COMSE 6156 Topics in Software Engineering

Overview of Techniques to Detect Software Security Vulnerabilities in Third-Party Libraries

Tianzhi Huang, th2888

March 2, 2022

Abstract

The number of incidents of cyber attacks targeting and exposing software security vulnerabilities has been increasing over the years. Recently, the log4jShell attack obtains software's control by injecting codes in user's log4j system and results in a mandatory version upgrade for all the log4j users[1][2]. Although techniques like static analysis and security testing have been widely applied to detect software security vulnerabilities within the software's own code base, identifying them in the Third-Party library is still a challenging task and is proved to be essential in terms of protecting the software integrity. In this paper, we will review various techniques and general ideas of detecting security vulnerabilities in the Third-Party library, as well as actions that could be taken to avoid those security incidents from happening. Moreover, we will compare their experimental result by presenting their result data to analyze their performance. Lastly, this paper will present a survey about this topic, analyze the result, and draw conclusions.

Introduction

Software Engineers have been training to maintain software's security standard by applying methods to protect against well-known cyber attacks like SQL Injection, XSS Injection, DDoS attack, and so on [1][2]. While those actions could be easily done in their own software code base, it becomes more challenging to deal with the Third-Party Libraries imported or included by the system. With the limited access to their source code and internal structure, the key to approach this issue becomes clear: how engineers could identify possible security vulnerabilities lying in the Third-Party Libraries. To solve this critical issue, engineers and researchers have been working on existing models and theories and introducing several techniques as capable and realistic solution

candidates, including requirement dependency analysis [3], TPL version control based on Control Flow Graphs [4], and Security-oriented static analysis [5].

In this paper, we will go over some of those techniques and ideas by categorizing them into two aspects: methods could be applied by the software developers, methods or ideas could be applied by those Third-Party Libraries developers. We will introduce their low-level implementations as well as all the technical setups. Then, we will discuss their evaluation experiments and by analyzing the result data. Eventually, we set up a questionnaire survey among both Computer Science students and professional software engineers to learn about their perspectives regarding this topic. By analyzing the result from the questionnaire, we would able to making assumptions based on their attitude and possible suggestions.

What Readers Could Learn From This Paper

By reading this paper, the readers could expect to learn about:

1. Techniques and their implementation targeting detecting software security vulnerabilities in the Third-Party Libraries.
2. Empirical studies and practical implications suggestions for the Third-Party Libraries developers to better secure every distribution and upgrade of their libraries.
3. Evaluation matrices and statistics model derived to evaluate the overall performance of each techniques and how they are comparing to other existing methods.
4. A survey containing personal perspectives and opinions related to this topic from both Computer Science students and professional software engineers.

To conclude, after reading this paper, the readers should be able the gain a comprehensive understanding of the current situation about software security, techniques that could be used to solve the Third-Party Libraries security vulnerabilities issue and their predicted performance, and perspectives from people within the computer science field about this topic.

Background

Horizontal and Vertical Traceability

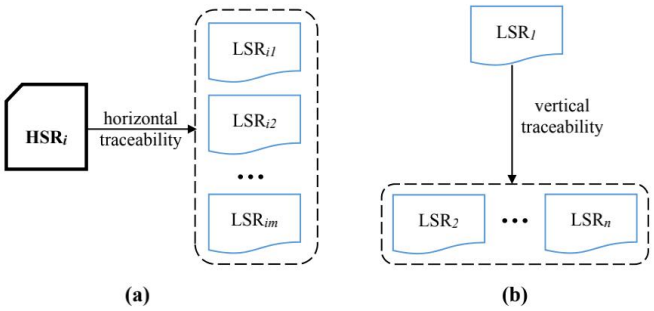


Figure 0: Horizontal and Vertical Traceability [3]

Horizontal Traceability [3][11] refers to a dependency relation occurring in the cross-module

situation. It focuses the potential dependency between different units, modules, sub-systems and is set to determine those dependencies between High-level Security Requirement and its corresponding Low-level Security Requirements. This type of traceability only concerns about all the individual requirements without establishing relations between their interdependencies.

Vertical Traceabilities [3][11], on the other hand, addresses the dependency relation within a single unit, module, or sub-system. It is a system implementation level traceability that focuses on Lower-level Security Requirements and their dependency relations between others. This type of traceability only concerns implementation level requirements and dependencies without caring about all the High-level Security Requirement complied in the system.

Techniques and Implementations

Requirement Dependency Analysis [3]

Requirement Dependency Analysis (RDA) introduces the idea of categorizing High-level Security Requirement (HSR) into five most common and representative categories, automatically tracing candidate set of Low-level Security Requirements (LSR) that compile with the corresponding HSR, and eventually inferring dependencies between set of LSRs with their compiled HSRs are dependent to each other. It not only analyze those requirements in the software own code base, but also includes every usage related to the Third-Party Libraries. This technique recognizes the drawbacks of only having vertical traceabilities [3], which ignores the horizontal dependent relationship between subsystems, and horizontal traceabilities [3], which discards the internal dependency requirements within a unit. It is set to be the complementary of the traditional static dependency analysis and with its semi-automatic processes, saves time and human power involved in the development and testing phases. The RDA emphasizes the importance of completing system-wise dependency analysis thoroughly in order to discover all possible dependency relations between HSRs and LSRs, and illustrates that this process plays a significant role in the later phase of security testing.

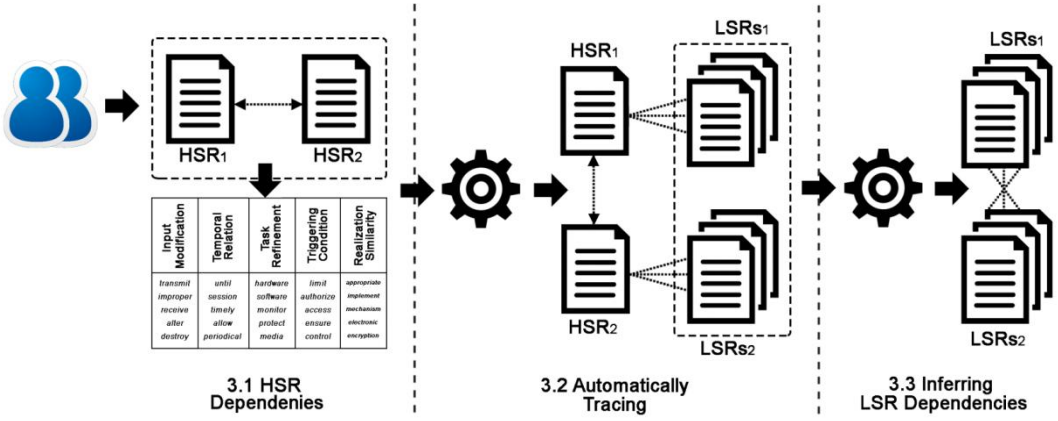


Figure 1a: Three Main Steps of the RDA Implementation [3]

Release Planning [12]	**Interaction Management** [11]	**High-Level Security** (our approach)	# of dependencies HIPAA	FIPS 200
Requires ("printer" requires "driver")	Structure (Req_1 is similar to Req_2)	Input Modification	5	11
Temporal ("add" is done before "delete")	Time (Req_1 is temporally related to Req_2)	Temporal Relation	2	3
Cvalue ("manual" decreases value)	Task (Req_1 describes a task for Req_2)	Task Refinement	2	7
Icost ("waiting" increases cost)	Causality (Req_1 is a consequence of Req_2)	Triggering Condition	4	4
Or ("draw" or "import" an image)	Resource (Req_1 & Req_2 rely on a resource)	Realization Similarity	3	2

Figure 1b: RDA's HSR Dependency Analysis Result From Step 1 [3]

Figure 1a shows the fundamental three steps to complete the RDA process. The first step, HSR dependency analysis, involves a manual process of analyzing system-level dependencies among all the HSRs, and categorizing them into five categories. This step is the only step which requires actual human power to complete and result in the reason why RDA could only be viewed as a semi-automatic technique. The result five categories are shown as the third column in Figure 1b. The second step, Automatically Tracing, describes the process of using pre-selected indicator terms integrated by relevance feedback [3][12] to generate candidate sets of LSRs which all of them are complied with the corresponding HSRs. Basically this step begins with a individual HSR, scans all the LSRs, selects ones which contributes to fulfill this HSR, and stores them in the same set. The final step, LSR dependency inference, refers to the process of interring dependent relations between LSRs with different candidate sets given the assumption that if HSR1 depends on HSR2, every LSR in HSR1's candidate set should have a dependent relation between all LSRs in HSR2's candidate set. This step is very straightforward and produces the desired dependency relationship between all lover level LSRs.

