*e.g.,* USENIX Security, IEEE S&P, CCS, NDSS). We find the following works on vulnerable detection in those venues — VulDeePecker [114], Vuddy [100], VCCFinder [158], Arbiter [203], Taint Analyzer [227], Fuzzer [217], Driller [194]. Note that, although Devign [238] is not from security conferences, we consider this work as a baseline candidate because it is highly related to our work and also well-recognized by other security papers. We reach out to the author of Devign, but fail to get access to their code/pre-trained models. So we follow one of the most popular implementations of Devign from [48]. Among the rest of the works, vulnerability analysis on binary programs [203, 210], taint analysis [227], author reputation model [158], and fuzzing-based works [217, 194] are out of scope of this chapter. We also explore other works from different conferences but are not able to compare them as a baseline due to the unavailability of source code [53, 115], vulnerability analysis on different platforms — web [229, 92, 141, 109] and Java [123], as well as different types of problematic code (*e.g.,* race conditions detection [224]).

We compare the performance between VulDeePecker [206], Devign [48], Vuddy [100] and VulDetector in Table 3.4. Overall, VulDetector outperforms other tools by 57% in F-1 scores. Note that, we also investigate off-the-shelves symbolic engines such as KLEE [22] on our test software, but it fails to converge even after keeping it running for 30 hours.

VulDeePecker performs poorly on our dataset with an average F1-Score of 18%, which is likely due to the model's inability to handle complex data structures. Devign had a slightly better performance with an average F1-Score of 32.7%, which is still not satisfactory. Vuddy, although it identifies vulnerable code clones, suffered from mispredictions due to missing snapshots of vulnerable code in its database. As the database is not updated, Vuddy performs poorly in detecting vulnerabilities in all the benchmarks except TensorFlow. Even for TensorFlow, our tool outperforms Vuddy.

In vulnerability analysis, we need to ensure a low false positive and false negative rate. Otherwise, the tool will report too many false alarms for engineers to handle or will miss lots of vulnerabilities hidden inside different parts of the software. Compared to the baselines, we observe an improvement of 61%, 57%, 73%, 56%, 54%, 72%, 67% F1-Score in Android, FreeRDP, GPAC, Linux, Pillow, TCPDump, and TensorFlow, respectively. Our tool has both high precision (*i.e,* low false-positive) and high recall (*i.e,* low false-negative), which is important for vulnerability detection.

**RQ2**: VulDetector outperforms state-of-the-art analyzers in terms of detecting vulnerable code.

### 3.4.4 Ablation Study

To understand the impact of our machine learning model and symbolic engine, we measure the performance with and without the symbolic engine. First, we use our machine learning model to detect vulnerabilities in all seven software. We find that our ML model achieves an average of 57% F1-Score across all seven software. Next, we plug the symbolic engine into VulDetector. We observe that overall it achieves 63% F1-Score. We look into the cases in which our symbolic engine could not improve. The root cause is due to the limitation of our k-bounded slices presented in Section 3.3.2. In particular, while the design of k-bounded slices tries to keep the code snippet at the minimum amount of context, in several cases, the slice algorithm may eliminate code that is not directly relevant but useful to infer invariants (*e.g.,* range of variables) that are crucial to obtain high precision.

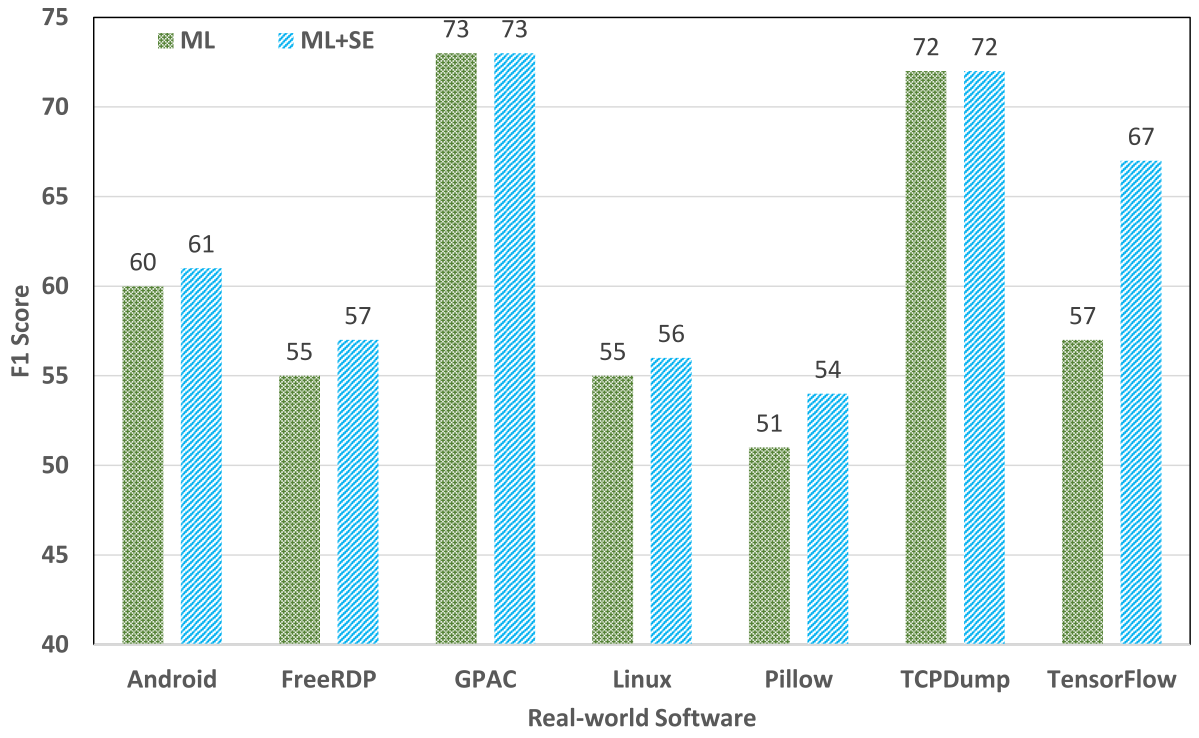


Figure 3.9: Ablation study of VulDetector.

**RQ3**: Our machine learning model and symbolic execution are the key to the efficiency of VulDetector.

## 3.5 Zero-day Vulnerability Detection

To find out whether VulDetector can discover unknown vulnerabilities in real-world software, we conduct an experiment to find zero-day vulnerabilities. In this section, we first describe the process of our zero-day vulnerability detection. Then we summarize the detection results of our tool, including 76 zero-day vulnerabilities in open-source projects with 12 accepted CVEs.

We collect the latest version of this software from their official GitHub website. At the time of running the experiment, 2.6 was the latest version in TensorFlow, and 8.4 was the latest for Pillow. Later, we use our tool to detect vulnerable code. We collect the slices which are predicted as vulnerable by our tool. We manually investigate those vulnerable slices to find the problematic operations. Once we suspect problematic behavior, we manually generate the exploit code. To generate the exploit code, we need to find the responsible function manipulation which will trigger the vulnerable characteristics. With the help of official documentation and experience working in the security domain, we generate test exploit codes. Then, we install the software

| **CVE** | **Type** | **CWE** | **CVSS** |
|---|---|---|---|
| CVE-2021-41208 | Buffer Overflow | CWE-824 | 9.3 |
| CVE-2022-21736 | Buffer Overflow | CWE-476 | 7.6 |
| CVE-2022-21740 | Buffer Overflow | CWE-787 | 7.6 |
| CVE-2022-21735 | Divide By Zero | CWE-369 | 6.5 |
| CVE-2022-23567 | Integer Overflow | CWE-190 | 6.5 |
| CVE-2022-21739 | Buffer Overflow | CWE-476 | 6.5 |
| CVE-2022-23568 | Integer Overflow | CWE-190 | 6.5 |
| CVE-2022-23569 | Buffer Overflow | CWE-617 | 6.5 |
| CVE-2022-21734 | Buffer Overflow | CWE-843 | 6.5 |
| CVE-2022-21737 | Buffer Overflow | CWE-754 | 6.5 |
| CVE-2022-21738 | Integer Overflow | CWE-190 | 6.5 |
| CVE-2021-29584 | Integer Overflow | CWE-190 | 6.5 |

Table 3.5: 12 accepted CVEs discovered by VulDetector.

(*e.g.,* TensorFlow 2.6, Pillow 8.4) on our local machine, and execute those exploit codes. Once the exploit is successful, we write a vulnerability report along with the proof-of-concept code, logs and screenshots, vulnerable type, and possible ways to fix the vulnerabilities. We immediately report the problem to the security team and offer them additional help if required to fix the vulnerability.

**Summary of the result.** Following the process above, we have found 64 vulnerabilities in TensorFlow and reported all of them to the TensorFlow security team (maintained by Google LLC) for further analysis. They have confirmed 59 zero-day vulnerabilities. Out of 59, other researchers reported 12 vulnerabilities at the same time when we reported those. That leaves us with 47 zero-day vulnerabilities that no one has reported before. They have assigned 12 CVEs (seven buffer overflow, four integer overflow, and one divide by zero) for our findings. The rest 35 vulnerabilities are similar to those 12 CVEs. We have listed those in Table 3.5. Here, we have included the Common Vulnerability Scoring System (CVSS) score for each of those vulnerabilities. CVSS refers to a numerical (0-10) representation of the severity of the vulnerability [39]. NVD provides three severity ratings based on CVSS. They are— low (0-3.9), medium (4.0-6.9), and high (7.0-10.0). We can observe that the top three accepted CVEs receive CVSS scores of 9.3, 7.6, and 7.6, respectively.

In addition to TensorFlow, we manually verify new vulnerabilities identified by VulDetector in other software. We have manually verified 17 other zero-day vulnerabilities related to buffer overflow, heap overflow, and integer overflow in the other six software. The breakdown of these zero-day vulnerabilities is as— Android: 4, FreeRDP: 3, GPAC: 1, Linux: 4, Pillow: 3, TCPDump: 2. We have reported our findings to vendors, and our reports are still under review and we are actively communicating with the vendors.

**Key insights.** We summarize the key insights that we get from analyzing zero-day vulnerabilities. We believe this will help developers in the future to build vulnerable free software.

```
1 OpInputList stats_summary_list;
2 OP_REQUIRES_OK(context, context->input_list("stats_summary_list",
  ↪  &stats_summary_list));
3 const int64_t num_buckets = stats_summary_list[0].dim_size(1);
4 // Check for single logit: 1 gradient + 1 hessian value.
5 DCHECK_EQ(stats_summary_list[0].dim_size(2), 2);
```

Listing 4: Incomplete validation in boosted trees code.

**A. Incorrect Type & Size.** Different functions and APIs expect a specific type and size of variables. However, sometimes developers forget to validate those before passing the data. For example, some functions expect a parameter to be an 'int32' type or within the range of 'int32', but developers pass 'int64' type values which causes overflow.

**B. Null Value.** Another common type of vulnerability that we find is missing the check of *null* values. For several functions, they assume that input data from the user will contain a non-null value. However, if the user does not provide those values, then it will cause an assertion failure error.

**C. Divide by Zero.** We find evidence of divide by zero error in real-world software. For example, while computing tensors it is very common to use divide operations. But in some functions, they have not checked the value of the divisor. As a result, it causes a floating-point exception (*i.e.,* zero-division error).

**Case Study.** In this section, we discuss interesting vulnerabilities which are detected by VulDetector.

**CVE-2021-41208: Incomplete validation in boosted trees code.** VulDetector finds buffer overflow vulnerable code with a CVSS score of 9.3 (out of 10). We find the vulnerability in 'stats_ops.cc' file of TensorFlow (v-2.6). We list the corresponding vulnerable code in Listing 4. The key issue for the vulnerability is missing validation in boosted trees operation. So it ends in a 'CHECK' failure in Line 5 of Listing 4. It expects 'stats_summary_list' to be a 2-dimensional tensor. Once we verify this vulnerability manually, then we craft the exploit code by providing 'hessians' to be a 1-dimensional tensor (as listed in Line 4 of Listing 5). In this way, we can validate the vulnerability and generate the exploit code. This vulnerability leads to triggering a denial of service (via dereferencing of 'nullptr' or 'CHECK' failures). Thus, attackers can perform read or write operations from heap buffers.

Once we reported this vulnerability to the TensorFlow security team, they acknowledged our findings within a day and were able to reproduce the vulnerability. Then they started to work on fixing the vulnerabilities. After three months they provided a patch for this vulnerable code by introducing validation of several tensor

```
tf.raw_ops.BoostedTreesSparseAggregateStats(
    node_ids=[1],
    gradients=[1.0],
     hessians=[1.0],
    feature_indices=[1], feature_values=[1],
    feature_shape=[1,1],
    max_splits=1, num_buckets=1, name=None
)
```

Listing 5: Proof-of-concept code for CVE 2021-41208.

```
const int64_t num_buckets = stats_summary_list[0].dim_size(1);
// Check for single logit: 1 gradient + 1 hessian value.
OP_REQUIRES(context, stats_summary_list[0].dim_size(2) == 2, errors::InvalidArgument("stats_summary_list[0]
↪  must have exactly 2 dimensions, obtained: ",                       stats_summary_list[0].dim_size(2)));
```

Listing 6: Patch code for CVE 2021-41208.

shapes in the corresponding file (‘stats_ops.cc’). We have added the patch code for this vulnerability in Listing 6 which is also predicted as benign by VulDetector.

**CVE-2021-29584: CHECK-fail due to integer overflow** VulDetector generates multiple *ProgramSlices* for *sparse_split_op.cc* file. We have listed the code in Listing 7 which our tool predicts is a vulnerable code. The hotspot is situated at Line 2 of the Listing 7. Here, it creates a *SparseTensor* with the given argument values.

We craft the exploit code manually. We find the exploit value (*e.g.,* $2^{60} - 1$) for which it may trigger an integer overflow. After that, we search the TensorFlow documentation and find the Python wrapper function which is *tf.raw_ops.SparseSplit*(). We have listed the exploit code in Listing 8. Then we provide exploit value to this function. Here, we are providing the exploit variable as ‘shape’ value. An attacker could launch a denial of service attack via this overflow while creating a new tensor shape. We reported this problem as soon as we discovered it. TensorFlow security team respond to us within a week and they provided the fix after three weeks.

We list the patch code in Listing 9. Previously, a dense shape was constructed without checking its dimensions, which would cause overflow. Later ‘CHECK’ operations were added to the constructor. Now, when creating ‘SparseTensor’ it uses *BuildTensorShapeBase* or *AddDimWithStatus* to prevent CHECK-failures in the presence of such overflows.

```
1   sparse::SparseTensor sparse_tensor;
2  OP_REQUIRES_OK(context,sparse::SparseTensor::Create (input_indices,input_values,
  ↪  TensorShape(input_shape.vec<int64>()), &sparse_tensor));
```

Listing 7: Executing ‘SparseSplit’ function in TensorFlow.

```
val_exp=2**{60}-1
sparse_data= tf.raw_ops.SparseSplit(
 split_dim= 1,
 indices= [(0, 0),(0, 1),(0, 2),(4, 3), (5, 0), (5, 1)],
 values= [1.0, 1.0, 1.0, 1.0, 1.0, 1.0],
 🐞 shape= (val_exp, val_exp),
 num_split= 2,
 name= None
)
```

Listing 8: Proof-of-concept code for CVE-2021-29584.

```
TensorShape dense_shape;
const auto input_shape_flat = input_shape.flat<int64>();
for (int i=0; i<input_shape.NumElements(); i++) {
  OP_REQUIRES_OK(context,
  dense_shape.AddDimWithStatus(input_shape_flat(i)));
}
sparse::SparseTensor sparse_tensor;
OP_REQUIRES_OK(context, sparse::SparseTensor::Create
(input_indices, input_values, dense_shape,  &sparse_tensor));
```

Listing 9: Patch code for CVE-2021-29584.

## 3.6 Discussion

Next, we discuss ethical considerations for vulnerable disclosure and the limitations of our tool.

**Ethical Consideration.**

By exploiting the vulnerabilities, attackers can take control of certain operations or the whole system. That's why we need to make sure the confidentiality of the vulnerability disclosure until the software provides a fix. We have responsibly let all the manufacturers know about our findings through private channels and give them 90 days for patching those vulnerabilities.

**Limitations.** Like any other program analysis tool, VulDetector also has a few limitations.

First, we require human involvement to generate the exploit code. An automatic exploit generation tool will be very useful. This will be interesting future work. Second, our tool only analyzes vulnerabilities in the C/C++ programming language. Generalizing it across different programming language platforms will be very useful. Finally, although our performance is significantly higher than the baselines, there are still a small amount of false positive cases and false negative cases. How to further eliminate these cases is an interesting question.

