Detecting Insecure Implementation of Vulnerable Code Snippets found on Stackoverflow

Mazharul Islam

ABSTRACT

Online programming discussion platforms such as Stack Overflow have a rich source of ready to use code snippets for software developers. It is the de-facto place where developers go to find solutions from the coding snippets given as answers to posted problems by the online developer community. However, previous research work has shown that developers have a tendency to directly copy-paste insecure code snippets from Stack Overflow into their production level code. As a result without any countermeasures Stack Overflow is becoming one of major sources of vulnerability of production level code. To address this problem, in this project, we tackle the problem of analyzing code snippets found on Stack overflow. This is challenging since code snippets from Stack overflow are often erroneous, incomplete, and lack dependencies which makes it harder to analyze them by state-of-the-art static tools. Our goal is to build a static analysis tool that can identify common insecure patterns found on a dataset containing a collection of 1.6K code snippets from Stackoverflow.

1 INTRODUCTION

Stack Overflow (SO) is regarded as one of the most popular online helping platforms for software developers [1]. SO seeks to help by creating an eco-system of online developer community. In this eco-system one developer can ask for solutions to the to the problems one is facing, while other developers can interact by posting code snippets, advices, to solve those asked problems. As such, SO has become a rich source of ready-to-use code snippets for software developers. This richness has caused SO to enter the agile software development cycle allowing fast prototyping, and an efficient workflow. Particularlly novice developers treasure the quick-direct help from the community providing easy ready-to-use code snippets. Interestingly copying-pasting code snippets into production level source code found on SO, is a common practice, not only shared by the inexperienced developers, but also by large parts of the developer community. Ecouraging, sometimes experienced developers can potentially promote best- practices by engaging with other developers, providing high quality code snippets, and eventually improving code quality on a large basis.

Unfortunately for secure coding practices, there is a sharp contrast. Fischer et al. [9] quantitatively measured that an Android developer seeking help on SO, can find accepted popular code snippets suggesting insecure X509 certificate validation, misusing Android's cryptographic API as shown in Figure 1. Moreover a developer struggling with handling Java Spring security framework's configuration errors, can find code snippets suugessting turning off CSRF security protection entirely to avoid errors.

Insecure code snippets found on Stack Overflow itself is not a serious problem. However these insecure code snippets can naturally enter the development cycle by developers who are copy-pasting them into the production level code – casuing vulnerabilities in

Figure 1: An insecure code snippet on Stack Overflow which contains vulnerable implementation of `X509TrustManager` interface. The wide acceptance of this insecure code snippet can be understood by the green tick and 106 updvotes.

softwares. Moreover to add salt to the problem, these insecure code snippets are being accepted, promoted even by users having high repuration [11]. Even so condescending comments are being directed at security conscious users [12]. This gives the developers a high level of false confidencee while copy-pasting insecure code-snippets that they are widely accepted by online community, and hence, would not cause any security problems.

To solve this problem, we need a way to flag the insecure patterns present on code snippets. This will help developers to be more cautions before copy pasting insecure code snippets. This type of flagging can also draw more attention to the developers before upvoting or advising for an insecure code snippet. Consequently, this can promote writing secure code snippets in Stack Overflow eco-system. However there is no existing tool used by Stack Overflow to analyze and flag a code snippet when it contains potential insecure patterns.

Towards solving this problem, in this paper, we aim to develop a static analysis tool which can identify which part of the code snippet is insecure and suggest secure alternative suggestions to developers. However analyzing code snippets for insecure patterns presents some unique challenges. Since code snippets are oftentimes erronous, incomplete, we have to apply code repair techniques to be able to convert them to an intermediate representation (IR) before running any static analysis. We also need to identify which insecure patterns are common.Therefore, we first compile a list of 8 most insecure patterns related to code snippets in Java language by studying the literature. We then apply simple lexical parsing repairs to the code snippets before converting them to an intermediate representation (IR) language called Jimple. Finally we use a combination of keyword matching, and backword flow analysis based detection method to idenitify the 8 most common insecure patterns in code snippets collected from SO.

In summary, we make the following the contributions in the paper:

- We present 8 common insecure patterns apearning most frequently in Java language code snippets posted on SO by surveying the literature (Section 2).
- We develop code repair tecniques from simple parsing fixes, and apply them on code snippets from SO to convert them to Jimple intermidate representation (IR). (Section 3.1).
- We run keyword matching and backword flow analysis based detection method to identify the presence of 8 insecure patterns (Section 3.3).
- We manually verify our code repair and detection accuracy on a collection of 1.6K code snippets from SO (Section 4).

The rest of the paper is organized as the following: section 2 describes 8 most common insecure patterns found on code snippets, and explains why they are insecure, section 3 describes our methodology and section 4 presents experimental evalvuation results. We conclude the paper by presenting some complelling discussions on ML based vulnerability detection models (section 6), limitations of our study (section 5), and future work (section **??**).

2 THREAT MODEL

Insecure Pattern #	Description	Vulnerability
1	AES default encryption mode ECB	Side channel attack
2	Insecure cryptographic hash	Collision attack
3	Abuse of X509TrustManager Verifier Interface	SSL/TLS MitM attack
4	Weak key length	Brute force attack
5	Static/constant/predictable keys/IV	
7	Presence of AllHostNameVerifier	SSL/TLS MiTM
8	Turning of CSRF protection	CSRF attack

Table 1: The 8 most insecure patterns found on code snippets.