The Requirement Dependency Analysis achieved significant performance enhancement in term of precision and recall of detecting LSRs dependencies. It's underline idea of combining Horizontal and Vertical Traceability together effectively eliminates the drawbacks of insufficient scanning scale and depth that traditional methods have, and by the Scholar@CU use case study discussed in their paper [3], proves the effectiveness of applying this techniques to detect security vulnerabilities including those in the Third-Party code within large-scale system.

ATVHUNTER: Third-Party Libraries Version Detection and Vulnerability Identification [4]

ATVHUNTER is developed as a system of detecting Third-Party Libraries (TPL) versions in android application and, by comparing the versions to a pre-generated TPL version reference database, identify possible vulnerable TPLs [4]. The system, although designed specifically for android application for now, recognizes the difficulties of identifying TPLs in application, and the fact that existing TPL detection methods have not met the optimal performance of solving those challenges. Besides, because its implementation and architecture choices, the ATVHUNTER has the extensibility of applying to all kinds of application by changing the traversing method and extracted metadata format, which contributes the potential of enlarging its compatibility to be deployed for various types of application within different platform. The ATVHUNTER system develops a two-phase detection process to identify all the TPLs' version in the application, and the core technique in this process is to extract and construct the whole Control Flow Graphs (CFG) in order to match the potential TPL to the pre-generated TPL database.

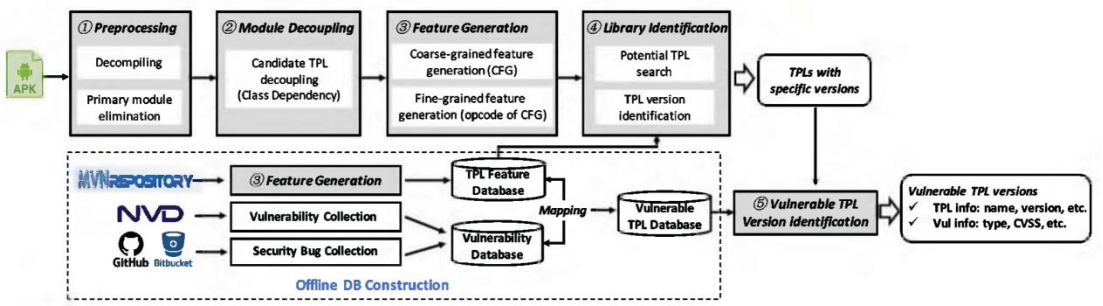


Figure 2: The Process and Workflow of ATVHUNTER [4]

Figure 2 shows the internal workflow and process of the system. The implementation of the ATVHUNTER takes an android application (apk files) as the input parameter, and generates a list of vulnerable TPLs and their corresponding versions based on the reference databases [4]. The system consists of two major phases: TPL-V detection, and Vulnerable TPL-V identification.

The TPL-V detection is responsible for identifying every specific version of TPLs applied and used in the applications [4]. It contains four sub-phases, as shown in the above figure 2: Preprocessing, Module decoupling, Feature generation, and TPL identification. Preprocessing is a two-phase step which, firstly, decompiles and de-structures the input apk file and converts the bytecode file generated from the decompiling phase into the corresponding intermediate representations (IRs). Secondly, it iterates the whole application package, trying to find all the modules under the host namespace, and deleting all of them in order to prevent the interference form the host application. Module Decoupling intends to move the non-primary module in the application into "different independent library candidates" [4] by using the Class Dependency Graph. Feature generation is done after finishing Module Decoupling. It uses both Coarse-grained Feature Extraction and and Fine-grained Feature Extraction [4] to extract all achieved features and generates signature to represent the TPL files. TPL identification phase also consists of two sub-phases. Version Identification uses the signature generated in the previous phase and calculates the similarity ratio over two TPL files to identify the actual version of the TPL being used. Potential TPL Identification searches the reference databases and compare the discovered TPL file against the known existing TPLs. It the system could determine that over 70% of the extracted features re the same, the discovered TPL file will be considered as a potential TPL.

Vulnerable TPL-V identification phase takes on the duty of identifying all the TPLs the system discovers and identifies from the application, comparing them to the constructed vulnerable TPL-Vs database, and eventually identifying those with potential vulnerabilities. This phase consists of two parts: Database Construction and Vulnerable TPL Identification. Database Construction involves constructing the vulnerable TPL-Vs database using Known TPL Vulnerability Collection [4]. Vulnerable TPL Identification is the process of searching for a match in the constructed database with the identified TPL. If there is a match, that TPL will be marked as vulnerable and the system will generate a detailed report about the information of this TPL file.

The ATVHUNTER system recognizes the importance of TPL version's in the process of exposing potential threat and weakness of vulnerable TPLs in the application. It precisely identifies TPL with its version and by comparing them to the vulnerable TPL-Vs database, marks the potential

vulnerable TPL files. The experiment and evaluation in the paper [4] illustrates the efficiency and effectiveness of the system with a relatively high precision, and subsequently highlights its extensibility of usage on other types of applications and platforms.

CRYPTOGUARD: Cryptographic Vulnerabilities Detection in Java Projects [5]

CRYPTOGUARD is a set of innovative program slicing algorithms invented in the paper [5], which targets at the revealing cryptographic vulnerabilities in large-scale Java Projects, both in the project's own code base and their Third-Party imported code. This set of algorithms re-defines and implements the program slicing [5] by identifying and selecting irrelevant elements based on the .class files generated by the JVM, which significantly reduces the false positive cases detected. The algorithms relies on the data dependency graph [5] to keep track of every variables' value and potential reference change, and chooses the most suitable and optimal slicing methods to analysis the .class files as well as the code snippets.

```
1 class PasswordEncryptor {
2
3  Crypto crypto;
4
5  public PasswordEncryptor(){
6   String passKey = PasswordEncryptor
                        .getKey("pass.key");
7    crypto = new Crypto(passKey); (p)
8  }
9
10 byte[] encPass(String [] arg){
11  return crypto.encrypt(arg[0], arg[1]); (p)
12 }
13
14 static String getKey(String src){
15  String key = Context.getProperty(src);
16  if (key == null) {
17    key = "defaultkey";
18  }
19  return key;
20  }
21 }
```

```
22 class Crypto {
23
24  String ALGO = "AES";
25  String ALGO_SPEC = "AES/CBC/NoPadding";
26  String defaultKey;
27  Cipher cipher;
28
29  public Crypto(String defKey){
30   cipher = Cipher.getInstance(ALGO_SPEC);
31   defaultKey = defKey; // assigning field
32  }
33
34  byte[] encrypt(String txt,String key){
35   if (key == null){
36    key = defaultKey; (f)
37   }
38   byte[] keyBytes = key.getBytes("UTF-8");
39   byte[] txtBytes = txt.getBytes();
40   SecretKeySpec keySpc =

  new SecretKeySpec(keyBytes, ALGO);
41   cipher.init(Cipher.ENCRYPT_MODE,keySpc);

42   return cipher.doFinal(txtBytes);}}
```

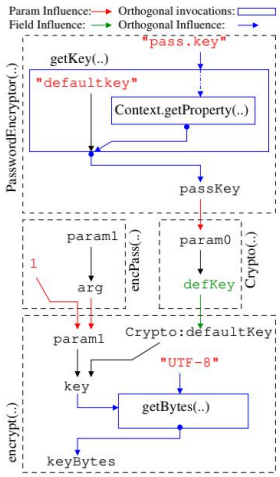


Figure 3a: CRYPTOGUARD Analysis Features[5] Figure 3b: Data Dependency Graph[5]

Figure 3a shows the demonstration of the algorithms running result. Here f represents the potential "influence through the fields" [5], and p represents "influence through the method parameters"[5]. Figure 3b shows an example of the data dependency graph of codes in figure 3a used by the program slicing refinement algorithm. By generating this data dependency graph as the input resource, the algorithms have the capabilities of traversing this graph with every notable static or dynamic allocated fields, as well as every parameters in the method, in order to trace their reference or value change which may lead to potential security vulnerabilities.