## 3.7 Related Work

**Static Analysis for Vulnerability Detection** Prior works [45, 43, 65, 190] introduced tools for software vulnerability detection using static analysis. Themis [31] combines automatic reasoning and static analysis to detect side-channel vulnerabilities. *ITS*4 [85], a lexical-based tool, maintains a codebase and identifies vulnerability by comparing the target codebase and its own database. UNO [78] detects software defects related to the undefined variable, null pointer reference, and array out of index via user-defined properties. Existing works focused on discovering vulnerable patterns use human experts or semi-automated methods (*e.g.,* [168, 42, 66, 30, 38, 209, 228, 11, 227, 74]. However, pattern-based approaches lack coverage and have high false-negative rates, failing to precisely locate vulnerabilities. It also requires expert knowledge and is hard to transfer to other software/languages. A technique capable of bridging the semantic gap across software is needed.

Code analysis [123, 235, 187, 192, 92, 53, 211] is widely used to detect vulnerable code. Cloning vulnerable code can preserve the same vulnerability [106]. To find similar vulnerable code, researchers propose tools with different levels of granularity, such as, tokens [173, 93], trees [28, 89, 160, 226], or graph [107]. Studies show the effectiveness of combining program analysis with graph representation to model variable lifecycles [63, 101, 212]. Yamaguchi et al. introduced a code property graph to detect common vulnerabilities in C/C++ programs [229]. Researchers then extended this to different programming languages to identify vulnerabilities [12, 161, 109]. Our approach uses a K-bounded slice to preserve contextual information for finding vulnerable code, instead of a naive slice that can result in a large graph.

**Dynamic Analysis for Vulnerability Detection** Dynamic analysis techniques (such as, fuzz testing [68, 77, 72, 217] and dynamic taint tracking [143, 211]) have been used for vulnerability detection. Some studies combine static and dynamic analysis [203], while others use an abstract model of the application [33]. However, these methods are expensive and limited by the scalability of starting from program entry points [236]. Our approach uses K-bounded slices to overcome the issue of infinite edges in the program dependence graph.

**Machine Learning for Vulnerability Detection** Machine learning-based tools [81, 117, 116, 210, 46, 243, 113, 128] have been proposed to overcome the disadvantages of traditional vulnerability detection tools. Studies have explored their effectiveness [29] and application in multi-class vulnerability detection [242], Researchers have explored combining expert knowledge with graph neural networks [122, 121] and transfer learning [144]. However, none of these can find vulnerabilities in complex data structures or zero-day vulnerabilities in real-world software.

**Symbolic Execution** Researchers use symbolic execution to solve tasks like finding vulnerabilities and exploit generation [22, 27, 72, 154, 153, 194, 10]. However, existing works suffer from path explosion and coverage of search space. Researchers use different methods to overcome this, such as directed symbolic execution [126], constraint solving [185, 214, 125], domain-specific search [213, 224], selective symbolic execution [32]) to overcome this challenge. Chopper avoids uninteresting parts of the program [199] while Parvez et al. uses a best-first strategy [152]. To run the symbolic engine successfully, we need to provide the executable source code to the symbolic executor. So, we restrict the exploration paths within the variable's usage from the outputted slices from the ML model.

## 3.8 Summary

In this chapter, we present VulDetector, a novel tool that can discover crucial vulnerabilities in large open-source libraries. To efficiently detect vulnerabilities in large codebases, VulDetector employs a hybrid approach that combines machine learning with a specialized symbolic engine to reason about customized abstract domains. To demonstrate the effectiveness of VulDetector, we evaluate it on seven popular libraries with millions of downloads. Among these high-profile libraries, VulDetector has discovered 76 zero-day vulnerabilities (59 of them are acknowledged by Google LLC), 12 of which led to new CVEs.

# Chapter 4

# Checking Privacy Policy Compliance by Performing Cross-language Code Analysis

This chapter tackles the second challenge highlighted in Chapter 1: cross-language analysis. It explores this challenge in the context of web applications, specifically WordPress plugins (used to add functionalities to the website). With WordPress plugins incorporating interaction among multiple programming languages—HTML, JavaScript, PHP, and SQL—it presents a significant challenge to conducting a thorough analysis. We select GDPR (General Data Protection Regulation) compliance checking as an example to gauge the effectiveness of our framework, as evaluating GDPR compliance requires analyzing the comprehensive information flow through these four languages. While previous studies [176, 174, 41, 201] have undertaken partial verification of GDPR compliance, none have succeeded in conducting a complete analysis due to such complex interaction among those programming languages. To address this, we introduce a new cross-language code property graph (CCPG), designed to capture information flow across various programming languages in a web application. In this chapter, we develop ChkPlug, the first automated tool for checking the GDPR compliance of websites. By employing our tool, we model the GDPR policy to assess the compliance of web applications.

This chapter is based on the publication in the Proceedings of Network and Distributed System Security (NDSS) Symposium (NDSS) 2023, titled "CHKPLUG: Checking GDPR Compliance of WordPress Plugins via Cross-language Code Property Graph" ([182]).

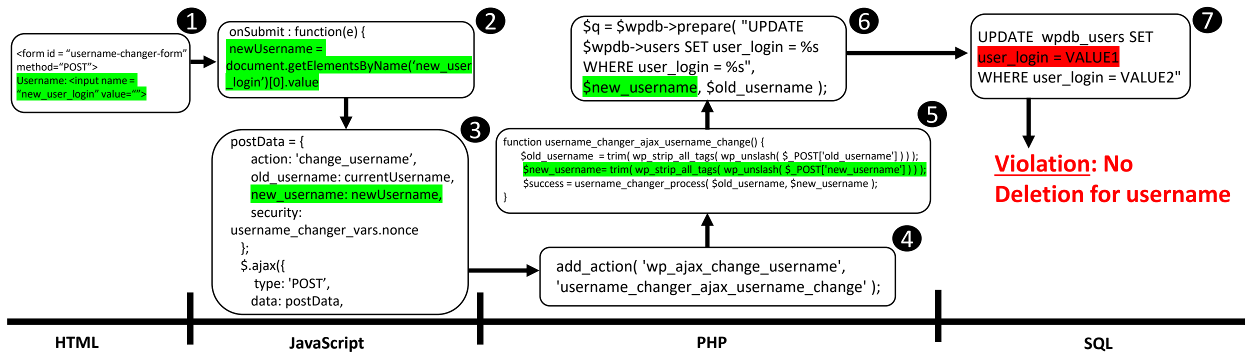
## 4.1 Introduction

WordPress, a well-known content management system (CMS), provides so-called plugins and themes—usually developed by third parties for website owners to augment the default functionalities of the CMS. To date, WordPress has around 60K plugins [55] and they generate over a billion dollars of revenue each year [83]. One challenging problem of using WordPress plugins is that they may collect and process personally identifiable information (PII), which is regulated by privacy regulations such as the European Union-introduced GDPR. These plugins are published in the global application market and can be accessed or integrated by anyone around the world using the WordPress store. That is unless they specifically block EU traffic, it is the default assumption that they will have potential European Union traffic [20] and need to obey GDPR. An existing article [20] shows that it is the site owner's responsibility to ensure all the plugins installed on their website follow GDPR. Compliance with GDPR will help plugin developers to increase possible installations of their plugins. Plugin developers often are not familiar with privacy regulations and therefore the need for an automatic checker is increasing for both developers and website owners.

To the best of our knowledge, there are no prior works that check the compliance of WordPress plugins against different GDPR articles related to private data. On one hand, researchers are studying the compliance of websites against certain GDPR articles, such as cookie and tracking opt-outs [176, 174]. Similarly, a multitude of free and paid services exist to evaluate the GDPR compliance of given websites and vary in complexity, ranging from consulting to cookie analyzers [41, 1, 179] to do-it-yourself checklists [201, 37]. However, these only cover a specific subset of GDPR requirements such as, cookie consent, or involve slow or expensive manual review [24, 186].

On the other hand, prior works investigated GDPR compliance in mobile application markets by checking the presence of user consent [146, 202, 147], privacy policies [7, 6, 118, 197, 239, 240], cookies [47, 18], and by analyzing network traffic data [87, 64]. While such works are important and successful in checking client-side GDPR compliance, unfortunately, they cannot be extended to WordPress plugins where PIIs are often collected on the client side using HTML and JavaScript, but processed at the server via PHP and SQL. That is, personally identifiable information is flowing between the client and server and is processed heterogeneously across the program language boundaries.

Such cross-language dataflows are challenging to identify, let alone be used for detecting GDPR violations because of the variety and involvement of different program languages. For example, JavaScript code is interacting with HTML using Document Object Model (DOM) APIs, but with the server-side PHP using asynchronous HTTP requests and responses. At the same time, PHP is interacting with HTML and JavaScript



```
<form id = "username-changer-form" method="POST">
Username: <input name = "new_user_login" value="">
```

```
onSubmit : function(e) {
newUsername =
document.getElementsByName('new_user_login')[0].value
```

```
postData = {
    action: 'change_username',
    old_username: currentUsername,
    new_username: newUsername,
    security:
username_changer_vars.nonce
  };
  $.ajax({
    type: 'POST',
    data: postData,
```

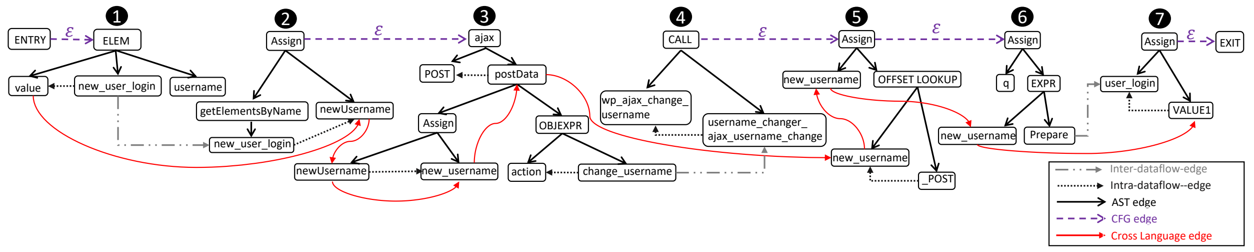
```
add_action( 'wp_ajax_change_username',
'username_changer_ajax_username_change' );
```

```
function username_changer_ajax_username_change() {
    $old_username  = trim( wp_strip_all_tags( wp_unslash( $_POST['old_username'] ) ) );
    $new_username= trim( wp_strip_all_tags( wp_unslash( $_POST['new_username'] ) ) );
    $success = username_changer_process( $old_username, $new_username );
}
```

```
$q = $wpdb->prepare( "UPDATE
$wpdb->users SET user_login = %s
WHERE user_login = %s",
$new_username, $old_username );
```

```
UPDATE  wpdb_users SET
user_login = VALUE1
WHERE user_login = VALUE2"
```

(a) An illustration of the GDPR violation using code from different languages.



(b) Cross-language Code Property Graph Representation of the source code presented in Figure 4.1a.

Figure 4.1: Motivating example for detecting data deletion violation in WordPress plugin.

using statements, such as 'echo', but with JavaScript through HTTP requests and responses as well. Such heterogeneous dataflows make it challenging to extract PII and track its flow across languages.

In this chapter, we design and implement ChkPlug, the first automated GDPR checker of WordPress plugins. ChkPlug identifies compliance with GDPR articles related to PIIs and processing of PIIs (*i.e.,* data access, data deletion, data sharing, and Security of PII) [1]. We select these GDPR articles based on the following criterion: (1) common articles that are shared by other privacy laws, e.g., California Consumer Privacy Act (CCPA) [23], and (2) articles that are directly relevant to data processing via computer programs. The key insight of ChkPlug is to match WordPress plugin behaviors—represented as data- and control flows and extracted via cross-language static analysis—against a set of predefined rules of GDPR articles. ChkPlug reports a violation if a match is not found for a GDPR article. As an example, consider data deletion. ChkPlug checks the lack of data flow between PIIs and deletion APIs as a violation of the data deletion article. Similarly, consider data sharing. ChkPlug looks for a privacy policy when PIIs are shared with a third party via network communication such as HTTP requests.

ChkPlug tackles and advances the cross-language challenge by representing WordPress plugins involving four different languages (HTML, JavaScript, PHP, and SQL) as a cross-language code property graph

[1]In this chapter, we refer to data access as $P_{access}$, data deletion as $P_{delete}$, data sharing as $P_{share}$, and security of PII as $P_{security}$

structure. That is, ChkPlug first generates intra-language graphs (*e.g.,* CPGs and DOM trees) and then connects different graphs across language boundaries into a CCPG. For example, ChkPlug connects HTML DOM nodes with JavaScript variables using key-value analysis, i.e., if the HTML node ID matches the JavaScript DOM function parameter. For another example, ChkPlug performs an inline analysis to generate placeholders for PHP code embedded as part of HTML and creates dataflows.

Once ChkPlug generates CCPG, it queries the CCPG starting from PIIs and tracks the PIIs flows for GDPR violations. Ground-truth evaluation on 200 plugins shows that our tool achieves a 98.8% TNR (true negative rate) and an 89.3% TPR (true positive rate) for detecting violations of $P_{access}$, $P_{delete}$, $P_{share}$ and $P_{security}$.

To understand the current compliance situation of plugins, we perform a measurement analysis and run ChkPlug on 2,722 plugins from the marketplace (Section 4.5). ChkPlug determined 381 plugins (14%) to be non-compliant with GDPR regulations (Section 4.6.3). Among them, 368 plugins violate $P_{delete}$, 19 plugins violate $P_{share}$, and 36 plugins violate $P_{security}$. We find no violations for $P_{access}$ because most of the time the plugins store PII data using WordPress core database and the rest of the time they provide export functionality.

## 4.2 A Motivating Example

In this section, we provide a motivating example that is found by ChkPlug to be in violation of GDPR Article 17, i.e., the right to erase (a.k.a. right to be forgotten). The violation is located in a popular WordPress plugin, called Username Changer[2], which has an active installation of 30,000+ users and an average rating of 4.5 stars. The plugin provides an easy way to change usernames via a nice interface. Note that we have responsibly reported the GDPR violation to the plugin's developer, but have not received any feedback yet at the time of our paper submission.

Figure 4.1 (a) shows how this WordPress plugin collects and stores private data but does not provide any deletion methods. The collection has four steps and spans four different programming languages. First, the plugin collects private user inputs (*e.g.,*username, display name, and nickname) via an HTML form with an id "username-changer-form". The collected data is considered as Personal Identifiable Information (PII) according to the literature [231]. Second, the JavaScript code of the plugin reads the data using DOM operations and sends the data via a 'POST' request to the server-side plugin. Third, the server-side code, written in PHP, obtains the data and prepares SQL statements. Lastly, the SQL statements store the private

[2]`https://wordpress.org/plugins/username-changer/`

data in the database. This collection and storage is a violation of GDPR's right to erase because the plugin does not provide a means to delete the data. Instead, the data is stored in the server database after being collected.

The major challenges in detecting this GDPR violation are the detection of PII collection and the lack of deletion of stored PII: The combination of these two conditions leads to such a violation. The PII collection detection is challenging because the private data flows across different language boundaries. That is, the dataflows between languages are heterogeneous: For example, HTML→JavaScript is based on a DOM operation, and JavaScript→PHP is based on HTTP requests. Furthermore, there are others that are not shown in the example, such as PHP→JavaScript via the echo statement. Similarly, the detection of a lack of deletion is challenging because a deletion may exist across different languages and be controlled by a user on the client side.

**Solution Overview** Succinctly, ChkPlug models WordPress plugins as a cross-language code property graph and violations of GDPR clauses as graph queries. Then, ChkPlug reports a GDPR violation if a match is found in the graph.

Figure 4.1 (b) shows the CCPG of our motivating example. There are three different types of inter-language dataflows that are represented in the graph. First, the data flow crosses from the HTML form node (i.e., the value of the "input" node with a name as "new_user_login") to the return value of a JavaScript function call, `getElementsByName`. Second, the data flow crosses from JavaScript to PHP. Specifically, when JavaScript calls `$.ajax` with a key-value pair `new_username:newUsername`, the PHP code receives the data via `$POST`. ChkPlug connects the JavaScript CPG node with the corresponding PHP CPG node based on the key "new_username". Lastly, the data crosses from PHP to SQL. ChkPlug connects two nodes, one in the PHP CPG and the other in the parsed SQL statement, via the `prepare` statement in PHP.

Once ChkPlug creates a CCPG such as the one shown in Figure 4.1 (b), the next step is to query it for GDPR violations. Such a query might follow a pattern like the one shown in Figure 4.2. Precisely, the query has two parts. The top shows that ChkPlug finds a PII coming from an HTML input field and being stored in the server-side database. Then, the bottom shows that ChkPlug finds that the deletion request is not connected with the database deletion, i.e., the PII is kept in the server-side database, leading to a violation of GDPR's right to erase in Article 17. Note that the top is a prerequisite of the bottom: If PII is not stored, ChkPlug will not look for deletion and possible GDPR violations.