We will now summerize the 8 common insecure patterns our method aims to detect in the comming paragraphs and in Table 1. For each insecure patterns, we will also describe the security risks it presents, and its secure usage from the literature. This will give us a sense what we want our method to detect i.e, the presence of insecure patterns or the absence of secure usage.

2.1 AES default encryption mode ECB

AES is one widely adopted and used encryption standards in the developer community. Therefore it is no surprise that a large number of code snippets uses AES for encryption [4]. In Java an instance of AES can be created using `javax.crypto.Cipher` class. However `javax.crypto.Cipher` class uses Electornic Codebook (ECB) as the default mode of operation when "AES" is passed as transformation parameter to `getInstance` method (as shown in listing 3). While ECB-encrypted ciphertext allows random access to each block, it can also leak information via side channel attacks [8]. However developers being unaware of this default behavior insecure behaviour of AES, share code snippets without any considerations that uses insecure ECB mode for encryption. Instead developers should be using Block Chaining (CBC) or Galios/Counter Mode (GCM) which are not vulnerable to side channel attacks.

2.2 Abuse of X509TrustManager Verifier Interface

X509TrustManager Verifier interface is popular among developers to instantiate TrustManager class. Ideally a secure implementation of X509TrustManager should i) throw exception after validating a cetificate in `checkServerTrusted` method, ii) provide a valid list of certificates in `getAcceptedIssuers` method, and iii) throw exceptions for self signed certificates. However while writing code snippets developers tend to leave empty methods to implement the X509TrustManager interface. As a result the X509TrustManager interface acceptes any certficate including the ones which are not signed by a trusted cettificate authority. This enables a provision for Man-in-the-middle (MitM) attacks.

2.3 Insecure cryptographic hash

A cryptographic hash function produces fixed-length unique alphanumeric string called message digest for any arbitrary long message. This unique message digest can be used latter for verifing important crypto properties of the message e.g., message integrity, digital signature, and authentication. However if two different messages produces the same message digest i.e., a collusion happens, then attcker can compromise these crypto properties. A cryptographic is broken if attacker has systemic practical way to produces collusion for different message. The list of popular but broken hash functions includes SHA1, MD4, MD5, and MD2. These hash functions produce collisions that cause cryptographic vulnerabilities, and hence should be avoided. However while writing code snippets developers have been using these popular broken hashes as shown in listing 1.

2.4 Absence of performing hostname verification

Ideally to perform a hostname verification, developer has to implement the `javax.net.ssl.HostnameVerifier` by using `java.net.ssl.SSLSession` parameter inside the `verify` method. However in many cases this `verify` method is always set to return true as shown in listing 4. The reason being while writing code snippets for brevity this dummy return true will not throw any excpetions. However this type of workaround can cause security threats such as URL spoofing attacks. URL spoofing makes it simpler for numerous cyber-attacks (e.g., identity theft, phishing) [3].

2.5 Weak key length

The strength of asymetric encryotion (e.g., RSA, ECC) depends on using sufficiently large key length. Since 2015, NIST recommends a minimum of 2048-bit keys for RSA an update to the widely-accepted recommendation of a 1024-bit minimum since 2002 [2]. This ensures that the key space is large enough to prevent any practical brute force attack. However while writing code snippets developers have been using key length of less than 2048 disregarding this recommendation as shown in listing 5.

2.6 Static/constant/predictable keys/IV

Predictable keys/ initialization vectors (IV) are a major source insecurity in the code snippets. Raw keys and raw IVs created

from empty byte arrays are easily guessable by attackers. Additionally some code snippets derive keys directly from simple and insecure passphrases as shown in listing 6. Static constant keys are suspectible to leaks. As oftentimes attackers can decompile the application and get the static hardcoded keys. To avoid this kind of attacks, developers should avoid using static constant keys. `javax.crypto.spec.SecretKeySpec` and `javax.crypto.spec.PBEKeySpec` are two popular ways to generate secret keys used for encryption. Both of these API takes a byte array to generate the secret keys. However if the byte array is constant or hardcoded inside the code, the adversary can easily read the cryptographic key and may obtain sensitive information. This is the same case for storing keys in a keystore using `java.security.KeyStore API`. The secret keys by which is key stored is locked for safely storing the keys should take a byte array is not static.

2.7 Presence of AllHostNameVerifier

`org.apache.http.conn.ssl.SSLConnectionSocketFactory` provides a static field ALLOW_ALL_HOSTNAME_VERIFIER to allow accepting all certificates. As this is a very easy way to avoid errors, in code snippets developers insensibly uses them frequently without considering the insecurity associated with using it as shown in listing 7.

2.8 Turning of CSRF protection

Cross site request forgery (CSRF) is a serious attack that tricks the a web browser by abusing the brower cookie authentication mechanism to execute privilege unwanted actions. To protect against such attacks ideally CSRF-Token should be included in all POST, PUT, DELETE requests. However the from code snippets related to Java Spring security framework, we have found that the developers tend to turn of the CSRF protection forcefully to avoid getting errors (see listing 8).