No	Vulnerabilities	Attack Type	Crypto Property	Severity	Our Analysis Method
1	Predictable/constant cryptographic keys.	Predictable Secrets	Confidentiality	H	↑ slicing & ↓ slicing
2	Predictable/constant passwords for PBE		Confidentiality	H	↑ slicing & ↓ slicing
3	Predictable/constant passwords for KeyStore		Confidentiality	H	↑ slicing & ↓ slicing
4	Custom Hostname verifiers to accept all hosts	SSL/TLS MitM	C/I/A	H	↑ slicing (intra)
5	Custom TrustManager to trust all certificates		C/I/A	H	↑ slicing (intra)
6	Custom SSLSocketFactory w/o manual Hostname verification		C/I/A	H	↓ slicing (intra)
7	Occasional use of HTTP		C/I/A	H	↑ slicing
8	Predictable/constant PRNG seeds	Predictability	Randomness	M	↑ slicing & ↓ slicing
9	Cryptographically insecure PRNGs (e.g., java.util.Random)		Randomness	M	Search
10	Static Salts in PBE	CPA	Confidentiality	M	↑ slicing & ↓ slicing
11	ECB mode in symmetric ciphers		Confidentiality	M	↑ slicing
12	Static IVs in CBC mode symmetric ciphers		Confidentiality	M	↑ slicing & ↓ slicing
13	Fewer than 1,000 iterations for PBE	Brute-force	Confidentiality	L	↑ slicing & ↓ slicing
14	64-bit block ciphers (e.g., DES, IDEA, Blowfish, RC4, RC2)		Confidentiality	L	↑ slicing
15	Insecure asymmetric ciphers (e.g, RSA, ECC)		C/A	L	↑ slicing & ↓ slicing (both)
16	Insecure cryptographic hash (e.g., SHA1, MD5, MD4, MD2)		Integrity	H	↑ slicing

Table 3c: Output Result and Analysis [5]

The Table 3c shows the a sample analysis result of running CRYPTOGUARD. In the result report, type of cryptographic vulnerabilities are indicated in the column “Vulnerabilities” [5]; the attack type and crypto property is being identified. The algorithm marks the security level of the vulnerabilities from High (H) to Low (L), accompanying with the analysis method it chooses to reveal the corresponding vulnerabilities. Here, ↑ represents backward slicing and ↓ represents forward slicing.

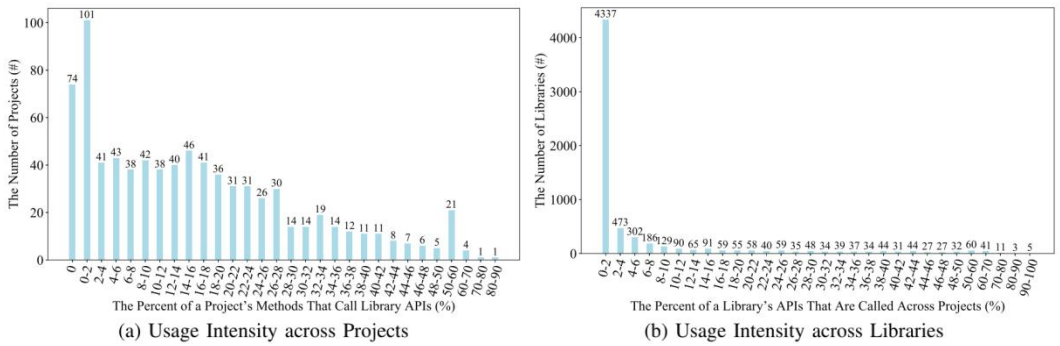
The CRYPTOGUARD focuses on algorithms reform and innovations, integrating program refinement and slicing techniques into the usage of data dependency graph, and being able to identify cryptographic vulnerabilities with acceptable precision and speed. Since its design and implementation reply more on generics internal algorithms, it proves the possibility of detecting vulnerabilities in both original code base and Third-Party Libraries, as well as the extensibilities of using on projects developed by other programming languages.

Empirical Studies and Practical Implications Suggestions

Aside from introducing and inventing new system, techniques, or algorithms, other researchers have conducted a variety of empirical studies trying to understand the current situation within the field, approaching the Third-Party Library security vulnerabilities issue from the perspectives of insufficient human action inputs, unfollowing standard rule, and other possible human factors, providing realistic practical implications suggestions based on the study results.

Usages, Updates and Risks of Third-Party Libraries [6]

The paper [6] carries out a series of Third-Party library usage analysis, Third-Party library update analysis, and Third-Party library risk analysis [6] with the scope of open-source Java projects in order to get a picture of situation of Third-Party library management in major Java projects. They quantify their study result and represent them using bar chart for better visualization of the data. By analyzing the data and comparing with others, they present the following findings and corresponding practical implications suggestions to the software engineers.



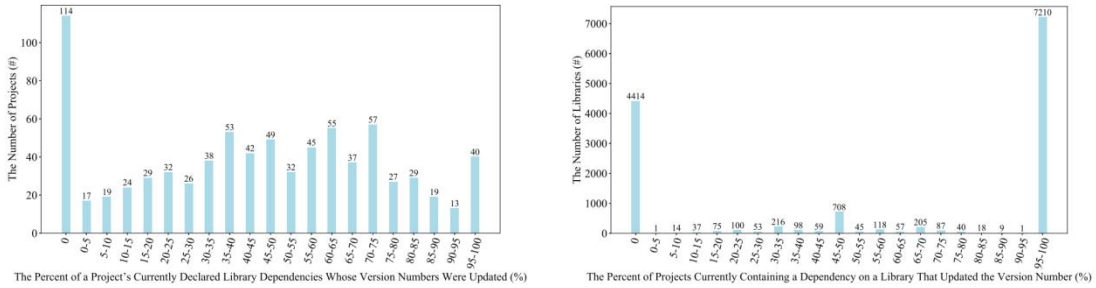
(a) Usage Intensity across Projects (b) Usage Intensity across Libraries

Figure 4a: Usage Intensity across Projects and Libraries [6]

Finding 1 By analyzing the result distribution of the usage intensity in figure 4a, the paper

concludes that projects often have a moderate dependency on the library API calls; and in order for those API to be used in the project, programmers sometimes imports the whole library as resources, which requires extra efforts when the library gets updated.
Suggestion 1 Based on the situation described in Finding 1, the paper suggests that Third-Party Libraries developers should better utilize and allocate their library resources in terms of evolution; and project developer should prevent including unused features from the libraries.

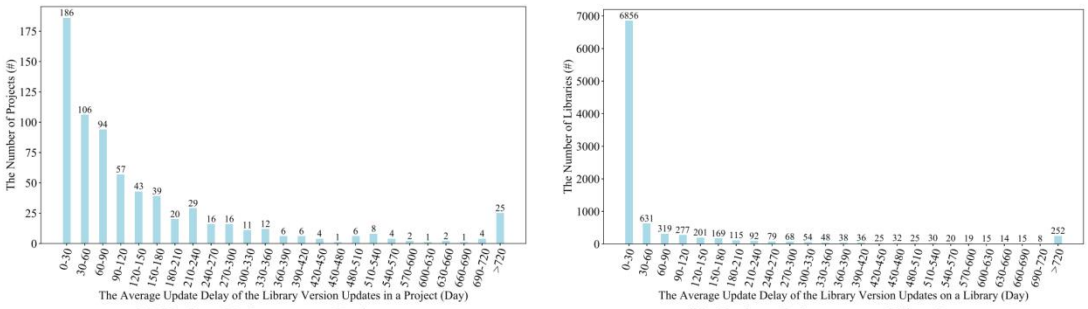
Finding 2 Beside finding about API usage in a project using figure 4a, the paper also concludes that one-third of the projects are involved in using multiple version of the same library and using only snapshot version of the library, which would potentially cause dependency conflicts and result in the increasing of human power to resolve.
Suggestion 2 The paper suggests that programmers should integrate extra tools to monitor and manage them better in term of multi-version and snapshot usage.