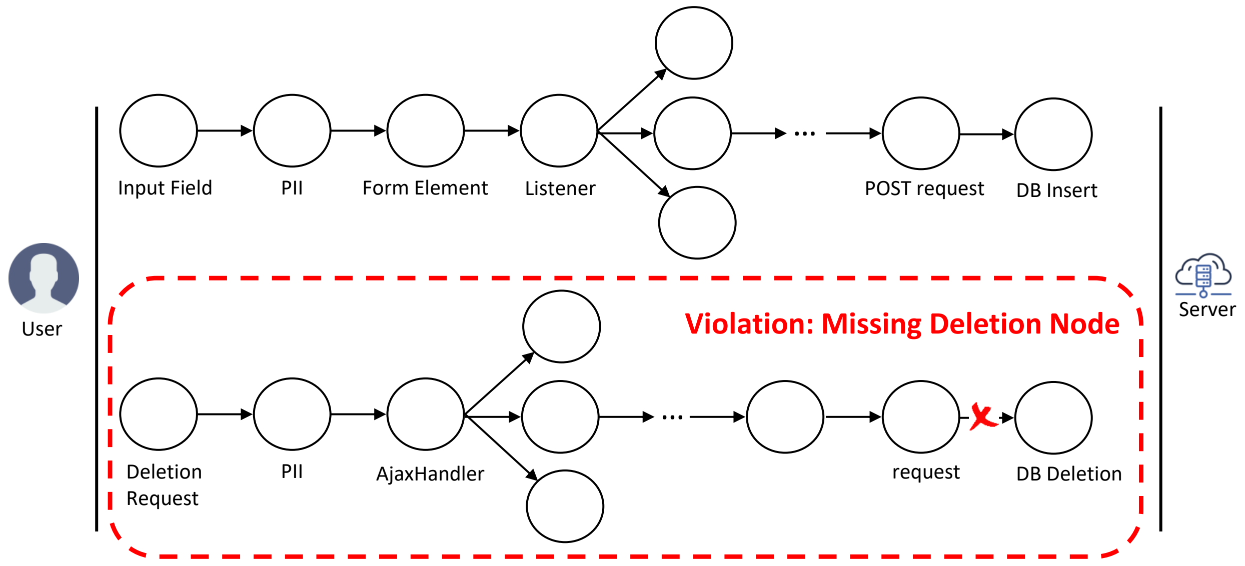


Figure 4.2: Queries used for identifying data deletion violation.

## 4.3 Related Work

In this related work section, we start by describing previous works on GDPR analysis. Next, we present existing static analysis tools, particularly those on PHP and JavaScript, and how our tool is different from them. We end with discussing prior works on website privacy analysis.

### 4.3.1 GDPR Analysis

Investigating GDPR violence in published applications is a hot research topic. Researchers are performing experiments to detect whether the application includes proper privacy policy [7], and whether they take user consent before storing user personal information [202]. To that end, they run several measurement analyses to have a deep understanding of the impact of GDPR across different regions [75] and how it evolves the existing application market [118]. A wide body of research works [94, 129, 200, 177] already discovered many problematic ways of granting cookie consent. Nouwens et al. [147] investigated 680 websites and found 90% of them do not comply with GDPR requirements. They pointed out several problems including – vague privacy policies [6], and over-sharing of information through third parties [138]. In sum, the existing works are focusing more on making the usage of personal information more transparent [120, 133, 197, 202, 208, 183], which typically aim at generating GDPR compliant privacy policies from application's source codes [232, 239].

However, those works mainly focus on one platform while doing analysis. In contrast, ChkPlug can detect GDPR violations across different programming languages. Specifically, we introduce an end-to-end detection tool that starts with how the application is collecting data from users, sends those data to the cloud, and how the cloud processes & stores those data.

### 4.3.2 Program Analysis

Program analysis, particularly either static or dynamic analysis, is a widely studied topic, and it likewise has been applied to a variety of situations, including but not limited to– code correctness, sanity checking, security, and privacy [12]. JFlow is an early application of static analysis to perform security and privacy for Java code [136]. Kim et al. [99] developed ScanDal, which statically analyzes Android applications for privacy leaks via tracking data flows from sinks to sources. AndroRisk is a risk quantifier for Android applications based on their requested device permissions along with dependent libraries [178]. Shezan et al. introduced TKPERM to transfer permission access knowledge across mobile, web, and IoT platforms [184]. Xiao et al. applied static analysis to TouchDevelop scripts for Android, and these scripts were analyzed for private information sources flowing to sinks to outside sources as well as the security and privacy of such information flows [223]. Similar to those works, ChkPlug can track personal information flow from the source to the sink. However, the purpose of ChkPlug is to detect GDPR violations rather than direct privacy leaks or vulnerabilities.

Security researchers use query languages to identify vulnerabilities and other security bugs [76, 155]. They identified SQL injections and cross-site scripting in Java programs [123]. Martin et al. discovered functional flaws and security vulnerabilities using the Program Query language [127]. These days, researchers leverage graph-based program analysis to discover defects in the artifacts and identify zero-day vulnerabilities [5, 90]. Yamaguchi et al. leveraged code property graphs to discover vulnerabilities in the Linux kernel [229]. All of these works are focused on one programming language. Whereas, we create a cross-language code property graph to bridge the connection among different web programming languages.

Jalangi [180] selectively records and replays front- and back-end JavaScript programs. ODGen [110] design and proposes an object dependence graph to detect Node.js vulnerabilities based on graph queries. DAPP [98] looks for AST and control-flow patterns for prototype pollution vulnerability detection. ObjLupAnsys [108] expands and maps two clusters during the abstract interpretation for vulnerability detection. Nodest [145], which is based on TAJS [86], detects command injection vulnerability via skipping unrelated packages. Black Widow [57] crawl and scan web applications in a black-box manner. People also proposed approaches to detect various client-side vulnerabilities, such as DOM-based XSS [130, 193] and CSRF [156]

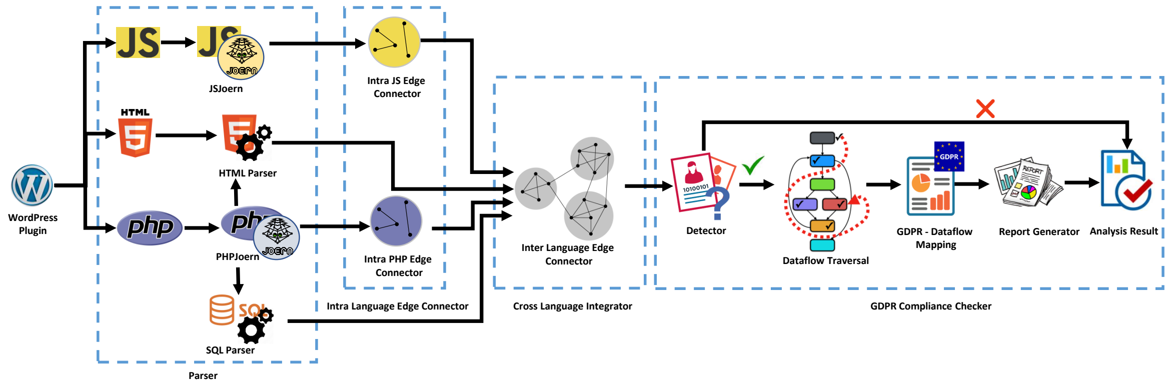


Figure 4.3: The system architecture of ChkPlug.

As a comparison, ChkPlug's static analysis crosses language boundaries, e.g., HTML, JavaScript, PHP, and SQL.

### 4.3.3 Website Privacy Analysis

There exist several privacy enforcement tools, including – privacy preferences [44], privacy badger [13]. They are investigating various ways to improve user protection [103]. To that end, they enforce users' cookie policies on the visited websites by interacting with a consent management system [146], blocking first-party cookies (such as Google Analytics) [18]. Compared to those works, we are analyzing the source code of websites to find potential privacy leakages.

Most of the current work on WordPress plugins focuses on measuring the vulnerabilities [170, 132, 26, 21], characterizing vulnerability exploits [104, 25, 237, 54]. Kasturi et al. investigate the impact of malicious plugins on CMS marketplaces [96]. Whereas, in our work, we investigate WordPress plugins to identify GDPR violations.

## 4.4 System Design

We introduce the design of ChkPlug in this section.

### 4.4.1 Overall System Architecture

Figure 4.3 shows the overall system architecture of ChkPlug, which takes the source code of the WordPress plugin as input and outputs a list of GDPR violations. ChkPlug has three major components: (1) parser, (2) CCPG generator including (2.1) intra-language edge connector and (2.2) cross-language integrator, and (3) GDPR compliance checker.

Here are the detailed descriptions of the three components. First, the parser is to detect and parses language-specific source code. Since the parser is standard and we do not claim any contributions, we leave the descriptions to Section 4.5.1. Second, ChkPlug generates CCPG via two steps: (i) intra-language CPG generation (especially event-related control- and data-flows), and (ii) cross-language CCPG generation. Lastly, ChkPlug traverses all paths involving PII operations to identify the requirements according to GDPR policy. For example, if a plugin stores data in the database, it must provide data deletion functionality. Here, we map all of the requirements for GDPR policies along with the generated graph using our GDPR compliance checker module (Section 4.4.3). We conclude with the analysis result listing whether a particular plugin violates GDPR.

### 4.4.2 CCPG Generation

In this subsection, we describe how ChkPlug generates CCPG via both inter- (i.e., cross-) and intra-language analysis.

**Cross-language Analysis**

ChkPlug's cross-language analysis depends on different language pairs and adds customized intra-language dataflow edges. From a high level, cross-language analysis can be classified into two major categories: (i) inline analysis and (ii) key-value analysis. Let us explain these two. First, inline analysis is based on the fact that the statements of one language are embedded into another language. For example, PHP can be embedded as part of an HTML file and outputs HTML via the "echo" statement. Similarly, PHP can prepare SQL statements and then execute prepared SQL statements with customized values. ChkPlug performs inline analysis by first analyzing the embedded statements (*e.g.,* PHP in the PHP-HTML case). Then, ChkPlug replaces the outputs of the embedded language statements with pseudo placeholders and then analyzes the latter language (*e.g.,* HTML in PHP-HTML case) to generate intra-language dataflows.

Second, key-value analysis is for these program languages that are connected via APIs (*e.g.,* HTML and JavaScript) or network protocols (*e.g.,* JavaScript and PHP). For example, when JavaScript sends an HTTP POST/GET request to the server, the server-side PHP code receives the values in either `$POST` or `$GET` variables. ChkPlug performs key-value analysis via first matching keys at both languages and then connecting the corresponding values with inter-language dataflows. For example, ChkPlug looks for the keys in `$ajax` calls at client-side JavaScript and matches the corresponding keys in `$POST` at PHP, and then creates dataflows between both values.

```
<p> a <?php echo 'dog' ?> </p>
```

Listing 10: PHP code embedded inside HTML.

Now, we illustrate several language pairs and describe how ChkPlug performs the aforementioned inline and key-value analysis to create cross-language dataflows using a few examples under each pair.

**HTML⟷PHP** On one hand, PHP can be embedded as part of HTML and output HTML code; on the other hand, HTML can be used to send HTTP requests to PHP. ChkPlug adopts inline analysis for the former and key-value analysis for the latter. We describe the details below:

- Inline PHP Traversal (PHP→HTML). ChkPlug connects dataflow edges from the AST of the inline PHP to the corresponding placeholder that represents the PHP output. For example, in Listing 10, ChkPlug connects the root node that corresponds to *echo 'dog'* to a corresponding text node in HTML that represents <*? php echo 'dog' ?*>.

- HTML Form Submission (HTML→PHP). A common type of communication between HTML and PHP is via form submission, in which the input data in a form is submitted and is handled by a `POST`, `GET`, or `REQUEST` variable in PHP. ChkPlug connects dataflow edges between every input field (*e.g.,* <input name = "foo" .../>) in HTML and places the input field that is referenced in PHP (*e.g.,* $_POST['foo']).

**PHP⟷SQL** Data can be passed from PHP code to a SQL query execution that stores data into a database (*e.g.,* through an INSERT statement); then, similar to SQL data storage, data can also be retrieved from databases via SQL queries, and then the data is received by PHP code (*e.g.,* a variable). ChkPlug performs these two types of inline analysis.

- SQL Data Storage (PHP→SQL). ChkPlug connects dataflows from data supplied to SQL query function call in PHP to the SQL AST for data storage SQL queries (*e.g.,* connect a dataflow edge from data that is passed to the INSERT statement to the SQL AST for the INSERT statement).

- SQL Data Retrieval (SQL→PHP). ChkPlug connects dataflows from the SQL AST to the PHP endpoint that receives the query result for data retrieval queries (*e.g.,* connect a dataflow edge from the SQL AST of a SELECT statement to the PHP variable that receives the queried result).

**PHP⟷JavaScript** PHP can be interpreted to yield JavaScript just as it can be interpreted to yield valid HTML. This allows data contained in PHP code to propagate to JavaScript expressions. At the same

```
<?php $pass='123';?>
<script>var jspass='<?=pass?>';</script>
```

Listing 11: Connecting PHP and JavaScript.

time, WordPress provides an Ajax Handler that receives Ajax requests from JavaScript, processes `$POST` or `$GET`, and then fires a WordPress Action along with the data sent from the Ajax requests. Then, plugins can receive the data in PHP via hooking a PHP handler to the fired Action.

- Inline JavaScript Traversal (PHP→JavaScript). ChkPlug connects edges between the PHP nodes and a placeholder that represents the resultant JavaScript. For instance, given the code snippet listed in Listing 11, we connect the PHP node for the variable to the node for its usage in JavaScript.
- WordPress Ajax Handler (JavaScript→PHP). ChkPlug models this data transmission logic by first locating Ajax requests that send data to the WordPress Ajax handler endpoint, and then analyzing the actions that are going to be fired by WordPress. Next, ChkPlug analyzes the PHP functions that are hooked as callback functions to the fired actions. With these, ChkPlug matches the data being sent out by the Ajax requests and the data being received in the PHP functions and connects data flow edges between them.

**HTML⟷JavaScript** JavaScript and HTML are communicating with each other via different types of DOM APIs. The connections are done via two types: DOM events and HTML selectors (such as, IDs, names, and class names). Once JavaScript obtains a DOM element, it can either read its data or write to it via properties like `innerHTML`. We now describe these two types.

First, DOM events may trigger JavaScript callbacks and interact with HTML. Let us use submit events as an example. When the onsubmit event of HTML elements is registered in JavaScript, ChkPlug creates dataflow edges between the form node in the HTML and the root node of the handler code.

Second, JavaScript may use selectors, e.g., jQuery or built-in selectors, to obtain HTML elements. Take jQuery selectors for example. ChkPlug parses the selector statements, maps the selector to the HTML AST, locates the selected HTML elements, and next connects data flow from the located HTML elements to the selectors with keys such as ID, names, and class names. After query selecting, ChkPlug also finds all usages of dataset objects and connects the usage to the actual data attributes in the HTML graph.

**Intra-language Analysis**

Our intra-language analysis follows traditional static analysis in generating different control- and data-flow edges. Prior research has shown this and we do not claim any contributions [4, 12]. Below, we introduce two types of control- and data-flow edges that are specific to WordPress or are traditionally challenging to resolve.

- *Hook-related call edges.* WordPress provides an event listening system called *hooks*, where plugins can either trigger events and send out data (through do_action() or apply_filter()) or register callback functions that listen to events and receive data sent out by the event trigger (through add_action() or add_filter()). ChkPlug models hooks by connecting dataflows from the data sent out by the event trigger to parameters of callback functions that are registered to receive such data. In this way, ChkPlug determines the triggered functions and thus traces the personal data flow.

- *Control- and data-flow edges related to object properties.* Method calls like \$obj−>\$foo()) and property accesses like \$obj−>\$var or \$this−>\$var are challenging to resolve because the host object's types are usually unknown due to the dynamic nature of PHP. ChkPlug traces back the constructions of such objects to find their types and checks the official doc comments (*e.g.,* PHPDoc) for functions or class variable declarations.

  There are two steps. First, ChkPlug identifies the parent-child relationship across all classes in the PHP source code. That is after the CPG is constructed, ChkPlug identifies all function calls using \$self and \$parent variables (*e.g.,* \$self::foo()) and connects call edges based on overriding principles. In addition, ChkPlug connects dataflow edges from the definitions of class constants or static variables (*e.g.,* connect edge from const CONSTANT = 1 to \$self::CONSTANT.

  Second, once the class hierarchy is constructed, ChkPlug determines object types and builds edges for method calls and class variables in an iterative process, until results converge and no new edge can be constructed.

- *Value resolver.* We build our resolver component to determine the values of variables that store URLs, HTML scripts, or SQL queries inside PHP. It statically determines the values for variables in case they are assigned values of basic data types (*e.g.,* string, integer, float). Specifically, for a given variable node, ChkPlug first backtracks dataflow edges to all places they are being assigned. After tracing back to a variable's assignment, ChkPlug traverses the AST to identify the data that is being assigned to the variable, and their relationships with each other (*e.g.,* they are concatenated together). So

```
$foo = esc_html('<p>hello</p>');
$bar = 'a' . 'b' . $foo;
```

Listing 12: Resolving value using value resolver.

that ChkPlug can resolve the complete value for that variable. Furthermore, ChkPlug captures the behaviors of string filtering and sanitization functions provided by WordPress (*e.g.,* esc_html(), $wpdb—>$prepare()) to accurately resolve string values. However, if part of a variable's value does not have any resolvable value (*e.g.,* the value comes from a database), ChkPlug uses a dummy value instead to complete the resolving process.