3 METHODOLOGY

In this section, we will discuss the pipeline we follow to detect insecure patterns in code snippets as shown in Figure 2. We will first discuss about repairing the code snippets (section 3.1), and then converting the repaired to code snippets to an Intermidiate Representation (IR) to run analysis (section 3.2). Lastly we will finish by describing the techniques we have applied on the converted IR to detect the insecure patterns (section 3.3), which we have previously described in section 2.

3.1 Code Repair

While writing code snippets as answers to posted questions, developers tend to be concise and short. The reason being long code snippets has lower chance of being accepted and upvoted by others in online platforms such as Stack Overflow. Within a few lines of code, developers try to convey the intent hinting at a working solution by asuuming everything other are in place to for sucessful compilation. However this very mindset of developers can leave syntatic errors, missing classes in the code snippets. As a result, converting these code snippets to IR for analysis becomes difficult.

For identifying insecure patterns for which only keyword searching is sufficient (e.g., Rule 7, 8 as shown in Table 2) this is not a problem since we don't need to convert them to any IR. However for idenitfying insecure pattern (e.g., Rule 1-6 as shown in Table 3) which requires running analysis this poses a problem. Therefore to identfy them, we need to add some repairs to the code snippets. For the purpose of this paper, we have applied the following repairs to the code snippets.

3.1.1 Syntatic repair. To remove the syntatic error present on the code snippets we do the following syntatic repairing.

- We remove illegal characters (e.g., ">", "<", "&", """ "&#x27", etc). Many of these illegal characters appered as the dataset was crawled from Stackoverflow website's raw HTML and HTML sanitizes some characters which are used in the code snippets. Also some code snippets have comments without any comment sign, and dots to implify some code would be here which not relevent to question posted.
- Some code snippets do not have match brackets, and extra quotes for strings.
- Some code snippets have `@Overide` notation implying it is implementing an interface. However the partial program analysis tool we will discuss to convert code snipeets to IR, can not handle `@Override` notation.

3.1.2 Missing package, class, and method names: Partial Program Analysis (PPA) tool which we have used to convert the code snippets to IR, can not consume a lines of code missing classname, package name, method names. Therefore we applied the following repairs.

- If the code snippet is missing any class name, we wrap the code snippet inside a public class name. If the code snippet already has a public class, we rename the file according to that public class.
- If the code snippet does not have any package name, we place the public class in a package, and add the package name to the code snippet. If the code snippet has package name, we create proper directory strcuture according to the package name and place the code snippet there before converting them to IR.
- We also add dummy implementation of missing methods as developers tend to have methods names in code snippets but does not give any implementation within the code snippets.
- Finally, we load the some popular crypto classes in Java to the runtime of PPA tool which are imported, implemented by code snippets frequently. This helped us to avoid missing class name, unknown interface error thrown by PPA in many cases.

3.2 Converting code snippets to IR

After repairing the code snippets, we tried to convert them to IR grammer named Jimple. Jimple [19] is a 3-address intermediate representation that has been designed to simplify analysis. Jimple was inspired from SIMPLE an AST to represent C statements. To convert the code snippets we used a tool named partial program analysis (PPA). Dagenais et al. developed PPA [5] with goal of analyzing only subset of a program source code which matches with our use case of analysing code snippets. PPA can infer types where types are not present that subset of the code. In case of

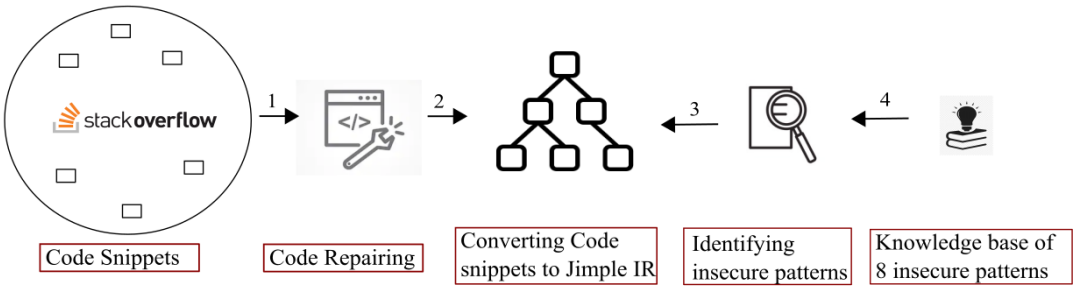

Figure 2: Overall processing pipleline of insecure pattern detection (1) collecting 1.6K code snippets from Stack Overflow, (2) applying code repairs on collected code snippets, (3) converting code snippets to Jimple 3-address intermidate representation, (4) detecting 8 insecure most insecure patterns using keyword searching and backword flow analyis.

failure PPA will place special type "MAGICCLASS", "MAGICCLASS", and "MAGICMETHOD". This is necessary since without types it is not possible to build the abstract syntax tree, and eventually convert the code snippet to Jimple for a strongly typed language such as Java. As PPA can overcome this problem by inferring types of the objects used in the subset of the program source code, it can convert the subset program source code. We leverage PPA after making the code repairs presented in previous subsection 3.1. Otherwise a large number of code snippets was throwing errors as PPA can not handle errounos code snippets.