(a) Update Intensity across Projects (b) Update Intensity across Libraries

Figure 4b: Update Intensity across Projects and Libraries

Finding 3 Based on the analysis using figure 4b above, the paper concludes that more than half of the imported libraries in half of the studied project never get updated; and one-third of the libraries used in more than half of the selected projects are not updated.
Suggestion 3 The paper suggests that programmers update practice should be strengthened, and their awareness of the essential of updating libraries constantly should be raised [6].

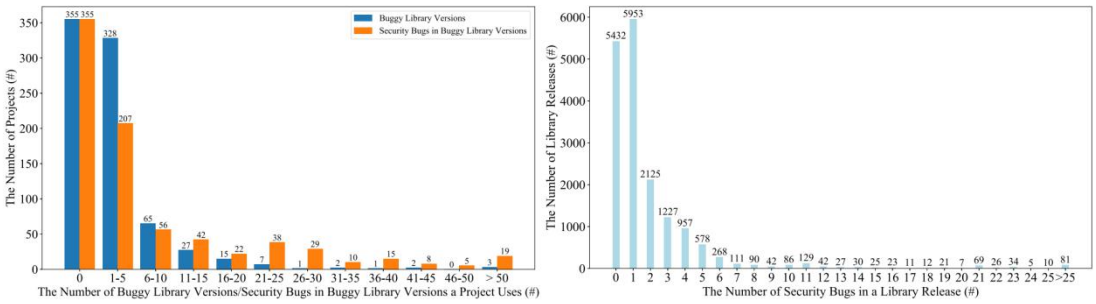


(a) Update Delay across Projects (b) Update Delay across Libraries

Figure 4c: Update Delay across Projects and Libraries [6]

Finding 4 By analyzing the update delay across projects and libraries data in figure 4c, the paper concludes that project developers nowadays tend to have a much slower reaction to both new releases of the libraries and actions should be taken when those new releases happen, which increase the risk of using outdated libraries.

Suggestion 4 The paper suggests regulations or guidance should be introduced to guide the actions when new library releases occur, and those regulations should better be enforced to guide the developer to take necessary actions.



(a) Usage Risk across Projects (b) Usage Risk across Library Releases

Figure 4d: Usage Risk across Projects and Library Releases [6]

Finding 5 By analyzing the usage risk across projects and library releases data in figure 4d, the paper concludes that over 50% of the projects using Third-Party Libraries have bugs regarding security issues; and over 67% of the libraries releases being used by projects contain bugs regarding security issues.

Suggestion 5 The paper suggests that project developers including testing personals should enhance their ability to deliver less-bug projects; and library developer should also monitor those bugs in every releases and fix them as soon as possible.

Frequency and pattern of Developers updating Third-Party Libraries in Mobile Apps [7]

One of the issue and observation that the above paper [6] addresses is the fact that programmers rarely update the Third-Party Libraries whether a new version of the library has been released. Based on this fact, this paper [7] goes one step further, trying to understand mobile application programmers' behavior in the following aspects: 1) how often do developers update the library version. 2) if the programmers do update the libraries, what type of Third-Party Libraries are getting the most or fewest updates. 3) is there a pattern that programmers follow while updating the libraries. The paper investigates 291 actively used mobile applications and presents its study results in the form of charts and tables, providing result analysis and practical implications suggestions for the developers.

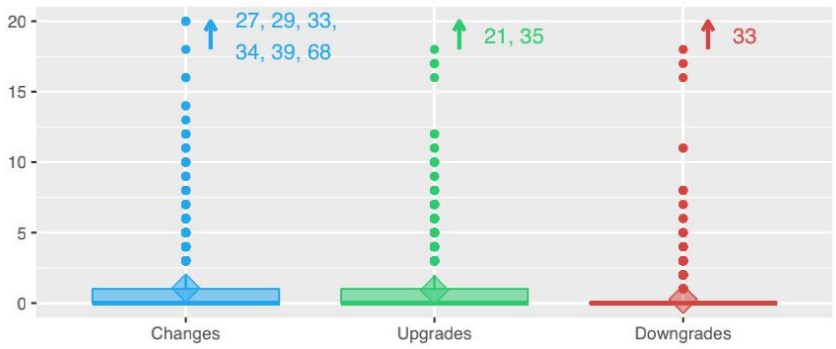


Figure 5a: Libraries Changes, Updates, Downgrades [7]

Finding 1 By analyzing the number of libraries changes, updates, and downgrades in those selected 291 applications, the paper concludes that only 2% of all the commits are for library updates, which supports the previous findings [6]. They prefer to perform an total upgrade of the whole libraries to another version, and sometime they even downgrades the version to solve incompatible issue and dependency conflicts.

Category	Changes	Upgrades	Downgrades
Graphical User Interface	808	607	201
Utilities	478	351	127
HTTP	60	38	22
Page Navigation	42	24	18
JSON	30	18	12
Annotations	23	23	0
Google Services	15	12	3
Date and Time	14	12	2
Device	14	9	5
HTML Parser	9	7	2

Figure 5b: 10 types with more changes [7]

Category	Changes	Upgrades	Downgrades
Defect Detection	1	1	0
Bug Fix	1	1	0
Network	1	1	0
Protocol Buffers	1	1	0
SQL	1	1	0
Test Automation	1	1	0
Notification	2	1	1
Barcode	2	2	0
Cryptographic	2	2	0
Event Bus	2	2	0

Figure 5c: 10 types with less changes [7]

Finding 2 By presenting types of libraries getting more and less version changes in figure 5b and figure 5c, the paper concludes that libraries related to graphical UI and utilities of supporting development tools are getting the most version updates. The reason may be the fact that developers tend to maintain the latest UI tendencies, and to incorporate constantly changed Android version. On the other hand, libraries related to defect detection, network, and else shown in figure 5c rarely gets updates because there may lack of available version to perform update.

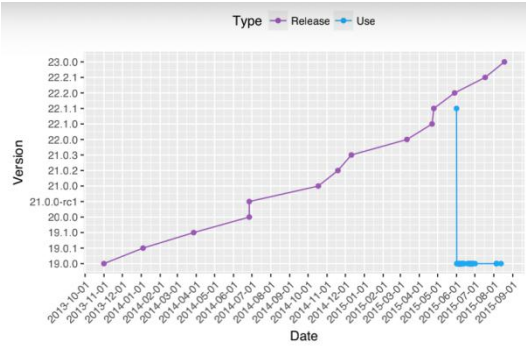


Figure 5d: Example of "jump down" pattern [7]

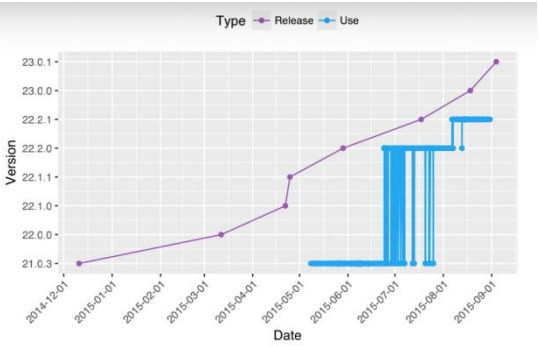


Figure 5e: Example of "back & forth" pattern[7]

Finding 3 By exploring those two update patterns described in figure 5d and figure 5e, the paper concludes that developers, while dealing with Third-Party Libraries updates, tends to follow peculiar patterns. Over 63% of the cases where libraries used are never updated after their first ever introduction to the application, and only 13% of them are getting constantly updates performed by the developers.