To better illustrate, let us consider the example in Listing 12. If the program tries to resolve the value for $bar, it will trace back to its assignment, and find that the value is the result of a concatenation among 'a', 'b', and '<p>hello</p>' (the value of $\$foo$ after filtered through esc_html()). Therefore, the resolved value is 'ab<p>hello></p>'.

- *Object type determination.* ChkPlug statically determines the class/type of an object. ChkPlug first produces a representation of all classes and their method calls and class variables. Then it parses PHPDoc comments and strict typing for method declarations and variable declarations. While determining the type of an object, ChkPlug first backtracks to a node with types already recorded, and then checks the recorded type. However, if this process fails, ChkPlug also attempts to backtrack to where the object is being created (*e.g.,* $\$obj$ = new Foo();) or initialized to determine its type. In this way, it obtains the types for class variables, arguments, and returns values of methods.

### 4.4.3 GDPR Compliance Checker

Once ChkPlug finishes constructing the CCPG, it starts to gather information to identify GDPR compliance. Specifically, we divide the GDPR compliance checker into three connected sub-components: (i) detectors, (ii) graph query for GDPR violation, and (iii) violation report generator.

#### Detector

The purpose of the detector is to find three types of nodes: source, sink, and security. *Source nodes* are those that include PII collection/retrieval functionalities. *SECURE* nodes are those intermediate nodes that run security/hashing mechanisms on the inputted data. *Sink node* are those related to the processing of PII, e.g., those points where personal data is being stored, updated, deleted from the database, or sent to a remote address. Let us describe the detection of these three types of nodes below.

Table 4.1: Retrieving personal data using Regex.

| **PII** | **Regex** |
|---|---|
| email | .*(email).* |
| first name | .*(first.*name).* |
| last name | .*(last.*name).* |
| password | .*([∧a-zA-Z]pass). |
| address | .*(address). |
| country | .*(country).* |
| state | .*([∧a-zA-Z]state). |
| zipcode | .*(zipcode). |
| postcode | .*(postcode).* |
| city | .*([∧a-zA-Z]city).* |
| birth | .*([∧a-zA-Z]birth).* |
| username | .*(user.*name). |
| IP address | .*([∧a-zA-Z]ip(.*address.*—[∧a-zA-Z]—.*addr.*)).* |
| phone | .*(phone).* |

**Source Detector** ChkPlug identifies three different types of source nodes from where personal data can be collected or retrieved i.e., (1) HTML form inputs, (2) WordPress core database, and (3) custom database.

- HTML Form Inputs. The most common way of collecting user information is through a user interface written in HTML. With such an interface, a plugin can collect various personal information, including, but not limited to, a user's email, name, password, location, address, and date of birth. ChkPlug finds those entry points by querying HTML form input nodes and identifying ones with displayed field names that contain personal data keywords using a manually crafted rule-based approach. We list the PII according to the existing article [69, 231]. We devise our rule-based approach for identifying PII in the user interface. Our intuition is that developer will ask PII in the form of an input field. We map the corresponding label with our manually crafted rules. We build this rule set based on investigating different website interfaces (*e.g.,* registration page, login interface) which ask for user input. We have listed all the rules in Table 4.1.

- WordPress core database. Plugins can retrieve personal data from WordPress's core database tables which contain user data and are shared across plugins. For this, ChkPlug considers two cases: (1) the retrieval function directly gets personal data (*e.g.,* get_userdata($user_id) gets the user's profile data such as email and name), or (2) the retrieval function gets user data based on a key that has a matching personal data keyword (*e.g.,* get_user_meta($user_id, 'shipping_address') gets the user's shipping address, which is previously stored as the user's metadata). ChkPlug keeps a record of the argument position of the key for each function based on the WordPress documentation locally in a file for mapping purposes.

Table 4.2: Security Functions.

| **Source** | **Function** | **Type** | **Is state-of-the-art?** |
|---|---|---|---|
| php | crypt | encrypt | no |
| php | md5 | hash | no |
| php | sha1 | hash | no |
| php | password_hash | hash | yes |
| php | openssl_encrypt | encrypt | yes |
| php | openssl_digest | hash | yes |
| php | hash | hash | algorithm dependent |
| php | hash_hmac | hash | algorithm dependent |
| defuse | encrypt | encrypt | yes |
| phpseclib | encrypt | encrypt | yes |
| wordpress | wp_hash_password | hash | yes |

- Custom database. Plugins can retrieve personal data from database queries (*e.g.,* SELECT query) on tables that contain personal data. This database is specific to each plugin and is not shared across other plugins. ChkPlug first scans for all database table creation queries and labels tables with fields that have matching personal data keywords (Table 4.1). Note that ChkPlug will not be able to detect storage where the table's field name does not contain any matching personal data keyword but is used for storing personal data. Next, ChkPlug scans for all database queries that retrieve data from the tables that record personal data.

**Security Detector** GDPR laws require encrypting personal data before sending it over the network (either to first- or third parties). To identify the existence of such functionalities in a plugin's code, ChkPlug analyzes the presence of encryption or hashing functions. We label nodes that perform security operations (encryption or hashing) on personal data as SECURE (see Table 4.2). According to GDPR, one should use state-of-the-art encryption techniques to comply with $P_{security}$. That's why after listing all the encryption algorithms/functions, we mark those which are already broken by existing attacks. For example, at the time of writing this paper, *md5* is not considered to be a safe encryption algorithm. So, during our traversal whenever we find nodes that perform encryption operations on PII, we mark that plugin as $P_{security}$ violation. Note that, the security of hash and hash_hmac depends on how they are implemented. Further analyzing the algorithm is out of the scope of this work. For now, we only consider those as safe.

**Sink Detector** ChkPlug finds four types of sink nodes that perform operations related to (1) WordPress core database, (2) custom database, (3) remote request, and (4) file storage. We discuss those in detail below.

- WordPress core database. Plugins can store, retrieve or delete data from WordPress's core database tables (*e.g.,* add_user_meta() stores metadata for a user.

Table 4.3: Remote Request Functions.

| **Function** | **Language** |
|---|---|
| $.post / jQuery.post | JavaScript |
| $.get / jQuery.get | JavaScript |
| $.ajax / jQuery.ajax | JavaScript |
| curl_exec | PHP |
| wp_remote_post | PHP |
| wp_remote_get | PHP |
| wp_remote_head | PHP |
| wp_remote_request | PHP |

We collect the list of all the WordPress functions from the official website. In total, we investigate 2,885 functions and manually label the type of functionalities of these functions. We find 30 storage functions, 65 retrieval functions, and 28 deletion functions. We provide the full list in the following link- `https://drive.google.com/file/d/1vy98kZGY91IIDlMrEjkBgMBzed1LYene/view?usp=sharing`. We identify and mark all WordPress-provided function calls as sinks because they can potentially process personal data depending on the inputs. In addition, we label the argument inputs of these function calls that are either the data or the key used in the database operation.

- Custom database. Plugins can store or delete data through queries (*e.g.,* INSERT and DELETE statements) on tables that contain personal data. We identify all such database operations and record their operation type, table name, and fields. Furthermore, we backtrack the dataflow edge from SQL to PHP (as explained in Section 4.4.2) to identify the PHP node that passes data to the SQL query and executes the query, and then we mark it as a sink node. For example, in Figure 4.1, *username* information is passed to the SQL query to update the user information.

- Remote request. Plugins send personal data to third parties via remote requests (*e.g.,* curl). There are three ways to send such data in WordPress- (1) PHP native functions, (2) WordPress functions in PHP, and (3) jQuery requests (*e.g.,* jQuery.post()) in JavaScript. We list the functions used for remote requests in Table 4.3. For each such sink point, we identify the argument input that corresponds to the URL. We then perform backward dataflow traversal to resolve possible URLs (Section 4.4.2).

- File storage. In addition to storing data in the WordPress core database or custom databases, plugins can also store data in files. For this, it can use several PHP functions (*e.g.,* using fput()). ChkPlug scans for the usage of PHP native functions for detecting nodes that store any data to files. We follow the PHP documentation to collect all the functions that can be used to set or retrieve data from a file. We list all of those functions and their types in Table 4.4. We label the function used for storing information as 'set'. Whereas, the functions which can be used to get data as 'retrieve'.

Table 4.4: Storing data in file using PHP functions.

| **function** | **action type** |
|---|---|
| fwrite | set |
| file_put_contents | set |
| fputs | set |
| fputcsv | set |
| touch | set |
| fgetc | retrieve |
| fgetcsv | retrieve |
| fgets | retrieve |
| fgetss | retrieve |
| file_get_contents | retrieve |
| file | retrieve |
| fread | retrieve |
| fscanf | retrieve |
| readfile | retrieve |

## Graph Queries for GDPR Violations

The second sub-component of the GDPR compliance checker is to conduct graph queries for GDPR violations based on different GDPR articles. Specifically, Table 4.5 listed the required dataflows that should be present in WordPress to comply with the corresponding GDPR policy. We now describe the details based on different policies separately.

Table 4.5: GDPR policies covered by ChkPlug.

| **ID** | **Action** | **Flow** |
|---|---|---|
| $P_{access}$ | Store | $collect_{HTML}(PII)\|\|retrieve_{DB}(PII) \rightarrow intermediate_{node_1,node_2,....,node_N} \rightarrow storage_{DB}(PII)$ |
| | Export | $counter_{retrieveDB}(PII) \rightarrow intermediate_{node_1,node_2,....,node_N} \rightarrow export_{interface}(PII)$ |
| $P_{delete}$ | Store | $collect_{HTML}(PII)\|\|retrieve_{DB}(PII) \rightarrow intermediate_{node_1,node_2,....,node_N} \rightarrow storage_{DB}(PII)$ |
| | Delete | $counter_{retrieveDB}(PII) \rightarrow intermediate_{node_1,node_2,....,node_N} \rightarrow delete_{interface}(PII)$ |
| $P_{share}$ | Send | $collect_{HTML}(PII)\|\|retrieve_{DB}(PII) \rightarrow intermediate_{node_1,node_2,....,node_N} \rightarrow remote_{request}(PII)$ |
| | Disclosure | $doesExist(privacy - policy)\&\&disclose(sent_{PII}, remote_{URL})$ |
| $P_{security}$ | Send | $collect_{HTML}(PII)\|\|retrieve_{DB}(PII) \rightarrow secure_{node} \rightarrow remote_{request}(PII)$ |

**[Article 15] Data Access ($P_{access}$).** Article 15 mandates user access to stored personal data. If the plugin utilizes any custom database for storing personal data, the plugin is required to provide data export functionality to comply with $P_{access}$. Note that a plugin is not strictly required to provide data access functions, as WordPress natively provides an exporter tool that can access data from WordPress's core databases. Yet, WordPress still encourages plugins to implement a data access function as a best practice [221]. In summary, we devise the following rules integrated into ChkPlug to determine whether a plugin follows $P_{access}$:

- A plugin stores PII via a custom database and the plugin provide the option to download all the stored PII $\rightarrow$ `COMPLY`
- A plugin stores PII in WordPress core database $\rightarrow$ `COMPLY`

- A plugin does not store PII → `COMPLY`
- A plugin stores PII in a custom database but provides partial/no set of PII to export → `VIOLATION`

While data can be exported in many ways, ChkPlug considers the sink nodes based on WordPress's official privacy guidelines [221].

WordPress provides website owners with a privacy tool to manage data export and gives plugins an interface to supply the exported data. ChkPlug specifically checks if each type of personal data stored is supplied to the interface. To achieve this, for each type of storage method, ChkPlug searches for a counterpart method that can retrieve the data from the storage method as listed in Table 4.5. For a WordPress storage function, the counterpart data retrieval function has a matching data type and data key. For example, if a plugin stores email through update_user_meta($id, 'email', $email), the corresponding retrieval function would be get_user_meta($id,'email'), as it operates on the matching data type user_meta, and has the same data key as 'email'. For a database operation, the counterpart deletion function has a matching table name. For example, if a plugin stores personal data in table tab1 through an *INSERT INTO* statement, the corresponding retrieval function would be a *SELECT* statement operating on the same table with the same personal data. In Table 4.5, we can observe that $P_{access}$ involves two types of actions, store, and export. Store actions indicate the process of collecting PII either from the user interface or database. The action inserts that particular PII into the database. However, export actions refer to the presence of retrieval functionalities. In this process, ChkPlug seeks the database retrieval for that PII.

More specifically, if a plugin complies with $P_{access}$, for each instance of data storage nodes, ChkPlug searches for instances of counterpart retrieval node that has a data flow path to the exported data supplied to WordPress. This indicates that the personal data is retrieved and supplied to WordPress's exporter tool. If there is at least one storage node that has no counterpart retrieval node supplied to WordPress, ChkPlug marks the plugin as violating the $P_{access}$.

**[Article 17] Data Deletion ($P_{delete}$).** $P_{delete}$ is almost similar to $P_{access}$, except for $P_{delete}$ plugin needs to provide deletion functionality even for the storage in WordPress core database according to article 17. We list the following rules to identify whether a plugin violates $P_{delete}$.

- A plugin stores PII and provides the option to delete all the stored PII → `COMPLY`
- A plugin does not store any PII → `COMPLY`

- A plugin stores PII but only provides partial or no set of PII to delete → `VIOLATION`

Similar to the analysis for $P_{access}$, for each type of storage method, we define a counterpart method that can delete the data from the storage method. Table 4.5 illustrates the required two flows, one for storage and another for delete. These are necessary for a plugin to comply with $P_{delete}$. For a WordPress storage function, the counterpart deletion function has a matching data type and data key. For example, if a plugin stores email data through update_user_meta($id,'email',$email), the corresponding deletion function would be delete_user_meta($id,'email'), as it operates on the matching data type user_meta, and has the same data key email. For a database operation, the counterpart deletion function has a matching table name. For example, if a plugin stores personal data into table $tab1$ through an *INSERT INTO* statement, the corresponding deletion function would be a *DELETE* statement operating on table $tab1$.

To be specific, if a plugin complies with $P_{delete}$, for each instance of a data storage node, we search for instances of counterpart deletion node. If there is at least one storage node that has no counterpart deletion node, we determine the plugin as violating $P_{delete}$.

**[Article 28] Third-party Data Sharing ($P_{share}$).** The plugin is responsible for disclosing third-party data sharing to users. If ChkPlug finds any remote request sink that personal data sources can traverse to, it implies that the plugin sends certain personal data to a third party. In such a case, the plugin is required to comply with $P_{share}$ according to article 28. If plugins do not send PII to a third party, then they do not need to follow $P_{share}$, and thus are automatically compliant. Whenever the plugin is sharing data with a third party it needs to disclose it in its privacy policy. Failing to do so will result in a GDPR violation. Moreover, $P_{share}$ applies regardless of the URL, because even if the receiving endpoint is the plugin developer, such a case is still considered third-party data sharing, as the plugin developer is considered a third-party from the perspective of the website owner that deploys the plugin in their website. In particular, we build the following rules to check violation of $P_{share}$ using ChkPlug:

- A plugin does not collect any PII → `COMPLY`
- A plugin does not share any PII → `COMPLY`
- A plugin shares PII with a third party (including the plugin developer website) and discloses it in the privacy policy → `COMPLY`
- A plugin shares PII with a third party (including the plugin developer website) but does not disclose it in the privacy policy → `VIOLATION`

- A plugin shares PII and does not have any privacy policy $\rightarrow$ `VIOLATION`

As a result, we search for all the remote request sink points that send out PII and aggregate all personal data types sent out via these sinks. Note that we mark all remote requests as sending data to third parties, because a plugin developer, itself, is considered a third party to the website that deploys the plugin. That is why sending data to the plugin developer is considered third-party data sharing. We search for the flow of send and disclosure (Table 4.5) in the plugin CCPG. Unique to analyzing $P_{share}$, we consider privacy policy to complete the flow for disclosure action. We extract the privacy text the plugin provides to WordPress through *wp_add_privacy_policy_content()* (WordPress's recommended way to provide privacy policy texts [220]). We then leverage existing work on privacy policy, PolicyLint [6] to analyze the privacy text and identify the list of personal data collected by third parties. Next, we compare this list with ChkPlug's output. If the plugin fails to disclose any data collected by a third party (or does not have a privacy policy at all), we report that plugin as in violation of $P_{share}$.