The idea is to feed the Jimple representation of the code snippet to Soot – a state-of-the-art program analysis tool [17]. Soot API can consume a Jimple representation, and perform data flow analysis which is as discussed in the next subsection 3.3, required for detecting insecure patterns.

3.3 Identifying insecure patterns

In this subsection we will discuss the two techniques we have used to identify the insecure patterns discussed in section 2. Specifically we haved used two techniques found in the existing literature. One of them is keyword based detection, and the former one is using back flow sensitive analysis.

Rule No	keywords
7	SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER
8	*.csrf.disable()

Table 2

3.3.1 Key word base analysis. In keyword based analysis, we want to write an regex which can capture the common way developers write the insecure patterns, and then searching in the code snippets for matching the written code snippets. This method was used by Rahman et al. [15] to detect insecure practices present in Python code snippets on Stackoverflow. Although being a simple technique, it worked surprising well for them i.e, does not introduce any false positives. However, in our case, as we will show in the next, that capturing all the insecure pattern by writing regex can introduce false-positives even simple code snippets.

Rule No	Slicing criteria
1	KeyGenerator.getInstance(*)
2	MessageDigest.getInstance(*);
3	public void checkClientTrusted public void checkServerTrusted public X509Certificate[] getAcceptedIssuers()
4	keyPairGenerator.initialize(keySize);
5	public boolean verify
6	new SecretKeySpec(keyBytes, "AES") *.load(*.openStream(), new String(keyBytes).toCharArray()); new PBEKeySpec(new String(keyBytes).toCharArray(),*);

Table 3

≪**Mazharul 3.0:** Can you show the slicing critera in Jimple form?≫

Therefore, according to our manual observation, we can detect only two insecure patterns using keyword searching. This follows from the resoning that Rule 7 and 8 can be written by any developer, in the exact pattern as shown in Table 2. For detecting the other 6 insecure patterns, we have to restore to backword flow program analysis as described next.

3.3.2 Backword flow base analysis. We use the backword flow analysis intrdocued by Rahaman et al. [14] in their CryptoGurd Project. They introduced specialize def-use analysis [20] based on program slicing techniques [6] to detect 16 common cryptographic API misuses in Apache, and Android projects. Def-use dataflow analysis builds a dependency relation based on the definition and use statements. Given a slicing criteria, which is a statement, or a parameters of an API, backword flow analysis computes the set of program statements that affects the slicing criteria in terms of data flow. The key design choice, hence, here is to specify special function invokation places as the slicing crteria. The slicing criteria used for paper are highlighted in Table 3.

Now we will detail why simple keyword based analysis is not sufficient for detecting rules 1-6 as they can introduce FP even for simple rules. To demonstrate this, consider the example code snippet shown in listing 1 corresponding to insecure pattern rule 2 – detecting insecure broken cryptographic hashes MD5, MD4, MD2, SHA1. We can use keyword searching based technique on the name of broken hashes, and successsfully detect that the insecure pattern that code snippet in listing 1 have used broken hash.

However it will introduce Fasle positive for same insecure pattern on the code snippets shown in listing 1.As there are multiple ways developers can use these broken hashes, unlike rule 7-8, we set `MessageDigest.getInstance(*);` as the slicing criteria. Then we start backword def-use analysis to see if any of the program sets affects the paramters of `MessageDigest.getInstance(*);`, and has a value of equal to name of the broken hash.

```
...
MessageDigest md = MessageDigest.getInstance("MD5");
md.update(str.getBytes());
....
```

Listing 1: A code snippet where keyword based detection work well

```
...
int flag = 2;
MessageDigest md = MessageDigest.getInstance("MD5");
if(choice > 1){
   md = MessageDigest.getInstance("SHA-256");
}
md.update(str.getBytes());
....
```

Listing 2: A code snippet where keyword based detection introduces FP

We modified the code base of as CryptoGuard[1] to achieve our analysis on code snippets. This is for two reasons. Firstly, CryptoGuard already has the basic skelection for use-def analysis, and we just have to change the slicing criteria. Secondly, CryptoGurd uses Soot as the underlying program analysis enginee. Hence we can provide our generated Jimple IR using the PPA tool to CryptoGuard, and CryptoGurd's program analysis enginee Soot can do backword analysis based on the slicing criteria defined by us.

Figure 3 shows one such example. Taking `SecretKeySpec` as the slicing criteria, we identify the set of statements which affects the first parameter of `SecretKeySpec` – which is `MyDifficultPassw` here. Eventurally we will backtrack to the definition program set and can reason that it is randomly generated - rather a hard coded secret. Since this makes the `SecretKeySpec` class object `sks` predictable, we have detected the presence of insecure pattern 6.

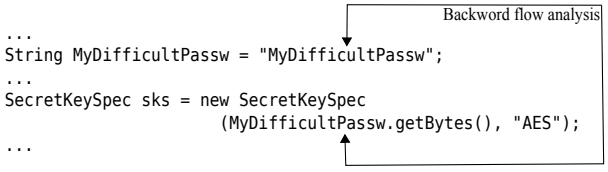

Figure 3

4 RESULTS

In this section, we will first describe our data-collection process. Next we will show the code repair, and detection accuracy techniques on a small subset of this collected dataset.