The study result of the paper reveals severe issues about application developers' insufficient or inadequate actions about updating Third-Party Libraries when required. Based on the facts and the possible reason to cause them, the paper [7] proposes four practical implication suggestions for the communities:

Suggestion 1 More empirical studies about this issue is needed to produce a more comprehensive

understanding of the underline problems.

Suggestion 2 Consider having automatic technical support for the Third-Party Libraries updates, which could include automatic update of the application and immediate notification when new versions become available.

Suggestion 3 The awareness and effort of updating libraries constantly should be prioritized, and developer should properly rank the update required libraries according to their emergency or potential threat level.

Suggestion 4 Developers should get a sense of the trends of the library update and be aware of these potential changes while developing the application.

More Generalized Findings and Suggestions [8][9][10]

Beside the developers behaviors discussed by the previous papers, the following papers [8][9][10] stand on a higher perspective and consider more generalized issues as well as provide guidance and suggestions. In this paper [8], instead of choosing actual application as the study cases, it selects Apache HTTP server and Apache Tomcat to study Third-Party security vulnerabilities among patch releasing and analyze CVE entries [8] to verify fixing commits. It pays attention to three general aspects of the vulnerabilities fix: who actually fixes those vulnerabilities, how many times needed to perform this fix, and what is the procedure if it.

The figure 6a and 6b show the relationship between vulnerabilities fixers' experience and average team's experience, and fixers' ownership who perform bug fixing. By analyzing the data in the figures, the paper concludes that in most cases, the

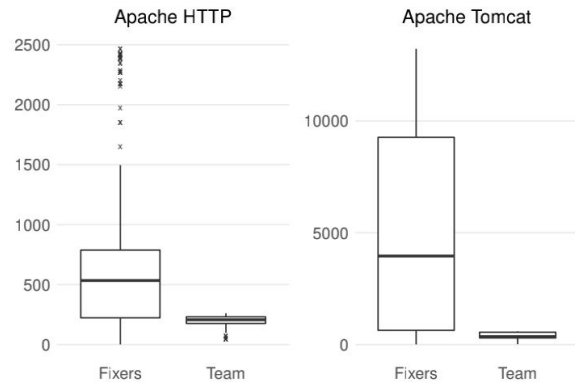


Figure 6a: Vulnerability fixers' vs. mean team's experience [8]

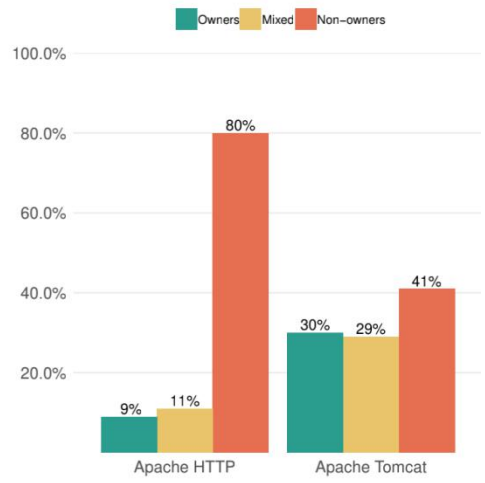


Figure 6b: fixers' ownership of modified files [8]

developers who are responsible for fixing those vulnerabilities are experienced developers beyond average level, and they almost are not the owner of the file caused or affected by the vulnerabilities. Moreover, the paper conducts researches about days for vulnerabilities to expose [8] and days between first and last commit, and concludes that although most of the HTTP and Tomcat vulnerabilities are fixed before releasing the CVE entries, there still exists some incomplete and incorrect fixes. By studying number of commits done and files/line changes in a single vulnerability fix, the paper draws the conclusion

that in most cases, vulnerability patches' size are often small for several commits, and they are local [8].

Elaborating on the study results, the paper summarizes the lessons learned from the case studies. Software security vulnerabilities including those in the Third-Party Library are often easy to fix, but discovering and testing could be very hard to do. When comes to vulnerability fixing, it is better to assign experienced developers to take on the responsibility, and this choice will make the fixing process more efficient. Besides, although vulnerability databases contains essential information about known vulnerable patches, it seems to be hard to establish links between them and fixing commits. This paper [9] also suggests a better use of existing static analysis tools, for examples those installed in modern IDEs, to help identify those vulnerabilities. Programmers who are not familiar with a certain type of IDE may not fully utilize all the tools and resources provided to spot potential vulnerabilities, which limits their vulnerability preventing capabilities. Another possible suggestion about using additional tools, proposed in this paper [10], is to develop a platform that tracks all the previous libraries updates, and alert developers when the time duration of a specific library not getting updated exceeds a certain threshold, and when any library's latest patch becomes available.

Techniques Performance Experiments and Evaluations

Requirement Dependency Analysis

In order to assess the performance of Requirement Dependency Analysis (RDA) detecting and identifying Low-level Security Requirements (LSR), the paper conducts an experiment quantitatively evaluating its recall, precision, and F2 value [3]. It chooses five open source and actively running projects from the domain of public health and education, each with different type of LSR. Two approaches are compared in this experiment, the traditional Vertical Requirement Tracing (VRT), and the proposed method Hybrid Requirement Tracing (HRT). Besides, in order to determine the best type of relevance feedback[12] to use, each method is implemented by four relevance feedback variants: TF-IDF (no relevance feedback), Standard, Adaptive, and Term-Based. The experiment results are shown in figure 7.

Project	Approach	Automated Requirements Tracing Method											
		TFIDF			Standard RF			Adaptive RF			Term-Based RF		
		R	P	F_2	R	P	F_2	R	P	F_2	R	P	F_2
CARE2X	VRT	0.67	0.42	0.59	0.63	0.67	0.64	0.71	0.63	0.69	0.77	0.64	0.74
	HRT	0.64	0.49	0.60	0.71 $*$	0.57	0.68 $*$	0.77 $*$	0.61	0.73 $*$	0.88 $**$	0.64	0.82 $**$
iTrust	VRT	0.64	0.55	0.62	0.65	0.61	0.64	0.73	0.65	0.71	0.79	0.67	0.76
	HRT	0.72 $*$	0.53	0.67 $*$	0.73 $*$	0.57	0.69 $*$	0.75	0.63	0.72	0.91$**$	0.71$*$	0.86 $**$
WorldVistA	VRT	0.52	0.42	0.50	0.53	0.47	0.51	0.62	0.48	0.59	0.72	0.46	0.65
	HRT	0.64 $*$	0.47 $*$	0.60 $*$	0.65 $**$	0.48	0.61 $**$	0.67 $*$	0.51	0.63 $*$	0.87 $**$	0.52 $*$	0.77 $**$
Scholar@UC	VRT	0.69	0.37	0.59	0.74	0.38	0.62	0.71	0.37	0.60	0.74	0.41	0.64
	HRT	0.72	0.43 $*$	0.63	0.81	0.44 $*$	0.69	0.82 $*$	0.43	0.69 $*$	0.89 $**$	0.45	0.74 $**$
Moodle	VRT	0.43	0.44	0.43	0.54	0.46	0.52	0.58	0.45	0.55	0.67	0.43	0.60
	HRT	0.49 $*$	0.47	0.49 $*$	0.62 $*$	0.49	0.59 $*$	0.65 $*$	0.49	0.61 $*$	0.79 $**$	0.51 $*$	0.71 $**$
Average	VRT	0.59	0.44	0.55	0.62	0.51	0.59	0.67	0.52	0.63	0.74	0.52	0.68
	HRT	0.64	0.47	0.60	0.70	0.51	0.65	0.73	0.53	0.68	0.87	0.57	0.78

Figure 7: Experiment Result for RDA [3]

Analyzing the data in figure 7, several observations and conclusions could be drawn. First of all,

on average cases, the HRT method, proposed in the paper [3], significantly improves both recall and F2 values; and it is also arguable that HRT method outperforms the traditional VRT method in every single selected test project. Secondly, if we look at a single method, we could possibly conclude that method implemented by Term-Based relevance feedback achieves better overall performance in terms of higher F2 value. Last but not least, although HRT method manages to improve recall and F2 value in each case, it does not improve the precision notably. In other words, from the above data, we could confidently conclude that HRT method helps improve the recall, but it is unclear whether it performs the same improvement on precision. One underline reason, which is described in the paper [3], is that the assumption of existence of dependencies between LSRs in dependency inference phase may cause false positive cases, which brings the precision level down.