**[Article 32] Security of PII ($P_{security}$).**

According to Article 32, plugins need to encrypt or perform hash operations on personal data before sending it over the network. Even if they use a secure channel for such communication, then it is considered protected. Failure to do so will violate $P_{security}$. We determine a plugin violating $P_{security}$ by checking the following rules-

- A plugin sends the encrypted PII to secure/insecure channel $\rightarrow$ `COMPLY`
- A plugin sends PII to any remote URL a secure communication channel (*e.g.,* HTTPS) $\rightarrow$ `COMPLY`
- A plugin sends PII to any remote URL an insecure communication channel (*e.g.,* HTTP) $\rightarrow$ `VIOLATION`

Both $P_{share}$ and $P_{security}$ requirements are related to whether the plugin sends personal data to a remote URL. While many security settings are out of plugin developers' control, we search for direct evidence that the plugin is transmitting personal data via an insecure method.

Specifically, for each remote request sink point that personal data can traverse to, we parse the possible endpoint URLs which are previously analyzed by our detector (Section 4.4.3) and check whether the URLs use HTTP rather than HTTPS protocol. Furthermore, for each such sink point, we check if the node has a *SECURE* label, which would imply that the data is hashed or encrypted. If a plugin uses an insecure method to transmit personal data and does not secure the data in any way, it violates $P_{security}$.

**Report Generator**

After all the analyses are done, ChkPlug compiles a human-readable report that includes the final compliance decision and, if applicable, all evidence that a plugin violates GDPR. Moreover, to help developers comply with GDPR, ChkPlug produces a fix report that guides plugin developers step by step to fix their violations. We have uploaded an example report to [162]. In the following, we illustrate how ChkPlug generates compliance reports.

- Violation evidence. ChkPlug outputs all evidence that shows the plugin is required to comply with a GDPR law, whereas the plugin lacks the actions to comply. Take the example shown below, for a piece of evidence that shows the plugin is not compliant with $P_{delete}$, ChkPlug outputs the data storage sink of personal data that indicates the plugin needs to comply with $P_{delete}$, whereas the plugin has no corresponding deletion method. ChkPlug indicates the code location where the developer can find this sink, as well as the personal data type that is being stored. We have illustrated a case in which ChkPlug detects PII storage in a specific file of the plugin's source code. As a result, ChkPlug is indicating the applicable GDPR policies for such storage.

  [Art.17, Right to erasure] WordPress storage of PII through update_user_meta ($current_user → ID, 'last_name', $asmember_register_name) does not have a corresponding deletion method. Storage method found in file `templates/single-asmember-memberships.php` at line 771. Storage method stores user data of type: last name.

- Fix report. To help plugin developers comply with GDPR, ChkPlug refers to the plugin handbook provided by WordPress [220] and provides a breakdown of all specific steps needed to comply with GDPR for all laws the plugin violates, along with auto-generated template code and *TODOs* inside the code based on the specific law requirements of the plugin.

With that, ChkPlug determines whether a plugin complies with GDPR by analyzing the cross-language code property graph.

## 4.5 Implementation and Dataset

In this section, we describe our open-source implementation and dataset.

### 4.5.1 Implementation

Our implementation is open-source with 24,251 lines of code (LoC) *excluding* any third-party libraries or open-source tools. The implementation is available at: `https://github.com/faysalhossain2007/`

CHKPLUG. We adopt an open-source HTML parser [80] and make changes with 1,190 LoC in Python. We also adopt an open-source SQL parser [191] and make changes with 436 LoC written in Python. Our implementation of JavaScript and PHP CPG is based on navex [4] and we make the following modifications.

- A customized JavaScript parser. We use Esprima to build a customized parser that converts JavaScript code to be accepted by Navex.

- Intra-procedural data-flow edges. For a given REACHES edge (*i.e.,* a traditional data flow edge that connects data flows between two lines of code) between two lines of code, ChkPlug searches for the corresponding nodes that represent the same data unit in the AST of the two lines of code and connect them.

- Hierarchical edges. ChkPlug traverses the hierarchy of ASTs for each line of code and connects data units inside the AST, capturing and connecting movements of data units within a line of code.

- Function call and return edges. First, ChkPlug finds call edges between different nodes. They are constructed between a function call and the function definition, and it is difficult to track the individual arguments passed to the function. As a result, ChkPlug follows call edges and connects dataflows between arguments within a function call and the parameters inside the function definition. Second,

  following the logic for function call edges, ChkPlug tracks how data is processed after it is passed to a function and then returned. Therefore, ChkPlug follows call edges to locate all the return statements inside the function definition and connects dataflows for the returned data.

- Traversal of PII and Security Nodes. After constructing all dataflow edges and identifying all the PII sources and all security nodes that hash or encrypt data, ChkPlug performs dataflow traversal using 'apoc.path.subgraphNodes', a Neo4J procedure that traverses all dataflow edge types we designate and outputs all reachable nodes within the graph. ChkPlug uses the procedure to traverse from all the source nodes and get all nodes reachable through the dataflow edges that it generated. Therefore, ChkPlug obtains all nodes that can possibly be reached by any of the personal data source or security nodes. Our tool labels all nodes that can be reached by a personal data source as *PERSONAL*, and all nodes that can be reached by a security node as *SECURE*, and ChkPlug records down the corresponding sources that can reach to the node. Furthermore, ChkPlug combines the personal data type of the personal data sources (*e.g.,* an array node may be reachable by a node with email data and another node with password data, and we mark the array node as containing email and password data). Similarly, for nodes reachable by security nodes, our tool combines the encryption/hashing methods

Table 4.6: Ground truth labeled data of 200 plugins for evaluation. ✓= violation ✗ = non-violation.

| **#Violation** | $\#P_{access}$ | $\#P_{delete}$ | $\#P_{share}$ | $\#P_{security}$ | **#Plugin** |
|---|---|---|---|---|---|
| ✗ | 200 | 131 | 190 | 190 | 124 |
| ✓ | 0 | 69 | 10 | 10 | 76 |

from the sources. Next, ChkPlug identifies all sinks among the nodes that are reachable by personal data sources, as these sinks conduct operations on the personal data sources; among these sinks, it also identifies the ones that are reachable by security nodes, as these sinks process encrypted or hashed data.

Next, we also describe our manual efforts in identifying these source/sink/security functions for our analysis. Specifically, we inspect the behavior of APIs in different languages (*e.g.,* DOM, PHP, and WordPress) manually. Let us use the WordPress APIs as an example. The WordPress team maintains a number of packages required for the proper operations of WordPress. These packages live in the `wp-include` directory of WordPress and can be used by plugin developers. Similarly, `wp-admin` contains functions related to administrative actions. To begin, we gather all of the functions defined in the `wp-admin` and `wp-include` directories of WordPress using our Python crawler. In total, there are 2,885 functions [218]. We then categorize these functions by behavior as either sources, sinks, or neither. Again, each function sets, deletes, or retrieves different user information. So, we label both operations and the types of collected user information manually. Before labeling, we read the complete descriptions of the WordPress functions from the official website. In total, we find 30 store, 65 retrieval, and 28 delete functions. ChkPlug uses these labeled functions while building the detector.

## 4.5.2 Dataset Collection

We collect WordPress plugins from the official website [219] in May 2022. To that end, we build a Selenium-based Python crawler that collects all the links of published plugins (so far) from `https://plugins.svn.wordpress.org/`. We visit each plugin's webpage (hosted in [219]). If a particular plugin is still active, then the user will be able to download it. Using our crawler, we are able to get the source code of all the plugins successfully. For each plugin, we collect the active installation numbers, last updated information, average ratings, current version, and plugin's code. We select 2,722 plugins from the marketplace. Out of these 2,722 plugins, we manually analyzed 200 plugins as ground truth for evaluating the performance of our tool.

### Manual Labeling

Important to our evaluation is a ground truth with which we can compare the results of our system analysis. To establish this ground truth we manually analyze the plugin behavior and source code. Three computer science students read the GDPR extensively and consulted with a domain expert—who has six years (2016-) of experience in working with GDPR and privacy regulations—to learn the labeling methods. Later, they manually label those plugins, and whenever they have a conflict, they resolve it by discussing it with the domain expert. Each manual analysis aims to establish compliance or non-compliance concerning the same properties that ChkPlug does. Those properties are data access, data deletion, data sharing, and security of PII. To begin, an analyst will install the plugin into a local version of WordPress. From there, they interact with the site and plugin operating as a regular user would. The analysts are alert to all of the instances where the plugin would prompt for information. For any case where PII is collected, they then seek out the appropriate access and deletion endpoints. If no such interfaces are found, the plugin is said to be non-compliant. If found, a further analysis still must be done to validate the implementation. This requires inspecting the database backing the site. More particularly, the analyst will locate the records in the database added by the plugin. Database management tools with user-friendly front ends (*e.g.,* Adminers) make this process simple. An analyst can then confirm the plugin's compliance by ensuring the database is updated appropriately when interfaces like those to remove PII are invoked.

Inspecting the local database will not help identify cases of third-party information sharing. For these cases, the approach is to inspect the source code. One area to pay close attention to is expressions. Third-party packages need to be included in the source file so their presence is a possible indication that user information may be shared externally. To confirm, it is helpful to load the plugin source into an *IDE* and navigate the call graph. Most IDEs have features to find usages of particular methods which can help trace data flow. Our analysis includes inspecting the source code in this way to identify violations. Lastly, we complement this static analysis with a more dynamic approach like using a debugger. If there are functions whose behavior cannot be determined or ambiguity in the code, we launch Xdebug, a popular PHP debugger. The debugger is connected to the WordPress instance and breakpoints are set in the areas of confusion. We could then interact with the plugin through the web interface, trigger the breakpoints, and gain access to data on the stack. This approach is helpful to confirm the exact information being transferred by the API calls. Ultimately, these methods are used to establish compliance concerning information sharing and information security.

While thorough, this process is very time-consuming. Often an analysis of a single plugin could take between 4-6 hours (approx.). In total, we manually inspect 200 plugins using this approach. Of these 200, 76 plugins violate one or many GDPR violations while 124 plugins comply with GDPR policies. Table 4.6 provides a breakdown of compliance across different GDPR policies. Note that we have not considered WordPress themes because those are for polishing the User Interface and website visual effects, which do not collect user data, in general, [84, 215]. We also manually verified 30 randomly-selected WordPress themes and found that none collect PII.

## 4.6 Evaluation

In the following section, we run a series of experiments to evaluate the performance of ChkPlug. In the end, we evaluate the end-to-end performance of ChkPlug (Section 4.6.4). We run the experiments on AWS with 6 EC2 instances. These machines are of the t3.medium variety. They feature Intel(R) Xeon(R) Platinum 8259CL Processors, 2 vCPUs, 4 GB of RAM, and run Amazon Linux 2.

### 4.6.1 Evaluation Questions and Metrics

To evaluate the performance of ChkPlug, we seek answers to the following questions:

- **RQ1.** What is the performance of ChkPlug in detecting GDPR violations in plugins?
- **RQ2.** How many plugins violate GDPR as reported by ChkPlug?
- **RQ3.** What is the computation overhead of ChkPlug?

The answer to the first question helps to show the effectiveness of ChkPlug, and the answer to the second question helps to understand the current compliance situation of plugins. Answering the third one will ensure the scalability of the tool.

### 4.6.2 RQ1: Performance Evaluation of ChkPlug

To evaluate the ability of ChkPlug to identify plugin violations, we compare the results of ChkPlug with manual analysis results. As mentioned in Section 4.5.2, we manually labeled 200 plugins as ground truth. In the following, we report the accuracy of ChkPlug by evaluating these 200 plugins.

We report the true positive (TP), false positive (FP), true negative (TN), and false negative (FN) rates of the analysis in Table 4.7. We consider violation as positive data and non-violation as negative data. TP indicates plugins labeled by both ChkPlug and the human annotator to be in violation, FP is labeled by ChkPlug to be in violation whereas manual analysis determines them to be compliant (non-violation), TN denotes plugins labeled by both ChkPlug and manual analysis to be compliant, and FN are labeled by ChkPlug to be compliant but human labels them as in violation.

We observe that our tool achieves an average of 98.8% TNR and 89.3% TPR in detecting four different GDPR violations. In the following, we discuss in detail about performance in each of those categories-

Table 4.7: RQ1: Detailed performance of ChkPlug on 200 plugins for identifying different GDPR violations. Here, TP= True Positive, FP = False Positive, TN = True Negative, and FN = False Negative.

| **Policy** | **TP** | **TN** | **FP** | **FN** | **TPR** | **TNR** |
|---|---|---|---|---|---|---|
| $P_{access}$ | 0 | 200 | 0 | 0 | 100% | 100% |
| $P_{delete}$ | 66 | 127 | 5 | 2 | 97% | 96.2% |
| $P_{share}$ | 7 | 189 | 1 | 3 | 70% | 99.5% |
| $P_{security}$ | 9 | 189 | 1 | 1 | 90% | 99.5% |
| **Total** | **82** | **705** | **7** | **6** | **89.3%** | **98.8%** |

### Evaluation of $P_{access}$

ChkPlug identified no plugins as violating an access policy. It is consistent with our manual analysis. ChkPlug achieves TNR of 100% and TPR of 100%. The main reason for all the plugins not violating access regulations is- whenever they store PII, they use WordPress core database. As a result, it is not strictly required to provide data export functionality to the user as WordPress will by default handle that.

### Evaluation of $P_{delete}$

Out of the 200 plugins with ground truth from manual analysis, ChkPlug marks 66 plugins as violating a deletion policy with a TPR of 97% (127 out of 131) and a TNR of 96.2% (66 out of 69). We notice that a common source of false positives on the delete policy has to do with identifying deletion endpoints. While we can identify the static deletion endpoints made available to plugins with a process similar to the data deletion functionalities that we found manually, a developer is free to write their own implementation to adhere to the GDPR requirements. For these endpoints, ChkPlug must fall back to pattern matching. In the false positive cases, the pattern matching is not exhaustive enough. For instance, in the plugin 'bp-featured-members', we observe PII being collected with a call to an add_user function. ChkPlug correctly flagged this as a case where a deletion endpoint must be provided. However, the developer implemented deletion endpoint was defined as remove_user. This definition was easy to spot during a manual analysis, however, this form was not part of the patterns the tool searched for.

### Evaluation of $P_{share}$

ChkPlug marked 7 plugins as violating a sharing policy with a TNR of 99.5% (189 out of 190) and a TPR of 70% (7 out of 10). Our tool predicted one false positive. The 'wp-user-avatar' plugin, previously was sending user information to 'https://www.gravatar.com/avatar/' inside the 'class-wp-user-avatar-functions.php' file. However, while manually investigating this plugin, a human annotator noticed that the plugin had saved the deprecated code files inside a folder that is no longer used. Our tool is not able to detect which files are active and which are not. It analyzes all source files inside the plugin folder.

```
$request = wp_remote_get(
    'https://www.google.com/recaptcha/api/siteverify?secret=' .$secret_key . '&response=' . $response .
 ↪  '&remoteip=' . $remote_ip
);
```

Listing 13: Security violations identified by ChkPlug.

**Evaluation of $P_{security}$**

ChkPlug marked 9 plugins as violating a security policy with a TNR of 99.5% (189 out of 190) and a TPR of 90% (9 out of 10). ChkPlug analysis of security violations is conservative. When ChkPlug cannot determine the endpoint receiving PII is using HTTPS, the corresponding plugin will be flagged as a violation. We see a case where this occurs in the 'google-site-kit' plugin. It tries to send PII using the URL: $\$url = \$this- > url(\$uri)$;. In the case of this plugin, the system was unable to resolve some of the components of the URL argument due to missing patterns. Even though manual analysis can identify the endpoint to be secure, since the full URL could not be resolved, it was marked as a violation. This is a case where ChkPlug was over-conservative and led to a false positive.

### 4.6.3 RQ2: Measurement Analysis

We run ChkPlug on 2,722 plugins from the WordPress store. Note that, these 2,722 plugins include the ones listed in Table 4.6. We find that 14% (381 out of 2,722) plugins do not comply with GDPR.

Table 4.9 breaks down violations across particular policies. We observe that the most common form of violation was a delete violation, where users are not empowered to delete their data. Around 13.52% (368 out of 2,722) plugins do not provide data deletion functionalities. We observe the violations in other categories as 0.7% in $P_{share}$, 1.3% in $P_{security}$, and 0% in $P_{access}$. For the 19 plugins that violate $P_{share}$, we found that 16 of them do not have a privacy policy but share PIIs; the remaining three have a privacy policy but fail to mention PII sharing in that policy. Most of the plugins use the WordPress core database for storing data. Since WordPress provides the export feature for such storage, we experience fewer violations in $P_{access}$.

We also analyze the PIIs in GDPR violations. Examples of such PIIs are- phone number, date of birth, and physical address. We also break down PIIs in the GDPR violations based on data categories, and the relations to the WP database (called WP and non-WP). We show the details in Table 4.8.

We provide a few case studies of violations in Section 4.7.