[1]https://github.com/CryptoGuardOSS/cryptoguard

4.1 Data collection

We use dataset from two previous sources [9, 11]. Both of these dataset contain code snippets posted on Stackoverflow. Fisher et al. crawled 1,161 code snippets posted on Stackoverflow related to Andrioid security [9]. They considered a code snippet related to Android security if the code snippets makes API calls to one of the security services such as Java cryptography, Java secure Communications, public key infrastructure X.509 certificates, and Java authentication - authorization services. The popular crypto libraries used by Andriod developers such as Bouncy Castle, SpongyCastle, Apache TLS/SSL, keyczar, jasypt, and GNU Crypto were also included.

Meng et al. extracted 503 code snippets from 22,195 Stackoverflow posts by filtering the posts based on votes, duplications, and absence of code snipeets [11]. In total our study is baded on the dataset by combining these two. Our dataset contains 1,664 code snippets. The timeline of these code snippets are from 2008-2017. The average size of the collect code snippets is 46 LoC having a very high standard deviation of 58.

4.2 Code repair success rate.

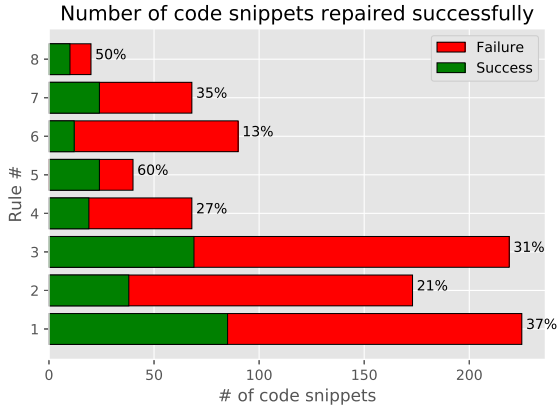

Figure 4: Percentage of code snippets successfully repaired.

We called a code snippet successfully repaired if we can convert it to a Jimple IR. In summary we are able to convert 813 code snippets out of collected 1.6K code snippets – a repair accurarcy of 30.31%. We also categorize each converted into one more or category based on the secure/insecure use of the rule presented on Table 1. Figure 4 illustrates the percentage of code snippets for each categorize, we have been able to parsed (i.e., convert to Jimple IR), using the code repair techniques discussed in section 3.1.

4.3 Insecure pattern detection accuracy.

After converting each successfully repaired code snippets to Jimple IR, we detect rule 7, and 8 using keyword searching as mentioned previously. However for rule 1-6, need to apply backword flow analysis. To do this we give the Jimple IR to our tool. Out tool is built on top of CryptoGurd. CryptoGurd uses Soot as its program analysis

enginee. Using Soot's `JimpleAST` API [2], we can enable backword flow analysis given the slicing criteria from Table 3. The idea is track the special method invokations parameter from the slicing criteria. In this way we can inspect the set of program statements which are affected by this slicing criteria and figure out the presence of insecure patterns.

Table 4 shows the total number of code snippets for of the 8 rules (they sum up to more than 813 as some code snippets has two or more rules). Parsed column refers to the number of code snippets, we have been able to convert to Jimple IR after applying the code repair discussed in section 3.1. We also shows the TP, FN, FN along side the precision, and recall. As it can seen we have not been able to run backword flow analysis for rule 3. The reasons are explain the in the appendix 10.

Rule no	Parsed	TP	FP	FN	Precision	Recall
1	225	85	0	2	-	-
2	173	38	0	4	-	-
3	219	-	-	-	-	-
4	68	19	0	8	-	-
5	40	24	0	3	-	-
6	90	12	0	7	-	-
7	68	24	4	0	-	-
8	20	11	3	0	-	-

Table 4: Table

5 LIMITATIONS

5.1 Limitations from conservative datasets.

We only considered a small subset of code snippets in Java available on Stackoverflow. As a result the 8 insecure patterns we have considered are tailored to this small subset of code snippets. A more rich dataset would have allowed to capture more naunce insecure patterns.Moreover, we only analzed code snippets with Java/Andrioid tags. If we have analyzed code snippets from other popular languages [13], we would have been able to discover more insecure patterns.

5.2 Limitations from program repair techniques.

The program repair techniques in section 3.1, we have applied on erronous code snippets, before converting them to Jimple IR, are simple semi-automated simple parsing repairs. It would have been better to see if state-of-the-art the program repair techniques can be adjusted to repair them. Especially automated program repair tools which can apply quick single edit fixes (e.g, Google's OSS-Fuzz and Microsoft's Springfield project) are quite enticing to apply for fixing erronous code snippets as discussed in this great survey paper [10].

[2]https://www.sable.mcgill.ca/soot/doc/soot/jimple/parser/JimpleAST.html

5.3 Limitations due to PPA/Jimple grammer.

The partial program analysis tool (PPA) [5] we have used theto run analysis depends on Jimple. Jimple, a 3-address IR, was designed to handle, and simplify several difficulties while performing optimizations on the stack-based Java bytecode directly [19]. However, the simple grammer of Jimple on which we are running backword flow analysis can not handle annonymous class. Annonymous classes are the most common way developers declare the X509TrustManager interface (insecure pattern # 3). Because of Jimple inability to express annonymous classes, we can not detect the insecure pattern #3.