ATVHUNTER

The experiment setup for the ATVHUNTER is designed to answer three questions regarding the performance: 1) Effectiveness, 2) Efficiency, and 3) Obfuscation-resilient Capability. The paper conducts three experiments with three different evaluation matrices to assess ATVHUNTER's performance. It chooses four publicly available and latest TPL detection tools (LibScout, OSSPoLICE, LibPecker, and LibID) [4] and compares their running result with the proposed ATVHUNTER. The experiment results are shown in the figure 8.

Tools	Library-level			Version-level		
	Precision	Recall	F1	Precision	Recall	F1
ATVHunter	98.58%	88.79%	93.43%	90.55%	87.16%	88.82%
LibID	98.12%	68.45%	80.64%	68.70%	66.42%	67.54%
LibScout	97.10%	46.65%	63.02%	44.82%	43.50%	44.15%
OSSPoLICE	97.91%	43.39%	60.13%	88.83%	42.25%	57.26%
LibPecker	93.16%	57.82%	71.35%	60.35%	57.67%	58.98%

Tool	ATVHunter	LibID	LibScout	OSSPoLICE	LibPecker
Q1	15.92s	51.43s	30s	33.48s	12168s
Mean	66.24s	59616s	83s	2052.34s	16396s
Median	47.78s	9286s	64s	80.42s	16632s
Q3	90.30s	38300s	100s	226.60s	23292s

Tool	No Obfuscation	Obfuscation			
		Renaming	CFR	PKG FLT	Code RMV
ATVHunter	99.26%	99.26%	90.13%	99.26%	75.57%
LibID	12.93%	12.93%	0.03%	1.58%	2.49%
LibScout	88.75%	88.75%	18.24%	17.69%	17.69%
OSSPoLICE	85.62%	85.62%	23.04%	39.52%	48.86%
LibPecker	98.79%	98.79%	86.63%	73.56%	79.28%

Figure 8: Experiment Results for ATVHUNTER [4]

From the test result shown in figure 8, we could draw several conclusions. Firstly, among all the tools being tested, ATVHUNTER achieves the highest value in precision, recall, and F1 value, both in Library-level and Version-level. The improvement that ATVHUNTER is able the provide is significant and it is persuasive and convincing to conclude that the ATVHUNTER is more effective than most of the TPL tools which are currently used. Secondly, ATVHUNTER generates the detection result with the lowest required time in all four measurements. Besides, by comparing the actual numbers, we could conclude that this efficiency improvement is also significant and it possesses a dominant efficiency advantage against other test tools. Lastly, ATVHUNTER is able to achieve the highest detection rate among five test tools, and it has descent performance in both

non-Obfuscation and Obfuscation scenarios. To summarize, ATVHUNTER's performance advantages in effectiveness, efficiency, and detection rate has been well illustrated and the improvements it brings are significant and persuasive.

CRYPTOGUARD

The experiment set to evaluate CRYPTOGUARD's performance is designed to compute four measurements: False Positive Rate, False Negative Rate, precision, and recall. Three commonly used and publicly available tools (CrySL, Coverity, SpotBugs) [5] are selected as the test tools. The experiment is conducted under two set of benchmarks: CRYPTOAPI-BENCH: Basic, and CRYPTOAPI-BENCH: Advanced. Each benchmark has different set of rules and know GTP (ground truth positive) which indicates the true number of positive cases. The experiment results is shown in figure 9.

Tools	CRYPTOAPI-BENCH: **Basic**							CRYPTOAPI-BENCH: **Advanced**													
	GTP:14			**Summary**				**Inter-Proce. (Two) GTP: 13**			**Inter-Proce. (Multiple) GTP: 13**			**Field Sensitive GTP: 13**			**FP Test/ Correct Uses GTP: 3**			**Summary**			
	TP	**FP**	**FN**	**FPR**	**FNR**	**Pre.**	**Rec.**	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**	**FPR**	**FNR**	**Pre.**	**Rec.**
CrySL[44]	10	4	4	40.00	28.57	**71.43**	**71.43**	10	3	3	0	12	13	0	1	13	0	2	3	85.71	76.19	**35.71**	**23.81**
Coverity[1]	13	0	1	0.00	7.14	**100.0**	**92.86**	3	0	10	3	0	10	1	0	12	0	0	3	0.00	83.33	**100.0**	**16.67**
SpotBugs[2]	13	0	1	0.00	7.14	**100.0**	**92.86**	0	0	13	3	10	10	0	0	13	0	0	3	76.92	92.86	**23.08**	**7.14**
CRYPTOGUARD	14	0	0	0.00	0.00	**100.0**	**100.0**	13	0	0	13	0	0	13	0	0	3	0	0	0.00	0.00	**100.0**	**100.0**

Figure 9: Experiment results for CRYPTOGUARD [5]

The test results shown in figure 9 is very straightforward and several conclusions could be made based on the data. First of all, CRYPTOGUARD significantly outperforms other three tools in both basic and advanced benchmark. The improvement, in terms of precision and recall, is obvious in basic set since both Coverity and SpotBugs achieve relatively descent performance. In the advanced benchmark, CRYPTOGUARD's dominance performance is proved by the pool result generated by other three tools. Notably, in both benchmarks, CRYPTOGUARD performs perfectly with 100% precision and recall, which is a indication of an overall excellent detection performance.

Survey Study and Result Analysis

Methodology and Setup

While reading all the papers describing efforts towards better detecting software security vulnerabilities in Third-Party Libraries (TPL), I understand the idea of improving TPL security could be approached from both inventing techniques and tools and regulating developers profession habits and behaviors. Different people with different professional or academic experience may hold distinct perspectives about this topic, and the gap of knowledge level between them will impact their vision about ways of solving this issue. It becomes meaningful for me to conduct a survey for Computer Science students and software engineers to ask about their opinion and experience of facing TPL vulnerabilities. The reason why this survey is only for Computer Science students and software engineers is because this topic is technical and specific within the software development scope. Understanding the underline context of the survey, as well

as technical terms in every questions requires certain level of computer science or related background knowledge. Although by reading through this paper, all the readers with different knowledge background are expected to gain a comprehensive understanding of the topic, it could be difficult for them to answer those questions by reflecting their own experience. Meanwhile, those answers are somehow meaningless when I analyze the survey results. For instance, if a participant knows nearly nothing about software development, or general computer science knowledge, I will assume that they will also know nearly nothing about TPL. Those answers are not contributing to the process of recording meaningful responses.

The form of the survey is an online Google Form and is distributed via share link. The full content of the survey questions could be found in the Appendix section. The survey contains multiple choice and short answer questions aiming for better reflecting participants perspectives. In order to protect privacy, the survey is taken anonymously and no personal information is recorded. By the time this paper is written, the survey received 19 responses, which, not a relatively large sample though, still contains meaningful information and I am able to draw reasonable conclusions and make assumptions based on the survey results. More responses will be analyzed and included in the paper's appendix if the survey receives them in the future.

Question Design

In this section, I will explain the design and underline intention of those questions, and what kind of perspectives I am expecting to get from them. The full list of the questions could be found in the Appendix. Question 1 to 4 are designed for knowing the background of the participants, making sure that they possess sufficient knowledge based and are familiar with the topic discussed both in the paper and survey. On thing to mention here is that by Question 1, I intend to have students and software engineers split evenly in order for better spread of the sample, and the actual distribution result is acceptable. Question 5 to 6-1 is especially designed for TPL topics. By answering those questions, I want to learn the participants' experience about using TPL and their habits in terms of updating TPL, then compare the result to the findings in paper [6][7] regarding the frequency of developer updating their used TPL. Question 7 to 8 asks participants whether they have experience dealing with vulnerabilities exposure incidents, and if they do, what are the main causes of that incident and whether it is related to TPL vulnerabilities. Question 9 to 10, in a higher level, ask participants about the possible reasons that cases software vulnerabilities and what actions should the developers take to prevent it from happening, as well as how many of them think it is also related to TPL issues. With this design, I am able to analyze responses from individual question, trying to find pattern and relations between answers and participants background. Besides, from the survey, I could be able to see the difference of answers between two major groups: students and professional software engineer, analyzing how experience difference could impact their answers and perspectives.