### 4.6.4 RQ3: Computational Overhead

We benchmarked the runtime performance ChkPlug on 2,722 plugins. The benchmarks ran on the same t3.medium machines described at the start of section 4.6. No unnecessary user programs were running during

Table 4.8: Breakdown PIIs into violations based on the relations to the WP database. WP=wordpress core database, non-WP=custom database.

| **PII** | $\#P_{delete}$ | | $\#P_{share}$ | | $\#P_{security}$ | |
|---|---|---|---|---|---|---|
| | **WP** | **non-WP** | **WP** | **non-WP** | **WP** | **non-WP** |
| Username | 310 | 11 | 15 | 1 | 11 | 3 |
| Email | 29 | 3 | 2 | 0 | 5 | 4 |
| Password | 18 | 1 | 0 | 0 | 0 | 0 |
| Address | 12 | 1 | 2 | 0 | 0 | 0 |
| First Name | 12 | 0 | 2 | 0 | 3 | 3 |
| Last Name | 12 | 1 | 2 | 0 | 3 | 3 |
| IP | 7 | 2 | 1 | 0 | 5 | 4 |
| State | 8 | 0 | 2 | 0 | 1 | 0 |
| Country | 7 | 0 | 1 | 0 | 1 | 0 |
| Phone | 6 | 0 | 0 | 0 | 0 | 1 |
| Postcode | 4 | 0 | 1 | 0 | 1 | 0 |
| City | 4 | 0 | 1 | 0 | 1 | 0 |
| Birthday | 1 | 0 | 0 | 0 | 0 | 0 |

```
public function init() {
        if ( !class_exists( '\\GF_User_Registration' )  !function_exists( '\\wlmapi_update_member' ) ) {
                // Gravity Forms User Registration Add-On and/or WishList Member not activated, so we do
  ↪  nothing
                return;
        }
      ....
}
....
private function insert_wlm_data( $user_id, $wlm_data ) {
        if ( empty( $wlm_data ) ) {
                return;
        }
        \wlmapi_update_member( $user_id, $wlm_data );
}
```

Listing 14: FP analysis of sending data to third-party.

Table 4.9: RQ2: Measurement analysis results of 2,722 plugins using ChkPlug.

| **Policy** | **#Violation** | **Percentage** |
|---|---|---|
| $P_{access}$ | 0 | 0% |
| $P_{delete}$ | 368 | 13.52% |
| $P_{share}$ | 19 | 0.7% |
| $P_{security}$ | 36 | 1.3% |

the benchmark. We observe that the system can process an entire plugin in just a few minutes. The average time taken for analyzing each of the plugins is approximately 9.1 minutes. We run the benchmarks with a three-hour timeout but no plugin has taken that long to be processed. The longest processing time is around two hours. In addition, we show in Figure 4.4 the cumulative distribution of plugins that take longer than the average time to analyze, i.e., the CDF of around 14% of plugins. We observe that more than 98% of plugins finish analysis within an hour.

We notice that the time it takes to process a plugin is proportional to its size. The larger the plugin is, the more processing time is needed. While we optimize the communication with the backing Neo4J database, batching reads and writes where possible, in profiling ChkPlug, we discover that a large portion of the

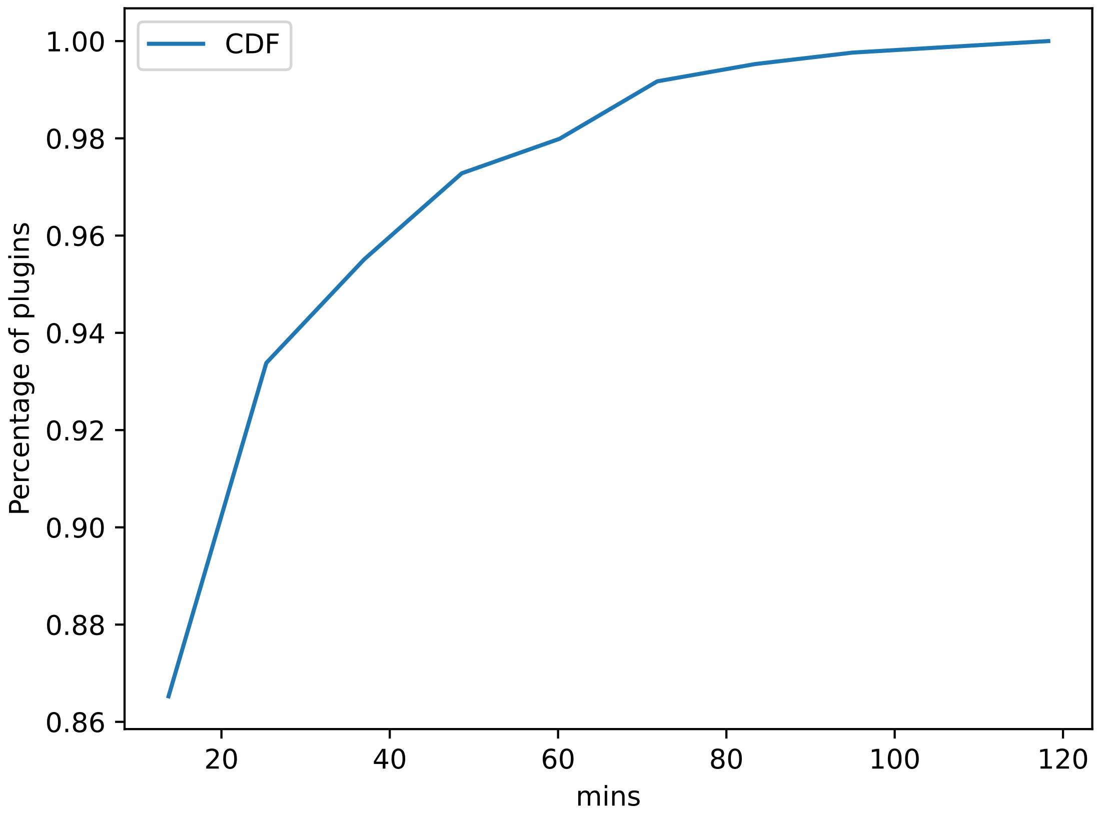


Figure 4.4: RQ3: Cumulative distribution function (CDF) of Computation overhead of CHKPLUG.

runtime is consumed by network overhead from these communications. This is a consequence of using an external database.

## 4.7 Case Study

In this section, we illustrate several case studies for the 381 plugins found by CHKPLUG, which violate GDPR policies. Specifically, we describe one violation example in each category. Note that we have responsibly reported all the violations to plugin developers; so far, we have not received any responses yet.

**Violation of $P_{delete}$** 'Namaste! LMS' (`https://wordpress.org/plugins/namaste-lms/`) provides a learning management system with an average rating of 4.5. Currently, it has 800+ active installations. It collects and stores user information in the database in file 'controllers/woocommerce.php' at line 62. It stores all PII in the WordPress core database. However, CHKPLUG does not find any evidence of providing users the option to delete those PII. It is in clear violation of $P_{delete}$.

**Violation of $P_{share}$** 'Vindi WooCommerce 2' (`https://wordpress.org/plugins/vindi-payment-gateway/`) provides a one-time payment solutions using Woocommerce subscriptions. It has been regularly maintained by Vindi (the developer of this plugin) with an active installation of 200+ users. At the time of writing this paper, it has an average rating of 3.7. Our tool detects this plugin as sharing data with a third party. In the 'controllers/CustomerController.php' file, it creates customer profile data. It also has the functionality to update customer data. Later, it sends that private data to the third party via API calls to `https://sandbox-app.vindi.com.br/api/v1/` in file *src/services/Api.php* at line 242. Unfortunately, it does not even have a privacy policy. As a result, it violates GDPR by not disclosing PII sharing in the privacy policy.

**Violation of $P_{security}$** 'Gallery Custom Links' (`https://wordpress.org/plugins/gallery-custom-links/`) allows linking images from galleries to a specified URL. It has 50,000+ active installations and an average rating of 4.5 stars. Our tool detects this plugin as violating $P_{security}$ because it is sending PII over an HTTP channel (to `http://meowapps.com`) instead of a secure channel. We find such problematic behavior in 'common/premium/updater.php' at line 418.

## 4.8 Discussion & Limitations

In this section, we summarize a few important discussions related to our data selection and tool evaluation, the limitations of our tool, adaptability to other CMS platforms, and future research directions.

**WordPress Plugin Selection in the Evaluation** While selecting the plugins for measurement analysis, we include the most popular ones as violations in these plugins would have the most impact on users. We use AWS for running all the evaluations and measurement analysis. It is expensive. Only around 1,000 plugins cost us $229.86 and it takes almost two and half days to finish running the analysis.

**Deployment Model** Website owners, plugin developers, and law professionals can use ChkPlug to detect GDPR compliance. ChkPlug will help website owners to identify plugins that violate GDPR laws. This will allow them to make informed decisions before integrating a particular plugin into their website. We have two mitigation suggestions for the plugin developers. First, developers can follow WordPress guidelines, which may help plugins to comply with many GDPR articles, e.g., data access, automatically. Second, our analysis report helps developers to make their plugins comply with GDPR. A sample report is shown in [162] Finally, law professionals can use our tool to help identify websites that violate GDPR laws. Note that, currently the WordPress team does not hold any responsibility for the GDPR compliance of the plugins. So, the website owner must take extra steps to ensure GDPR compliance with their website before integrating any WordPress plugins.

**False Negatives** Like all static analysis tools, ChkPlug may suffer from a lack of pattern coverage, leading to false negatives. However, from Section 4.6.4, we can observe that our tool's false negative cases are rare. It shows that we are not missing many plugins which violate GDPR regulations.

**False Positives** Due to the inherent nature of the static analysis, ChkPlug has false positives, e.g., due to imprecise modeling of call edges. However, according to our evaluation results, ChkPlug's TNR is very high.

**CMS Platforms Other than WordPress** We mainly perform our analysis on WordPress plugins as it is the most popular CMS powering over 32% of the World Wide Web [166]. However, our tool is extendable to plugins from other platforms (*e.g.,* joomla [91], drupal [52]) as long as those are written using PHP, HTML, JavaScript. ChkPlug will not be able to detect GDPR compliance in CMS platforms (*e.g.,* DotNetNuke [51], Kentico [97]) that use other programming languages (such as, C# and ASP.net).

**User Consents** In our analysis, we have not considered user consent. As a result, we may miss plugins that collect and store PII without user permission. User consent involves users' interaction (via UI elements) and reading the terms & conditions document. It is context-dependent and there are many ways to collect user consent. It is another interesting research topic to analyze user consent in different contexts. We leave it for future work.

## 4.9 Summary

WordPress plugins provide additional functionalities to websites built with WordPress but also face a new problem in the era of privacy, i.e., its compliance with privacy laws, particularly GDPR. To the best of our knowledge, no prior works have provided automated checks of various GDPR articles on WordPress plugins partially due to their cross-language nature (the involvement of HTML, JavaScript, PHP, and SQL).

In this paper, we design a tool, called ChkPlug, to automatically check whether WordPress plugins comply with GDPR. Ground truth evaluation shows that ChkPlug performs well, achieving an average of 98.8% TNR and 89.3% TPR in checking GDPR compliance. We believe that website owners as well as developers will get benefit by using ChkPlug.

We hope that ChkPlug can shed light on the need for future research on checking CMS plugins' compliance against GDPR articles. More importantly, we believe that as the first step towards a more private, law-compliant community, ChkPlug will help future researchers as well as developers to better understand GDPR and fill the gap between law enforcement and software development.

# Chapter 5

# Conclusion

The rapid advancement of emerging technology significantly enhances our everyday lives. Unfortunately, it also introduces many new applications that contain security and privacy threats. For example, applications that do not provide adequate reasons for accessing user-sensitive data can breach privacy and present a security risk. Similarly, any vulnerable third-party software can unwittingly introduce new security vulnerabilities into the application using that software.

This dissertation outlines and tackles two key challenges associated with identifying untrustworthy applications from a security and privacy perspective. They are– (1) the lack of labeled data, and (2) the cross-language analysis. These challenges are examined and tackled within the realms of emerging technologies in this dissertation.

Regarding the first challenge – the limited availability of labeled data – this dissertation first dives into the data-driven ML-based approach for detecting untrusted applications asking for extraneous access to user-sensitive data across multiple platforms (Chapter 2). Using this tool, we discover 329 such untrusted applications from web and IoT platforms. Later, we examine that relying only on ML-based approach is not sufficient. To address that issue, this dissertation then presents an ML-enhanced data-driven approach for identifying vulnerable code in large-scale software (Chapter 3). On the other hand, in response to the second challenge – cross-language analysis – this research introduces a cross-language code property graph that allows for tracking the information flow across multiple platforms (Chapter 4). This cross-language code property graph aids in checking GDPR compliance in web applications and identifies 381 web applications that do not follow one or more GDPR regulations.

The ideas and concepts in this dissertation have culminated in a number of tangible and beneficial resources that have been shared with the broader community: TKPerm, a tool to identify applications asking for unnecessary access to user-sensitive data; VulDetector, a tool to detect vulnerable code in large-scale real-world software; ChkPlug, a tool to check GDPR compliance in web applications. Furthermore, this dissertation's research has gained recognition in the industrial sector. Google has validated 59 zero-day vulnerabilities uncovered by VulDetector, assigning 12 CVEs. Together, these endeavors aid in reducing the risks linked with untrustworthy applications, laying the groundwork for confronting two primary challenges in building detection tools in computer security and privacy amidst rapidly evolving technology.

There are two immediate open questions that require further investigation and exploration in the future. The first question pertains to the robustness of these detection tools and the potential impact of adversaries. It remains to be examined how the presence of adversaries may affect the performance of these tools. Identifying such adversaries is crucial for making informed decisions. Thus, future research should focus on investigating the influence of adversaries and determining reliable methods to detect their presence, ultimately leading to more conclusive decisions. The second question revolves around delivering the outcomes of the introduced detection tool to various stakeholders, including– general users, policymakers, and developers. To promote wider adoption, it is important to improve the usability and user-friendliness of cross-platform detection tools. We leave it for future work to concentrate on designing and refining the user interfaces, incorporating intuitive visualizations to represent identified threats, and providing user-friendly documentation and tutorials. This will facilitate seamless deployment and utilization of these tools. Furthermore, investigation on incorporating user feedback will also enhance the performance of these tools.

# Bibliography

[1] *2GDPR*. `https://2gdpr.com/`. 2022.

[2] Yasemin Acar et al. "Sok: Lessons learned from android security research for appified software platforms". In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 433–451.

[3] Muhammad Jamal Afridi, Arun Ross, and Erik M Shapiro. "On automated source selection for transfer learning in convolutional neural networks". In: *Pattern recognition* 73 (2018), pp. 65–75.

[4] Abeer Alhuzali et al. "{NAVEX}: Precise and scalable exploit generation for dynamic web applications". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 377–392.

[5] Saed Alrabaee et al. "Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code". In: *Digital Investigation* 12 (2015), S61–S71.

[6] Benjamin Andow et al. "{PolicyLint}: Investigating Internal Privacy Policy Contradictions on Google Play". In: *28th USENIX security symposium (USENIX security 19)*. 2019, pp. 585–602.

[7] Benjamin Andow et al. "Actions Speak Louder than Words:{Entity-Sensitive} Privacy Policy and Data Flow Analysis with {PoliCheck}". In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 985–1002.

[8] *Android Security Bulletins*. `https://source.android.com/docs/security/bulletin`. 2023.

[9] Kathy Wain Yee Au et al. "Pscout: analyzing the android permission specification". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 217–228.

[10] Thanassis Avgerinos et al. "Automatic exploit generation". In: *Communications of the ACM* 57.2 (2014), pp. 74–84.

[11] Michael Backes, Boris Köpf, and Andrey Rybalchenko. "Automatic discovery and quantification of information leaks". In: *Proceedings of IEEE Symposium on Security and Privacy*. IEEE. 2009, pp. 141–153.

[12] Michael Backes et al. "Efficient and flexible discovery of php application vulnerabilities". In: *2017 IEEE european symposium on security and privacy (EuroS&P)*. IEEE. 2017, pp. 334–349.

[13] Privacy Badger. *Electronic Frontier Foundation*. 2019.

[14] Gustavo EAPA Batista, Ronaldo C Prati, and Maria Carolina Monard. "A study of the behavior of several methods for balancing machine learning training data". In: *ACM SIGKDD explorations newsletter* 6.1 (2004), pp. 20–29.

[15] Shai Ben-David et al. "A theory of learning from different domains". In: *Machine learning* 79.1-2 (2010), pp. 151–175.

[16] Guru Bhandari, Amara Naseer, and Leon Moonen. "CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software". en. In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21)*. ACM, 2021, p. 10. ISBN: 978-1-4503-8680-7. DOI: `10.1145/3475960.3475985`.

[17] Piotr Bojanowski et al. "Enriching word vectors with subword information". In: *Transactions of the Association for Computational Linguistics* 5 (2017), pp. 135–146.

[18] Dino Bollinger et al. "Automating cookie consent and GDPR violation detection". In: *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association. 2022.

[19] Samuel R Bowman et al. "A large annotated corpus for learning natural language inference". In: *arXiv preprint arXiv:1508.05326* (2015).