5.4 Limitations from disregarding comments.

Insecurity in production level code can also come from insecure advices given in text forms on Stackoverflow. As we are only considering code snippets we are missing out on them. Ideally we would like to use NLP techniques to detect the insecure advices. Also some code snippets do contains insecure patterns but the developers has warned against blindly copy pasting this insecure code snippets as comments in the code snippets. As we are disregarding comments, it is quite debateable that considering the code snippets would make sense at any one before copy pasting this code snippet would read the comment and already know abut the insecurity of the code snippet.

6 DISCUSSIONS ON ML BASED DETECTION TECHNIQUES

Identification of insecure patterns, vulnerabilities, and code-smells using ML based techniques have been gaining tractions recently. This is mainly for two reasons i) leveraging the wealth of open source code available ii) adaptive neural network models which can capture/represent the complex patterns of source code. While givng a overview of ongoing research in this area is outside the scope of the paper, we can mention some work exicting research work closely related to ours that uses ML techniques. For example, Zhou et al. proposed [21] vulnerability detection model "Devign" for code snippets in C language. Their key idea to detect vulnerability is by capturing the abstract syntax tree structure of the source code via Gated-Graph Neural Network (GGNN). The study in [16] proposed automated vulnerability detection tools using deep feature representation learning.

One problem associated with using machine learning models is that either they are function level detection model [16], do have limited abilitity to reason why source code is vulnerable [21], or suffers from subjectivity [7].

Static analysis based approaches such as ours, can overcome for two reasons i) it can reason about the source code, and ii) as insecure patterns are repetitive. As a result we can build a static analysis tool by observing the common insecure trends to reason about the source. We agree this can be an overstreaed claim, and leave it as a future work this paper.

7 RELATED WORK

Our work is motived by the following related research work. Subramanian, et al. [18] used Eclipse Java Development Tools (JDT) [3] to find structural models of code snippets in Stack Overflow. Consequently, by analyzing *solved* Stack Overflow questions having *Android* tag they present a common list of Android API types and methods – something which normal lexical parsers are unable to detect. Fischer et al. [9] quantitatively evaluated the observation that a large number of insecure code snippets are being directly copy-pasted, repeatedly resued. They showed that a simple stochastic gradient descent based classifier can confirm that among 1.3 million Google Play Android applications, 15.4% contains security-related code snippets from from Stack Overflow – out of which 97.9% contain at least one insecre code snippet. Meng et al. [11] did an empirical study on the on StackOverflow posts, aiming to understand developers' concerns on Java secure coding. This study highlights a number of popular-accepted insecure suggestions on StackOverflow including sugeestions to disabling the default protection against Cross-Site Request Forgery (CSRF) attacks, breaking SSL/TLS security through bypassing certificate validation, and using insecure cryptographic hash functions. These harmful insecure suggestions can easily misguide developer – the extend of which is still unknown today. Interestingly, Rahman et al. [15] did a study similar to Meng et al. [11], but for code snippets for Python language. They observed that 9.8% of the 7,444 accepted answers to include at least one insecure code block. Most importantly they also find user reputation not translate to the presence of insecure code blocks, implying that both high and low-reputed users are likely to introduce insecure code blocks.

8 CONCLUSION

In this paper, we explore the problem of identifying insecure patterns on source code snippets given in response to Stack Overflow (SO) questions. We motivate the problem by emphasizing that insecurity in code snippets present in SO, can potentially trickled down to production level source code – introducing vulnerabilities. We believe that by flagging the common 8 insecure patterns compiled from existing literature can encourage developers to accept, updvote, share secure code snippets. We propose a static analysis tool based on keyword searching and backword flow analysis to do so. Our experimental result on 1.6K code snippets written in Java language demonstats the promising outcome of our proposed solution.

REFERENCES

[1] S. Baltes, C. Treude, and S. Diehl. 2019. SOTorrent: Studying the Origin, Evolution, and Usage of Stack Overflow Code Snippets. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 191–194. https://doi.org/10.1109/MSR.2019.00038

[2] Elaine Barker, William Burr, Alicia Jones, Timothy Polk, Scott Rose, Miles Smid, and Quynh Dang. 2009. Recommendation for key management part 3: Application-specific key management guidance. *NIST special publication* 800 (2009), 57.

[3] Cloudflar. 2015. What Is Domain Spoofing? | Website and Email Spoofing. https://www.cloudflare.com/learning/ssl/what-is-domain-spoofing/.

[4] Cryptomathic. 2015. Symmetric Encryption Algorithms - Their Strengths and Weaknesses, and the Need for Crypto-Agility. https://www.cryptomathic.com/news-events/blog/symmetric-encryption-algorithms-their-strengths-and-weaknesses-and-the-need-for-crypto-agility.

[5] Barthélémy Dagenais and Laurie Hendren. 2008. Enabling static analysis for partial java programs. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. 313–328.

[6] A. De Lucia. 2001. Program slicing: methods and applications. In *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*. 142–149. https://doi.org/10.1109/SCAM.2001.972675

[7] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. 2018. Detecting code smells using machine learning techniques: Are we there yet?. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 612–621. https://doi.org/10.1109/SANER.2018.8330266

[8] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 73–84.