Survey Result and Analysis

In this section, I will analyze the survey result and present several findings and conclusions that I am able to reasonably draw from it. The raw data of all the responses could be found in the

Appendix, and for the purpose of analyzing, I will show results of some questions in chart to discuss the response patterns and elaborate on them for conclusions.

There are 19 responses in total recorded by the time this paper is written. By studying the result from Question 1 to Question 4, several observations could be made. Among those 19 participants, 12 (63.2%) of them are students and 7 of them are professional software engineers with more than 5 years of programming experience. Within those 12 students, 3 of them are undergraduate students with 0 to 2 years of programming experience and the other 9 of them are masters students with around 5 years of programming experience. Besides, indicated by the result of Question 4, almost all of them are familiar with the topic being discussed and seem to have sufficient knowledge base to answer those specific techniques-related questions. This separation between the numbers of student participants and software engineers, although not ideal, is sufficient to provide meaningful information based on the sample size.

The result of Question 5 indicates that all of them have used TPLs in their projects or systems. As the follow-up questions of this topic, the result of Question 6 and Question 6-1 are shown in figure 10 below.

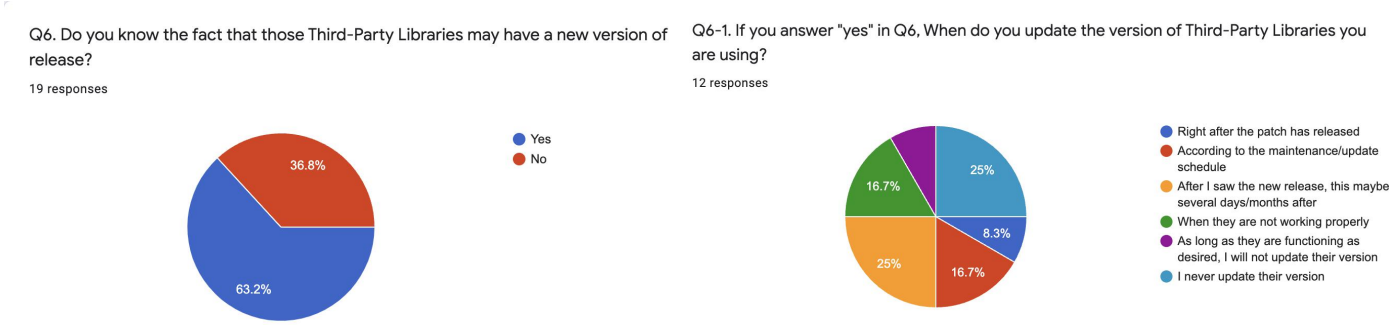


Figure 10: Results of Question 6 and Question 6-1

By studying the results above, 12 participants are aware of the situation that TPLs constantly have new releases, and all 7 software engineers answers “yes” in Question 6. This presents the observation that professional software engineers understand the fact that new patch are frequently released for TPLs. The result that the rest of the 5 students who do not know this concept yet is understandable if their projects are for the academic or educational purpose, or they are always provided with the newest version of the TPLs by their professors. When studying the result of Question 6-1, interesting things happen. Among all those participants who are aware of this situation, only 3 of them, who are all software engineers, perform required actions (update TPL right away, update according to schedule) [6], and 50% of them either not update them at all, or update them only when problems or bugs occur. This finding is somehow consist with the findings in paper [6][7], as they conclude that developers do understand the situation and necessity of updating TPLs as soon as possible, but their performance and actions are often below requirements and standards. Our survey result is consistent with this findings, and I assume one of the reason is that developers tend to trust those TPLs developed by big companies and assume that they could never go wrong, or expose vulnerabilities to their software. Thus, unless there are some problems or bugs discovered, they will tend to think everything is fine so far.

Question 7 and Question 8 indicate that only 3 (15.8%) of the participants have experienced software security attacks targeting their system. Those three are all software engineers and they have experienced attacks such as DDoS on server, SQL injection, and repeatedly requesting resources. This result is understandable since for collage students, their projects are often run locally, simple in structure, and quite small in term of project scale. Those projects are for educational purpose and will not be chosen as the target. Besides, their projects do not create value or actual profit, and it is simply not worthy for the attackers to attack those students projects. Even if the participants are professional software engineers, usually their work is not building a system from the stretch. They are developing incrementally on a well-constructed and mostly well-projected actively running system, the chance for them to being in an under-attack situation is very low.

Question 9 to Question 10 asks for possible causes of software vulnerabilities and methods to solve this issue, especially TPL vulnerabilities. The result of Question 9 is shown in the figure 11 below and The result of Question 10 could be found in the Appendix.

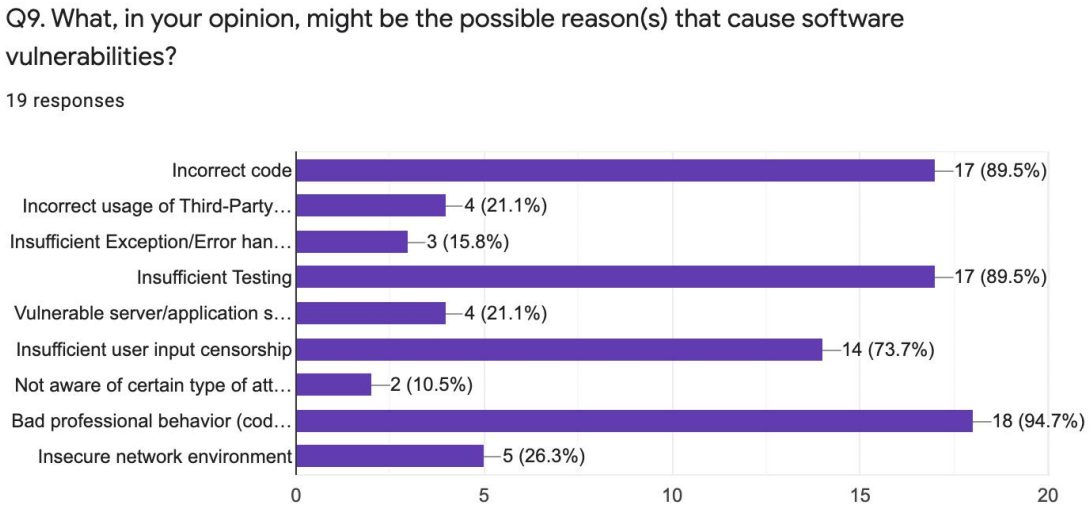


Figure 11: Result of Question 9

By studying the data above, we could see the top four selected reasons are: Incorrect code, Insufficient testing, Insufficient user input censorship, and bad professional behaviors. Majority of the participants believe that the issue could be solved on the developers, and technical issues such as coding, testing, and input checking are the major reasons. This reflects a perspective of the participants that they tend to focus on their side of work to approach this issue. Instead of worrying about TPL reliability, they believe make proper actions in the system they are developing should be the key point. If the system itself does not have severe security threats or weaknesses, or its has effective monitoring or recovering tools that provide protections, those vulnerabilities can not make damage to their system. In the papers [6][7], they also discuss what TPL releases should be responsible for. They approach this issue in another angle saying that in addition to better protection for the application, the developers developing those TPLs should make those releases more secure and reliable and fix vulnerabilities as soon as possible. This point could also consistent with the survey results since for those TPL developers, the reasons why they

fail to deliver secure and reliable TPL releases, are possible also these four selected by the participants. These four reasons cause application developers to face vulnerabilities and attacks, and cause TPL developers to release vulnerable and insecure patches.