[20] Brian Jackson. *WordPress GDPR Compliance – Everything You Need to Know*. https://kinsta.com/blog/wordpress-gdpr-compliance/#who-does-gdpr-impact. Sept. 2022.

[21] Juan Caballero et al. "Measuring {Pay-per-Install}: The Commoditization of Malware Distribution". In: *20th USENIX Security Symposium (USENIX Security 11)*. 2011.

[22] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[23] *California Consumer Privacy Act*. https://oag.ca.gov/privacy/ccpa. Sept. 2021.

[24] *Calligo*. https://www.calligo.io/gdpr-services-eu-representatives/. 2022.

[25] Davide Canali, Davide Balzarotti, and Aurélien Francillon. "The role of web hosting providers in detecting compromised websites". In: *Proceedings of the 22nd international conference on World Wide Web*. 2013, pp. 177–188.

[26] Ionuţ Cernica, Nirvana Popescu, et al. "Security evaluation of wordpress backup plugins". In: *2019 22nd International Conference on Control Systems and Computer Science (CSCS)*. IEEE. 2019, pp. 312–316.

[27] Sang Kil Cha et al. "Unleashing mayhem on binary code". In: *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 380–394.

[28] Saikat Chakraborty et al. "Codit: Code editing with tree-based neural models". In: *IEEE Transactions on Software Engineering* (2020).

[29] Saikat Chakraborty et al. "Deep learning based vulnerability detection: Are we there yet". In: *IEEE Transactions on Software Engineering* (2021).

[30] *Checkmarx*. https://checkmarx.com/. 2021.

[31] Jia Chen, Yu Feng, and Isil Dillig. "Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 875–890.

[32] Vitaly Chipounov et al. "Selective symbolic execution". In: *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*. CONF. 2009.

[33] Chia Yuan Cho et al. "MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery." In: *Proceedings of USENIX Security Symposium*. Vol. 139. 2011.

[34] *CHRISTOPHER D. MANNING. Dropping common terms: stop words*. https://nlp.stanford.edu/IR-book/html/htmledition/dropping-common-terms-stop-words-1.html.

[35] *Chrome Extension Crawler*. https://robwu.nl. 2022.

[36] *Chrome Extension Webstore*. https://chrome.google.com/webstore/category/extensions?hl=en. 2023.

[37] *Codeinwp*. https://www.codeinwp.com/blog/gdpr-compliance/. 2022.

[38] *CodeSonar*. https://www.grammatech.com/products/codesonar. 2022.

[39] *Common Vulnerability Scoring System*. https://nvd.nist.gov/vuln-metrics/cvss. 2022.

[40] Alexis Conneau et al. "Supervised Learning of Universal Sentence Representations from Natural Language Inference Data". In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, Sept. 2017, pp. 670–680. URL: https://www.aclweb.org/anthology/D17-1070.

[41] *Cookiebot*. `https://www.cookiebot.com/`. 2022.

[42] *Coverity*. `https://scan.coverity.com/`. 2021.

[43] *CppCheck*. `http://cppcheck.sourceforge.net/`. 2022.

[44] Lorrie Faith Cranor. "P3P: Making privacy policies more useful". In: *IEEE Security & Privacy* 1.6 (2003), pp. 50–55.

[45] *CVEChecker*. `https://www.oschina.net/p/cvechecker`. 2022.

[46] Hoa Khanh Dam et al. "Lessons learned from using a deep tree-based model for software defect prediction in practice". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 46–57.

[47] Martin Degeling et al. "We value your privacy... now take some cookies: Measuring the GDPR's impact on web privacy". In: *arXiv preprint arXiv:1808.05096* (2018).

[48] *Devign github implementation*. `https://github.com/epicosy/devign/`. 2022.

[49] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[50] Jeff Donahue et al. "Decaf: A deep convolutional activation feature for generic visual recognition". In: *International conference on machine learning*. 2014, pp. 647–655.

[51] *DotNetNuke*. `https://www.dnnsoftware.com/`. 2022.

[52] *Drupal*. `https://www.drupal.org/`. 2022.

[53] Xiaoning Du et al. "Leopard: Identifying vulnerable code for vulnerability assessment through program metrics". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 60–71.

[54] Ruian Duan et al. "Towards measuring supply chain attacks on package managers for interpreted languages". In: *arXiv preprint arXiv:2002.01139* (2020).

[55] Editorial Staff. *How Many WordPress Plugins Should You Install? What's too many?* `https://www.wpbeginner.com/opinion/how-many-wordpress-plugins-should-you-\install-on-your-site/`. Jan. 2022.

[56] William Enck et al. "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones". In: *ACM Transactions on Computer Systems (TOCS)* 32.2 (2014), pp. 1–29.

[57] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. "Black Widow: Blackbox Data-driven Web Scanning". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1125–1142. DOI: `10.1109/SP40001.2021.00022`.

[58] Adrienne Porter Felt, Kate Greenwood, and David Wagner. "The effectiveness of application permissions". In: *Proceedings of the 2nd USENIX conference on Web application development*. USENIX Association. 2011, pp. 7–7.

[59] Adrienne Porter Felt et al. "Android permissions demystified". In: *Proceedings of the 18th ACM conference on Computer and communications security*. ACM. 2011, pp. 627–638.

[60] Adrienne Porter Felt et al. "Android permissions: User attention, comprehension, and behavior". In: *Proceedings of the eighth symposium on usable privacy and security*. 2012, pp. 1–14.

[61] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. "Security analysis of emerging smart home applications". In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 636–654.

[62] Earlence Fernandes et al. "Flowfence: Practical data protection for emerging iot application frameworks". In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, pp. 531–548.

[63] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. "The program dependence graph and its use in optimization". In: *ACM Transactions on Programming Languages and Systems* 9.3 (1987), pp. 319–349.

[64] Pietro Ferrara and Fausto Spoto. “Static Analysis for GDPR Compliance.” In: *ITASEC*. 2018.

[65] *FindBugs*. `http://findbugs.sourceforge.net/`. 2022.

[66] *FlawFinder*. `http://www.dwheeler.com/flawfinder`. 2021.

[67] Joseph L Fleiss, Jacob Cohen, and Brian S Everitt. “Large sample standard errors of kappa and weighted kappa.” In: *Psychological bulletin* 72.5 (1969), p. 323.

[68] Vijay Ganesh, Tim Leek, and Martin Rinard. “Taint-based directed whitebox fuzzing”. In: *International Conference on Software Engineering*. IEEE. 2009, pp. 474–484.

[69] *GDPR regulations*. `https://gdpr-info.eu/`. 2021.

[70] *GitHub Advisory Database*. `https://github.com/advisories`. 2020.

[71] *Github Advisory on Tensorflow Framework*. `https://github.com/tensorflow/tensorflow/security/advisories`. 2020.

[72] Patrice Godefroid, Michael Y Levin, and David Molnar. “SAGE: whitebox fuzzing for security testing”. In: *Communications of the ACM* 55.3 (2012), pp. 40–44.

[73] *Google Play Store*. `https://play.google.com/store`. 2023.

[74] Gustavo Grieco et al. “Toward large-scale vulnerability discovery using machine learning”. In: *Proceedings of ACM Conference on Data and Application Security and Privacy*. 2016, pp. 85–96.

[75] Danny S Guamán, Jose M Del Alamo, and Julio C Caiza. “GDPR Compliance Assessment for Cross-Border Personal Data Transfers in Android Apps”. In: *IEEE Access* 9 (2021), pp. 15961–15982.

[76] Seth Hallem et al. “A system and language for building system-specific, static analyses”. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language Design and Implementation*. 2002, pp. 69–82.

[77] Christian Holler, Kim Herzig, and Andreas Zeller. “Fuzzing with code fragments”. In: *Proceedings of USENIX Security Symposium*. 2012, pp. 445–458.

[78] Gerard Holzmann. “Static source code checking for user-defined properties”. In: *Proc. IDPT*. Vol. 2. 2002.

[79] Jeremy Howard and Sebastian Ruder. “Universal Language Model Fine-tuning for Text Classification”. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2018, pp. 328–339.

[80] *HTML Parser*. `https://www.npmjs.com/package/htmlparser2`. 2022.

[81] Xuan Huo, Ming Li, Zhi-Hua Zhou, et al. “Learning unified features from natural and programming languages for locating buggy source code.” In: *IJCAI*. Vol. 16. 2016, pp. 1606–1612.

[82] *If this then that*. https://ifttt.com. 2023.

[83] *Is WordPress Really A 10 Billion Dollar Economy?* `https://www.presstitan.com/is-wordpress-really-a-10-billion-dollar-\economy/`. 2022.

[84] Jason Cosper. *WordPress Themes: Overview and Tips on Finding the Perfect One*. `https://www.dreamhost.com/blog/how-to-find-wp-themes/`. Sept. 2022.

[85] Ciera Jaspan, I-Chin Chen, and Anoop Sharma. “Understanding the value of program analysis tools”. In: *ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 2007, pp. 963–970.

[86] Simon Holm Jensen, Anders Møller, and Peter Thiemann. “Type Analysis for JavaScript”. In: *Proc. 16th International Static Analysis Symposium (SAS)*. Vol. 5673. LNCS. Springer-Verlag, Aug. 2009.

[87] Qiwei Jia et al. “Who leaks my privacy: Towards automatic and association detection with GDPR compliance”. In: *International Conference on Wireless Algorithms, Systems, and Applications*. Springer. 2019, pp. 137–148.

[88] Yunhan Jack Jia et al. "ContexloT: Towards Providing Contextual Integrity to Appified IoT Platforms." In: *NDSS*. 2017.

[89] Lingxiao Jiang et al. "Deckard: Scalable and accurate tree-based detection of code clones". In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE. 2007, pp. 96–105.

[90] Andrew Johnson et al. "Exploring and enforcing security guarantees via program dependence graphs". In: *ACM SIGPLAN Notices* 50.6 (2015), pp. 291–302.

[91] *Joomla*. `https://extensions.joomla.org/`. 2022.

[92] 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)*. IEEE. 2006, 6–pp.

[93] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. "CCFinder: A multilinguistic token-based code clone detection system for large scale source code". In: *IEEE Transactions on Software Engineering* 28.7 (2002), pp. 654–670.

[94] Georgios Kampanos and Siamak F Shahandashti. "Accept all: The landscape of cookie banners in Greece and the UK". In: *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer. 2021, pp. 213–227.

[95] Zhuoliang Kang, Kristen Grauman, and Fei Sha. "Learning with Whom to Share in Multi-task Feature Learning." In: *ICML*. Vol. 2. 3. 2011, p. 4.

[96] Ranjita Pai Kasturi et al. "Mistrust Plugins You Must: A Large-Scale Study Of Malicious Plugins In WordPress Marketplaces". In: *31th USENIX security symposium (USENIX security 22)*. 2012.

[97] *Kentico*. `https://www.kentico.com/`. 2022.

[98] Hee Yeon Kim et al. "DAPP: automatic detection and analysis of prototype pollution vulnerability in Node. js modules". In: *International Journal of Information Security* (2021), pp. 1–23.

[99] Jinyung Kim et al. "ScanDal: Static analyzer for detecting privacy leaks in android applications". In: *MoST* 12.110 (2012), p. 1.

[100] Seulbae Kim et al. "Vuddy: A scalable approach for vulnerable code clone discovery". In: *Proceedings of IEEE Symposium on Security and Privacy*. IEEE. 2017, pp. 595–614.

[101] David A Kinloch and Malcolm Munro. "Understanding C Programs Using the Combined C Graph Representation." In: *ICSM*. 1994, pp. 172–180.

[102] Ryan Kiros et al. "Skip-thought vectors". In: *Advances in neural information processing systems*. 2015, pp. 3294–3302.

[103] Georgios Kontaxis and Monica Chew. "Tracking protection in firefox for privacy and performance". In: *arXiv preprint arXiv:1506.04104* (2015).

[104] Teemu Koskinen, Petri Ihantola, and Ville Karavirta. "Quality of WordPress plug-ins: an overview of security and user ratings". In: *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing*. IEEE. 2012, pp. 834–837.

[105] Solomon Kullback. *Information theory and statistics*. Courier Corporation, 1997.

[106] Hongzhe Li et al. "CLORIFI: software vulnerability discovery using code clone verification". In: *Concurrency and Computation: Practice and Experience* 28.6 (2016), pp. 1900–1917.

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

[108] Song Li et al. "Detecting Node.js Prototype Pollution Vulnerabilities via Object Lookup Analysis". In: *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021.

[109] Song Li et al. “Mining Node.js Vulnerabilities via Object Dependence Graph and Query”. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/li-song.

[110] Song Li et al. “Mining Node.js Vulnerabilities via Object Dependence Graph and Query”. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/li-song.

[111] Xin Li et al. “Automated vulnerability detection in source code using minimum intermediate representation learning”. In: *Applied Sciences* 10.5 (2020), p. 1692.

[112] Zhen Li et al. “Sysevr: A framework for using deep learning to detect software vulnerabilities”. In: *IEEE Transactions on Dependable and Secure Computing* (2021).

[113] Zhen Li et al. “Vuldeelocator: a deep learning-based fine-grained vulnerability detector”. In: *IEEE Transactions on Dependable and Secure Computing* (2021).

[114] Zhen Li et al. “Vuldeepecker: A deep learning-based system for vulnerability detection”. In: *Proceedings of Network and Distributed Systems Security Symposium*. 2018.

[115] Zhen Li et al. “Vulpecker: an automated vulnerability detection system based on code similarity analysis”. In: *Proceedings of ACM SIGSAC conference on Computer & Communications Security*. 2016, pp. 201–213.

[116] Guanjun Lin et al. “Cross-project transfer representation learning for vulnerable function discovery”. In: *IEEE Transactions on Industrial Informatics* 14.7 (2018), pp. 3289–3297.

[117] Guanjun Lin et al. “POSTER: Vulnerability discovery with function representation learning from unlabeled projects”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2539–2541.

[118] Thomas Linden et al. “The privacy policy landscape after the GDPR”. In: *Proceedings on Privacy Enhancing Technologies* 2020.1 (2020).

[119] Fei Liu, Nicole Lee Fella, and Kexin Liao. “Modeling language vagueness in privacy policies using deep neural networks”. In: *Proc. of AAAI’16*. 2016.

[120] Frederick Liu et al. “Towards automatic classification of privacy policy text”. In: *School of Computer Science Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-ISR-17-118R and CMULTI-17* 10 (2018).

[121] Zhenguang Liu et al. “Combining graph neural networks with expert knowledge for smart contract vulnerability detection”. In: *IEEE Transactions on Knowledge and Data Engineering* (2021).

[122] Zhenguang Liu et al. “Smart Contract Vulnerability Detection: From Pure Neural Network to Interpretable Graph Feature and Expert Pattern Fusion”. In: *arXiv preprint arXiv:2106.09282* (2021).

[123] V Benjamin Livshits and Monica S Lam. “Finding Security Vulnerabilities in Java Applications with Static Analysis.” In: *USENIX security symposium*. Vol. 14. 2005, pp. 18–18.

[124] Mingsheng Long et al. “Learning transferable features with deep adaptation networks”. In: *arXiv preprint arXiv:1502.02791* (2015).

[125] Sicheng Luo et al. “Boosting symbolic execution via constraint solving time prediction (experience paper)”. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2021, pp. 336–347.

[126] Kin-Keung Ma et al. “Directed symbolic execution”. In: *International Static Analysis Symposium*. Springer. 2011, pp. 95–111.

[127] Michael Martin, Benjamin Livshits, and Monica S Lam. “Finding application errors and security flaws using PQL: a program query language”. In: *Acm Sigplan Notices* 40.10 (2005), pp. 365–383.

[128] Luca Massarelli et al. "Safe: Self-attentive function embeddings for binary similarity". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2019, pp. 309–329.

[129] Célestin Matte, Nataliia Bielova, and Cristiana Santos. "Do cookie banners respect my choice?: Measuring legal compliance of banners from iab europe's transparency and consent framework". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 791–809.

[130] William Melicher et al. "Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting". In: *Network and Distributed System Security Symposium (NDSS)*. `https://doi.org/10.14722/ndss.2018.23309`. 2018.

[131] *Memory Management in Linux*. `https://www.kernel.org/doc/htmldocs/kernel-api/mm.html`. 2022.

[132] Oslien Mesa et al. "Understanding vulnerabilities in plugin-based web systems: an exploratory study of wordpress". In: *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*. 2018, pp. 149–159.

[133] Daniela Yidan Miao. "PrivacyInformer: An automated privacy description generator for the mit app inventor". PhD thesis. Massachusetts Institute of Technology, 2014.

[134] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[135] Yi L Murphey, Hong Guo, and Lee A Feldkamp. "Neural learning from unbalanced data". In: *Applied Intelligence* 21.2 (2004), pp. 117–128.

[136] Andrew C Myers. "JFlow: Practical mostly-static information flow control". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1999, pp. 228–241.

[137] Y Nan et al. "Identifying User-Input Privacy in Mobile Applications at a Large Scale". In: *IEEE Trans. Inf. Forensics Secur.* (2017).