[9] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 121–136.

[10] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (Nov. 2019), 56–65. https://doi.org/10.1145/3318162

[11] Na Meng, Stefan Nagy, Danfeng Yao, Wenjie Zhuang, and Gustavo Arango Argoty. 2018. Secure coding practices in java: Challenges and vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*. 372–383.

[12] Stack Overflow. 2015. java - When I try to convert a string with certificate, Exception is raised. https://stackoverflow.com/questions/10594000/when-i-try-to-convert-a-string-with-certificate-exception-is-raised.

[13] Stack Overflow Developer Survey 2020. 2020. - Most Loved, Dreaded, and Wanted Languages. https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved.

[14] 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* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2455–2472. https://doi.org/10.1145/3319535.3345659

[15] A. Rahman, E. Farhana, and N. Imtiaz. 2019. Snakes in Paradise?: Insecure Python-Related Coding Practices in Stack Overflow. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 200–204. https://doi.org/10.1109/MSR.2019.00040

[16] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 757–762. https://doi.org/10.1109/ICMLA.2018.00120

[17] Soot. 1999. - A framework for analyzing and transforming Java and Android applications. https://soot-oss.github.io/soot/.

[18] Siddharth Subramanian and Reid Holmes. 2013. Making sense of online code snippets. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 85–88.

[19] Raja Vallee-Rai and Laurie J Hendren. 1998. Jimple: Simplifying Java bytecode for analyses and transformations. (1998).

[20] H. Y. Yang, E. Tempero, and H. Melton. 2008. An Empirical Study into Use of Dependency Injection in Java. In *19th Australian Conference on Software Engineering (aswec 2008)*. 239–247. https://doi.org/10.1109/ASWEC.2008.4483212

[21] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS) 32*. Curran Associates, Inc., 10197–10207.

APPENDIX

9 EXAMPLES OF INSECURE PATTERNS IN CODE SNIPPETS

```
1  private byte[] encrypt(byte[] raw, byte[] clear) {
2    ...
3    Cipher cipher = Cipher.getInstance("AES");}
4    ....
5    return encrypted
6  }
```

[3] http://www.eclipse.org/jdt

```

```

Listing 3: A code snippet showing insecure use of AES default encryption mode ECB.

```
   public static void allowAllSSL() {
            HttpsURLConnection.setDefaultHostnameVerifier
     (new HostnameVerifier() {

                @Override
                public boolean verify(final String
     hostname, final SSLSession session) {
                    return true;
                }
            });
      ...
   }
```

Listing 4: A code snippet showing insecure use of Absence of performing hostname verification.

```

public byte[] RSAEncrypt(final String plain) throws ... {
    kpg = KeyPairGenerator.getInstance("RSA");
    kpg.initialize(1024);
    kp = kpg.genKeyPair();
    publicKey = kp.getPublic();
    privateKey = kp.getPrivate();

    cipher = Cipher.getInstance("RSA");
    ...
}
```

Listing 5: A code snippet showing insecure use of weak key length.

```
    ...
    byte[] salt = "abababababababababa bab".getBytes();
    byte[] iv = "1234567890abcdef".getBytes();
    ...
}
```

Listing 6: A code snippet showing insecure use of static predictable IV.

```
    ...
    public ServiceConnectionSE(String url) throws
     IOException {
      ...
      ((HttpsURLConnection) connection).
     setHostnameVerifier(new AllowAllHostnameVerifier());
      ...
    }
}
```

Listing 7: A code snippet showing insecure use of AllHostNameVerifier.

```
    ...
    @Override
    protected void configure(HttpSecurity hs) throws
     Exception {
      hs.csrf.disable()
    }
    ...
```

Listing 8: A code snippet showing insecure use of turning off CSRF protection.

10 FAILURE TO DETECT ABUSE OF X509TRUSTMANAGER VERIFIER INTERFACE INSECURE PATTERN

11 FUTURE WORK ON SYNTHESIZING SECURE CODE

12 LINKS TO SOURCE FILES

- source code: https://github.com/islamazhar/Detecting-Insecure-Implementation-code-snippets/tree/master/PPA
- Data set of all code snippet: https://github.com/islamazhar/Detecting-Insecure-Implementation-code-snippets/tree/master/PPA/src/examples

```
1  package examples.X509TrustManager;
2  import android.util.Log;
3  import javax.net.ssl.HostnameVerifier;
4  import javax.net.ssl.SSLSession;
5  import javax.net.ssl.TrustManager;
6  import java.security.KeyManagementException;
7  import java.security.NoSuchAlgorithmException;
8  import java.security.SecureRandom;
9  import java.security.cert.CertificateException;
10 import java.security.cert.X509Certificate;
11 public class class_1056 {
12
13     private static TrustManager[] trustManagers;
14
15     public static class _FakeX509TrustManager implements javax.net.ssl.X509TrustManager {
16         private static final X509Certificate[] _AcceptedIssuers = new X509Certificate[]{};
17
18         public void checkClientTrusted(X509Certificate[] arg0, String arg1)
19                 throws CertificateException {
20         }
21
22         public void checkServerTrusted(X509Certificate[] arg0, String arg1)
23                 throws CertificateException {
24         }
25
26         public X509Certificate[] getAcceptedIssuers() {
27             return (_AcceptedIssuers);
28         }
29     }
30
31     public static void allowAllSSL() {
32
33         javax.net.ssl.HttpsURLConnection.setDefaultHostnameVerifier(new HostnameVerifier() {
34
35             public boolean verify(String hostname, SSLSession session) {
36                 return true;
37             }
38         });
39
40         javax.net.ssl.SSLContext context;
41
42         if (trustManagers == null) {
43             trustManagers = new TrustManager[]{new _FakeX509TrustManager()};
44         }
45
46         try {
47             context = javax.net.ssl.SSLContext.getInstance("TLS");
48             context.init(null, trustManagers, new SecureRandom());
49             javax.net.ssl.HttpsURLConnection.setDefaultSSLSocketFactory(context.getSocketFactory());
50         } catch (NoSuchAlgorithmException e) {
51             Log.e("allowAllSSL", e.toString());
52         } catch (KeyManagementException e) {
53             Log.e("allowAllSSL", e.toString());
54         }
55     }
```