Conclusion

This paper first introduces related background information and knowledge for the readers to better understand the context of the discussed topic. Then, it presents the high level ideas and low level implementations of three technical solutions, Requirement Dependency Analysis, ATVHUNTER, and CRYPTOGUARD, to serve as the potential methods of detecting TPL vulnerabilities. Besides technical solutions, the paper reviews several empirical studies which provide practical implications suggestions of solving this issue based on their quantitatively analyzing software developers behaviors, attitude, and working patterns. This paper also presents a online-conducted questionnaire survey of collage students and professional software engineers aiming for getting their perspectives and opinions about general software security vulnerabilities issue, and specifically those occurring in TPLs. The survey results are presented and analyzed, with several conclusions and assumptions drawn from them.

What I Learn from This Paper

By reading all the papers, references, conducting a survey and analyzing its result, and putting all those together as the author, there are several lessons that I learned from this process. Firstly, I learned how to choose a single and specific aspect to approach a high level issue. Recall the three techniques discussed in the paper, they are all designed with smaller domain. The RDA is constructed within software in health and education domain, and CRYPTOGUARD is only for Java projects. They all try to solve small-size problems with good performance, then find ways to extend to other larger domains or platforms. Secondly, I learned how to search for relevant papers and references, how to summarize important information, and how to organize them into a single paper with reasonable order. Last but not least, I gained understanding of various concepts and ideas about solving software securities problems. There is no simple answer when talking about real-world issues and all possible solutions should be considered and studied carefully.

There could potential be some future works regarding this topics: 1) develop fully automatic monitoring and detecting system to perform the vulnerabilities detection. 2) introduce specific written regulations or guidance guiding developers' behaviors or decision making. 3) set up vulnerable TPL databases for future reference and vulnerable TPL identification.

Bibliography

[1] LogShell CVE-2021-44228 Detail, National Vulnerability Databases, Retrieved December 10, 2021.
https://nvd.nist.gov/vuln/detail/CVE-2021-44228
[2] Log4j Vulnerability, Log4j Vulnerability FAQs, Secureworks. Retrieved December 17, 2021.
https://www.secureworks.com/blog/log4j-vulnerability-faqs
[3] W. Wang, F. Dumont, N. Niu and G. Horton (2020), Detecting Software Security Vulnerabilities via Requirements Dependency Analysis, in IEEE Transactions on Software Engineering, doi: 10.1109/TSE.2020.3030745.
https://ieeexplore-ieee-org.ezproxy.cul.columbia.edu/document/9222252
[4] X. Zhan et al., ATVHunter: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications, 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 1695-1707, doi: 10.1109/ICSE43902.2021.00150.
https://ieeexplore-ieee-org.ezproxy.cul.columbia.edu/document/9402031
[5] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. 2019. CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19). Association for Computing Machinery, New York, NY, USA, 2455–2472.
https://dl.acm.org/doi/10.1145/3319535.3345659
[6] Y. Wang et al., An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects, 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020, pp. 35-45, doi: 10.1109/ICSME46990.2020.00014.
https://ieeexplore-ieee-org.ezproxy.cul.columbia.edu/document/9240619
[7] P. Salza, F. Palomba, D. Di Nucci, C. D'Uva, A. De Lucia and F. Ferrucci, Do Developers Update Third-Party Libraries in Mobile Apps?, 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC), 2018, pp. 255-25510.
https://ieeexplore-ieee-org.ezproxy.cul.columbia.edu/document/8972993
[8] V. Piantadosi, S. Scalabrino and R. Oliveto, Fixing of Security Vulnerabilities in Open Source Projects: A Case Study of Apache HTTP Server and Apache Tomcat, 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), 2019, pp. 68-78, doi: 10.1109/ICST.2019.00017.
https://ieeexplore-ieee-org.ezproxy.cul.columbia.edu/document/8730158
[9] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu and H. R. Lipford, How Developers Diagnose Potential Security Vulnerabilities with a Static Analysis Tool, in IEEE Transactions on Software Engineering, vol. 45, no. 9, pp. 877-897, 1 Sept. 2019, doi: 10.1109/TSE.2018.2810116.
https://ieeexplore-ieee-org.ezproxy.cul.columbia.edu/document/8303758
[10] A. Cobleigh, M. Hell, L. Karlsson, O. Reimer, J. Sönnerup and D. Wisenhoff, Identifying, Prioritizing and Evaluating Vulnerabilities in Third Party Code, 2018 IEEE 22nd International Enterprise Distributed Object Computing Workshop (EDOCW), 2018, pp. 208-211, doi: 10.1109/EDOCW.2018.00038.

https://ieeexplore-ieee-org.ezproxy.cul.columbia.edu/document/8536124
[11] Wikipedia. Requirements Traceability. (2021). Jturpen007
https://en.wikipedia.org/wiki/Requirements_traceability
[12] Wikipedia. Relevance Feedback. (2022). BrownHairedGirl
https://en.wikipedia.org/wiki/Relevance_feedback

Appendix

Venn Diagram:

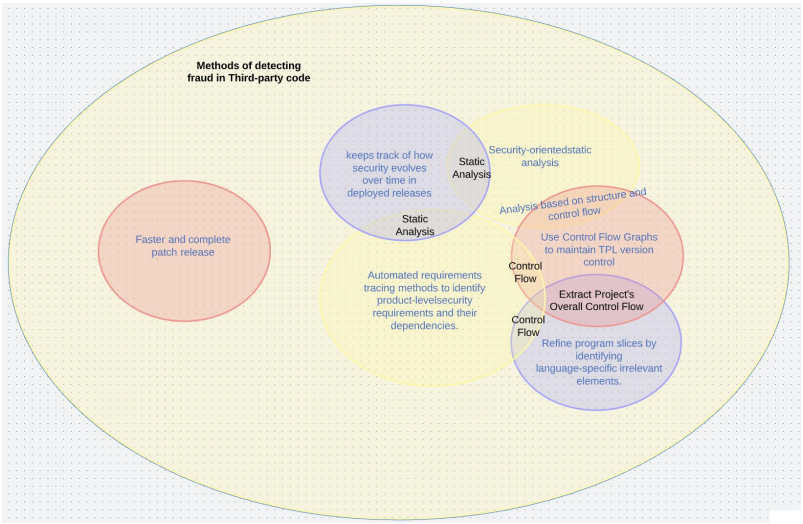


Figure 1. Methods of Detecting Vulnerabilities in Third-Party Libraries

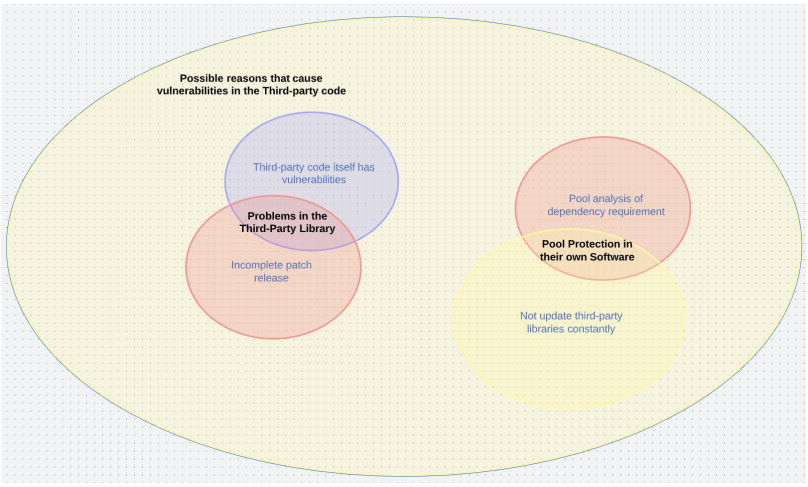


Figure 2. Reasons that cause Security Vulnerabilities

Survey Questions

Link:
https://docs.google.com/forms/d/e/1FAIpQLSdpKDPVVLjbz7UbFNRFlSKskea7S4l_WcRScz27A5VVap-KXw/viewform?usp=sf_link

Survey Result Raw Data

Link:
https://docs.google.com/spreadsheets/d/14MYbe1yNID8jux0Tswoe10NUNk7vj_gnq6C3KCilk20/edit?usp=sharing