[138] Yuhong Nan et al. "Finding Clues for Your Secrets: Semantics-Driven, Learning-Based Privacy Discovery in Mobile Apps." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2018.

[139] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. "Apex: extending android permission model and enforcement with user-defined runtime constraints". In: *Proceedings of the 5th ACM symposium on information, computer and communications security*. ACM. 2010, pp. 328–332.

[140] Muhammad Naveed et al. "Inside Job: Understanding and Mitigating the Threat of External Device Mis-Binding on Android." In: *NDSS*. 2014.

[141] Stephan Neuhaus et al. "Predicting vulnerable software components". In: *Proceedings of ACM SIGSAC conference on Computer & Communications Security*. 2007, pp. 529–540.

[142] *New tool automatically finds buffer overflow vulnerabilities*. `https://www.cylab.cmu.edu/news/2021/07/09-SyRust.html`. July 2021.

[143] James Newsome and Dawn Xiaodong Song. "Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software." In: *Network and Distributed Systems Security Symposium*. Vol. 5. Citeseer. 2005, pp. 3–4.

[144] Van Nguyen et al. "Cross Project Software Vulnerability Detection via Domain Adaptation and Max-Margin Principle". In: *arXiv preprint arXiv:2209.10406* (2022).

[145] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. "Nodest: Feedback-Driven Static Analysis of Node.Js Applications". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2019, pp. 455–465.

[146] Midas Nouwens et al. “Consent-O-Matic: Automatically Answering Consent Pop-ups Using Adversarial Interoperability”. In: *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. 2022, pp. 1–7.

[147] Midas Nouwens et al. “Dark patterns after the GDPR: Scraping consent pop-ups and demonstrating their influence”. In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 2020, pp. 1–13.

[148] Maxime Oquab et al. “Learning and Transferring Mid-Level Image Representations using Convolutional Neural Networks”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2014.

[149] Sinno Jialin Pan, James T Kwok, Qiang Yang, et al. “Transfer learning via dimensionality reduction.” In: *AAAI*. Vol. 8. 2008, pp. 677–682.

[150] Xiang Pan et al. “FlowCog: context-aware semantics extraction and analysis of information flow leaks in android apps”. In: *Proc. of USENIX Security’18*. 2018.

[151] Rahul Pandita et al. “WHYPER: Towards Automating Risk Assessment of Mobile Applications”. In: *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. 2013, pp. 527–542.

[152] Muhammad Riyad Parvez. “Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries”. MA thesis. University of Waterloo, 2016.

[153] Corina S Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. “Multi-run side-channel analysis using Symbolic Execution and Max-SMT”. In: *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. IEEE. 2016, pp. 387–400.

[154] Corina S Păsăreanu et al. “Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis”. In: *Automated Software Engineering* 20.3 (2013), pp. 391–425.

[155] Santanu Paul and Atul Prakash. “A framework for source code search using program patterns”. In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 463–475.

[156] Giancarlo Pellegrino et al. “Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1757–1771. ISBN: 9781450349468. DOI: 10.1145/3133956.3133959. URL: https://doi.org/10.1145/3133956.3133959.

[157] Jeffrey Pennington, Richard Socher, and Christopher Manning. “Glove: Global vectors for word representation”. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[158] Henning Perl et al. “Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits”. In: *Proceedings of ACM SIGSAC conference on Computer & Communications Security*. 2015, pp. 426–437.

[159] Matthew E Peters et al. “Deep contextualized word representations”. In: *arXiv preprint arXiv:1802.05365* (2018).

[160] Nam H Pham et al. “Detection of recurring software vulnerabilities”. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 2010, pp. 447–456.

[161] *PHPJoern*. https://github.com/malteskoruppa/phpjoern. 2020.

[162] *Plugin analysis report generated by CHKPLUG*. https://drive.google.com/file/d/119iZP0Ax6HuvX8YiTHXjqtAl1wkpXj1V/view. 2022.

[163] Zhengyang Qu et al. “Autocog: Measuring the description-to-permission fidelity in android applications”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 1354–1365.

[164] Amir Rahmati and Harsha V Madhyastha. “Context-specific access control: Conforming permissions with user expectations”. In: *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM. 2015, pp. 75–80.

[165] Rajat Raina et al. "Self-taught learning: transfer learning from unlabeled data". In: *Proceedings of the 24th international conference on Machine learning*. ACM. 2007, pp. 759–766.

[166] Randy A. Brown. *WordPress vs Other CMS Platforms: How Does WordPress Stack Up Against the Rest?* https://www.elegantthemes.com/blog/resources/wordpress-vs-other-cms-platforms-how-does-\wordpress-stack-up-against-the-rest. Sept. 2018.

[167] Michael T Rosenstein et al. "To transfer or not to transfer". In: *NIPS 2005 workshop on transfer learning*. Vol. 898. 2005, p. 3.

[168] *Rough Audit Tool for Security*. https://code.google.com/archive/p/rough-auditing-tool-for-security/. 2022.

[169] Nicola Ruaro et al. "SyML: Guiding symbolic execution toward vulnerable states through pattern learning". In: *24th International Symposium on Research in Attacks, Intrusions and Defenses*. 2021, pp. 456–468.

[170] Jukka Ruohonen. "A demand-side viewpoint to software vulnerabilities in wordpress plugins". In: *Proceedings of the Evaluation and Assessment on Software Engineering*. 2019, pp. 222–228.

[171] Norman Sadeh et al. *The usable privacy policy project*. Tech. rep. Technical report, Technical Report, CMU-ISR-13-119, Carnegie Mellon University, 2013.

[172] Tara N Sainath et al. "Convolutional, long short-term memory, fully connected deep neural networks". In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2015, pp. 4580–4584.

[173] Hitesh Sajnani et al. "Sourcerercc: Scaling code clone detection to big-code". In: *Proceedings of the 38th International Conference on Software Engineering*. 2016, pp. 1157–1168.

[174] Takahito Sakamoto and Masahiro Matsunaga. "After GDPR, still tracking or not? Understanding opt-out states for online behavioral advertising". In: *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2019, pp. 92–99.

[175] *Samsung SmartThings*. https://www.smartthings.com/. 2023.

[176] Iskander Sanchez-Rola et al. "Can i opt out yet? gdpr and the global illusion of cookie control". In: *Proceedings of the 2019 ACM Asia conference on computer and communications security*. 2019, pp. 340–351.

[177] Cristiana Santos, Nataliia Bielova, and Célestin Matte. "Are cookie banners indeed compliant with the law? deciphering eu legal requirements on consent and technical means to verify compliance of cookie banners". In: *arXiv preprint arXiv:1912.07144* (2019).

[178] Bhaskar Pratim Sarma et al. "Android permissions: a perspective combining risks and benefits". In: *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*. 2012, pp. 13–22.

[179] *Secure Privacy*. https://secureprivacy.ai/. 2022.

[180] Koushik Sen et al. "Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: Association for Computing Machinery, 2013, pp. 488–498. ISBN: 9781450322379. DOI: 10.1145/2491411.2491447. URL: https://doi.org/10.1145/2491411.2491447.

[181] Ali Sharif Razavian et al. "CNN features off-the-shelf: an astounding baseline for recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*. 2014, pp. 806–813.

[182] Faysal Hossain Shezan et al. "CHKPLUG: Checking GDPR Compliance of WordPress Plugins via Cross-language Code Property Graph". In: *Network and Distributed Systems Security Symposium*. 2023.

[183] Faysal Hossain Shezan et al. "NL2GDPR: Automatically Develop GDPR Compliant Android Application Features from Natural Language". In: *arXiv preprint arXiv:2208.13361* (2022).

[184] Faysal Hossain Shezan et al. “TKPERM: Cross-platform Permission Knowledge Transfer to Detect Overprivileged Third-party Applications”. In: *Network and Distributed Systems Security Symposium.* 2020.

[185] Ziqi Shuai et al. “Type and interval aware array constraint solving for symbolic execution”. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 2021, pp. 361–373.

[186] *Silent Breach.* `https://silentbreach.com/gdpr.php`. 2022.

[187] A Prasad Sistla et al. “CMV: Automatic verification of complete mediation for Java Virtual Machines”. In: *Proceedings of the 2008 ACM symposium on Information, computer and communications security.* 2008, pp. 100–111.

[188] *SmartThings Public GitHub Repo.* `https://github.com/SmartThingsCommunity/SmartThingsPublic/tree/master/smartapps`.

[189] *Software assurance reference dataset.* `https://samate.nist.gov/SRD/index.php`. 2018.

[190] *Splint.* `http://www.splint.org/`. 2022.

[191] *SQL Parser.* `https://www.npmjs.com/package/node-sql-parser`. 2022.

[192] Varun Srivastava et al. “A security policy oracle: Detecting security holes using multiple API implementations”. In: *ACM SIGPLAN Notices* 46.6 (2011), pp. 343–354.

[193] Marius Steffens et al. “Don’t Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild”. In: *Network and Distributed System Security Symposium (NDSS).* `https://publications.cispa.saarland/id/eprint/2744`. 2019.

[194] Nick Stephens et al. “Driller: Augmenting fuzzing through selective symbolic execution.” In: *NDSS.* Vol. 16. 2016. 2016, pp. 1–16.

[195] Bo Sun, Akinori Fujino, and Tatsuya Mori. “POSTER: Toward Automating the Generation of Malware Analysis Reports Using the Sandbox Logs”. In: *Proc. of CCS’16.* 2016.

[196] Joshua Tan et al. “The effect of developer-specified explanations for permission requests on smartphone user behavior”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM. 2014, pp. 91–100.

[197] Welderufael B Tesfay et al. “I read but don’t agree: Privacy policy benchmarking using machine learning and the EU GDPR”. In: *Proceedings of the The Web Conference.* 2018, pp. 163–166.

[198] Yuan Tian et al. “Smartauth: User-centered authorization for the internet of things”. In: *26th USENIX Security Symposium (USENIX Security 17).* 2017, pp. 361–378.

[199] David Trabish et al. “Chopped symbolic execution”. In: *Proceedings of the 40th International Conference on Software Engineering.* 2018, pp. 350–360.

[200] Martino Trevisan et al. “4 Years of EU Cookie Law: Results and Lessons Learned.” In: *Proc. Priv. Enhancing Technol.* 2019.2 (2019), pp. 126–145.

[201] *Usercentrics.* `https://usercentrics.com/resources/gdpr-checklist/`. 2022.

[202] Christine Utz et al. “(Un) informed Consent: Studying GDPR Consent Notices in the Field”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.* 2019, pp. 973–990.

[203] Jayakrishna Vadayath et al. “Arbiter: Bridging the Static and Dynamic Divide in Vulnerability Discovery on Binary Programs”. In: *31st USENIX Security Symposium (USENIX Security 22).* 2022, pp. 413–430.

[204] Anthony J Viera, Joanne M Garrett, et al. “Understanding interobserver agreement: the kappa statistic”. In: *Fam med* 37.5 (2005), pp. 360–363.

[205] *Virginia Consumer Data Protection Act (VCDPA).* `https://pro.bloomberglaw.com/brief/virginia-consumer-data-protection-act-vcdpa/`. 2023.

[206] *VulDeePecker algorithm implemented in Python.* https://github.com/johnb110/VDPython. 2022.

[207] *Vulnerabilities by date.* `https://www.cvedetails.com/browse-by-date.php`. 2022.

[208] T Franklin Waddell, Joshua R Auriemma, and S Shyam Sundar. "Make it simple, or force users to read? Paraphrased design improves comprehension of end user license agreements". In: *Proceedings of the CHI Conference on Human Factors in Computing Systems.* 2016, pp. 5252–5256.

[209] 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.* IEEE. 2014, pp. 23–33.

[210] Song Wang, Taiyue Liu, and Lin Tan. "Automatically learning semantic features for defect prediction". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE).* IEEE. 2016, pp. 297–308.

[211] Tielei Wang et al. "IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution." In: *Network and Distributed Systems Security Symposium.* Citeseer. 2009.

[212] Wenhao Wang, Xiaoyang Xu, and Kevin W Hamlen. "Object flow integrity". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* 2017, pp. 1909–1924.

[213] Zhongjie Wang and Shitong Zhu. "SymTCP: eluding stateful deep packet inspection with automated discrepancy discovery". In: *Network and Distributed System Security Symposium (NDSS).* 2020.

[214] Sheng-Han Wen et al. "Enhancing symbolic execution by machine learning based solver selection". In: *Proceedings of the NDSS Workshop on Binary Analysis Research.* 2019.

[215] *What is: Theme.* `https://www.wpbeginner.com/glossary/theme/`. 2022.

[216] Primal Wijesekera et al. "Android permissions remystified: A field study on contextual integrity". In: *24th USENIX Security Symposium (USENIX Security 15).* 2015, pp. 499–514.

[217] Maverick Woo et al. "Scheduling black-box mutational fuzzing". In: *Proceedings of ACM SIGSAC conference on Computer & Communications Security.* 2013, pp. 511–522.

[218] *WordPress database functions.* `https://developer.wordpress.org/reference/classes/wpdb/`. 2022.

[219] *WordPress Plugin.* `https://wordpress.org/plugins/`. 2021.

[220] *WordPress Plugin Handbook.* `https://developer.wordpress.org/plugins/privacy/`. 2022.

[221] *WordPress Plugin Handbook for Data Exporter.* `https://developer.wordpress.org/plugins/privacy/adding-the-personal-data-exporter-to-your-\plugin/`. 2022.

[222] Evan Wei Xiang et al. "Source-selection-free transfer learning". In: *Twenty-Second International Joint Conference on Artificial Intelligence.* 2011.

[223] Xusheng Xiao et al. "User-aware privacy control via extended static-information-flow analysis". In: *Automated Software Engineering* 22.3 (2015), pp. 333–366.

[224] Meng Xu et al. "Precise and scalable detection of double-fetch bugs in OS kernels". In: *2018 IEEE Symposium on Security and Privacy (SP).* IEEE. 2018, pp. 661–678.

[225] Rubin Xu, Hassen Saidi, and Ross Anderson. "Aurasium: Practical policy enforcement for android applications". In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12).* 2012, pp. 539–552.

[226] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. "Generalized vulnerability extrapolation using abstract syntax trees". In: *Proceedings of Annual Computer Security Applications Conference.* 2012, pp. 359–368.

[227] Fabian Yamaguchi et al. "Automatic inference of search patterns for taint-style vulnerabilities". In: *Proceedings of IEEE Symposium on Security and Privacy.* IEEE. 2015, pp. 797–812.

[228] Fabian Yamaguchi et al. "Chucky: Exposing missing checks in source code for vulnerability discovery". In: *Proceedings of ACM SIGSAC conference on Computer & Communications Security*. 2013, pp. 499–510.

[229] Fabian Yamaguchi et al. "Modeling and discovering vulnerabilities with code property graphs". In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 590–604.

[230] Jason Yosinski et al. "How transferable are features in deep neural networks?" In: *Advances in neural information processing systems*. 2014, pp. 3320–3328.

[231] *Your introduction to personally identifiable information: What is PII?* `https://matomo.org/blog/2020/01/your-introduction-to-personally-identifiable-information-what-is-pii/`. 2020.

[232] Le Yu et al. "Autoppg: Towards automatic generation of privacy policy for android applications". In: *Proceedings of the ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. 2015, pp. 39–50.

[233] Amir R Zamir et al. "Taskonomy: Disentangling task transfer learning". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 3712–3722.

[234] Eric Zeng, Shrirang Mare, and Franziska Roesner. "End user security and privacy concerns with smart homes". In: *Symposium on Usable Privacy and Security (SOUPS)*. Vol. 220. 2017.

[235] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. "Using {CQUAL} for Static Analysis of Authorization Hook Placement". In: *11th USENIX Security Symposium (USENIX Security 02)*. 2002.

[236] Yang Zhang et al. "Improving accuracy of static integer overflow detection in binary". In: *International Symposium on Recent Advances in Intrusion Detection*. Springer. 2015, pp. 247–269.

[237] Yajin Zhou et al. "Hey, you, get off of my market: detecting malicious apps in official and alternative android markets." In: *NDSS*. Vol. 25. 4. 2012, pp. 50–52.

[238] Yaqin Zhou et al. "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks". In: *Advances in neural information processing systems* 32 (2019).

[239] Sebastian Zimmeck, Rafael Goldstein, and David Baraka. "Privacyflash pro: automating privacy policy generation for mobile apps". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2021.

[240] Sebastian Zimmeck et al. "Automated analysis of privacy requirements for mobile apps". In: *2016 AAAI Fall Symposium Series*. 2016.

[241] Barret Zoph et al. "Learning Transferable Architectures for Scalable Image Recognition". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018.

[242] Deqing Zou et al. "$\mu$VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection". In: *IEEE Transactions on Dependable and Secure Computing* 18.5 (2019), pp. 2224–2236.

[243] Deqing Zou et al. "mVulPreter: A Multi-Granularity Vulnerability Detection System With Interpretations". In: *IEEE Transactions on Dependable and Secure Computing* (2022).