Figure 5: An example of insecure X509TrustManager implementation which we have not been able to detect. Corresponding Jimple representation is show in Figure 6

```
Warning: java.lang.ref.Finalizer is a phantom class!
Using PPA Mode: 3
Warning: javax.net.ssl.TrustManager is a phantom class!
Warning: examples.X509TrustManager.class_1056$1 is a phantom class!
Warning: javax.net.ssl.SSLContext is a phantom class!
Warning: javax.net.ssl.HttpsURLConnection is a phantom class!
Warning: MMAGICPPACKAGE.MagicClass is a phantom class!
Warning: javax.net.ssl.SSLSession is a phantom class!
Warning: javax.net.ssl.X509TrustManager is a phantom class!
Warning: javax.net.ssl.HostnameVerifier is a phantom class!
Warning: examples.X509TrustManager.class_1056$_FakeX509TrustManager is a phantom class!
Warning: android.util.Log is a phantom class!
Transforming examples.X509TrustManager.class_1056...
$r0 = new examples.X509TrustManager.class_1056$1
specialinvoke $r0.<examples.X509TrustManager.class_1056$1: void <init>()>()
staticinvoke <javax.net.ssl.HttpsURLConnection: MMAGICPPACKAGE.MagicClass setDefaultHostnameVerifier(javax.net.ssl.HostnameVerifier)>($r0)
$r2 = <examples.X509TrustManager.class_1056: javax.net.ssl.TrustManager[] trustManagers>
if $r2 != null goto context = staticinvoke <javax.net.ssl.SSLContext: javax.net.ssl.SSLContext getInstance(java.lang.String)>("TLS")
$r3 = newarray (javax.net.ssl.TrustManager)[1]
$r4 = newarray (javax.net.ssl.TrustManager)[1]
$r5 = new examples.X509TrustManager.class_1056$_FakeX509TrustManager
specialinvoke $r5.<examples.X509TrustManager.class_1056$_FakeX509TrustManager: void <init>()>()
$r4[0] = $r5
<examples.X509TrustManager.class_1056: javax.net.ssl.TrustManager[] trustManagers> = $r4
goto [?= context = staticinvoke <javax.net.ssl.SSLContext: javax.net.ssl.SSLContext getInstance(java.lang.String)>("TLS")]
context = staticinvoke <javax.net.ssl.SSLContext: javax.net.ssl.SSLContext getInstance(java.lang.String)>("TLS")
$r6 = <examples.X509TrustManager.class_1056: javax.net.ssl.TrustManager[] trustManagers>
$r7 = new java.security.SecureRandom
specialinvoke $r7.<java.security.SecureRandom: void <init>()>()
virtualinvoke context.<javax.net.ssl.SSLContext: MMAGICPPACKAGE.MagicClass init(null_type,javax.net.ssl.TrustManager[],java.security.SecureRandom)>(null, $r6, $r7)
$r9 = virtualinvoke context.<javax.net.ssl.SSLContext: MMAGICPPACKAGE.MagicClass getSocketFactory()>()
staticinvoke <javax.net.ssl.HttpsURLConnection: MMAGICPPACKAGE.MagicClass setDefaultSSLSocketFactory(MMAGICPPACKAGE.MagicClass)>($r9)
goto [?= return]
e := @caughtexception
$r11 = virtualinvoke e.<java.security.NoSuchAlgorithmException: java.lang.String toString()>()
staticinvoke <android.util.Log: MMAGICPPACKAGE.MagicClass e(java.lang.String,java.lang.String)>("allowAllSSL", $r11)
goto [?= return]
e := @caughtexception
$r13 = virtualinvoke e.<java.security.KeyManagementException: java.lang.String toString()>()
staticinvoke <android.util.Log: MMAGICPPACKAGE.MagicClass e(java.lang.String,java.lang.String)>("allowAllSSL", $r13)
goto [?= return]
return
this := @this: examples.X509TrustManager.class_1056
specialinvoke this.<java.lang.Object: void <init>()>()
return
```

Figure 6: Jimple represention of the code snippets in Figure 5. The interesting part is `_FakeX509TrustManager` is declared as phantom class.