Suggesting Secure Implementation to Vulnerable Code Snippets on Stackoverflow.

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.

1 INTRODUCTION

≪**Mazharul 1.0:** Add a ss from stackOverflow≫

In this project I want build a static analysis tool which will achieve the following.

- Analyze the code snippets from Stackoverflow for identifying out which part of the code is vulnerable by showing warning signs to highlight that part of the code to the developer.
- When developer clicks on the warning sign a secure implementation while be shown to the developers. In case of failure of building generating a secure implementation, the tool will show insightful/helpful messages explaing why this part of the code is flaged as insecure.

The problem is interesting for two reasons.

Difficulty of writing crypto code securely. Writing/implementing cypto code securely is a diffculut task for programmers. Any potential bug in crypto code can lead to serious vulnerablities open for attackers. Even so unlike other code, crypto code can be insecure even if it works perfectly on traditional test-suite's input/output which is used only to prove the implementation correctness of the program.

Online platforms roles in spreading insecure code. Online programming discussion platforms such as Stack Overflow have a rich source of ready to use code snippets for software developers. It is the defac-to place where developers go to find solutions of their problems and turn to the community for answers to their problems. Insecure code snippets found on Stackoverflow itself is not a serious problem. However, Fischer et al. has shown that developers have a tendency to directly copy paste code form Stack Overflow [3]. Therefore there are chances that any insecure code snippets posted on Stackoverflow can potentially find it way into production level code. To make matters worse, Meng et al. [4] has showed that many accepted answers on Stackoverflow have seriously insecure code and often-times given by users having high reputation. This adds to the problem copy pasting vulnerable code from online platform and furthermore increases the chances of the insecure code snippet being trickled down to production level code. Unfortunately there is not state-of-the-art tool to analyzie if a code posted by developer on Stackoverflow is secure or not. In absense of such tool, Stackoverflow is potentially contributing as a major source of vulnerability in production level code.

A static tool which can identify which part of the code snippet is insecure and suggest secure alternatives can help stopping the flow of insecure code from Stackoverflow to production level code.

≪**Mazharul 1.0:** Talk about key challenges here. Say there are existing tools that can detect these rules on complete source codes. But code snippets presents some unique challenges. such has

- code snippets are erronous
- code snippets are incomplete

≫

2 THREAT MODEL

Rule No	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

We will now summerize the insecure patterns our method aims to detect in the following 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 [FIXME: Add some % number]. In Java an instance of AES can be created using `javax.crypto.Cipher`. 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 [FIXME: See the code in the Appendix A]. While ECB-encrypted ciphertext allows random access to each block, it can also leak information via side channel attacks [FIXME: Cite source]. 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 as shown in appendix.

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 message. This unique message digest can be used latter for verifing 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 in code snippets developers have been using these popular broken hashes as shown in listing **??**.

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 **??**. 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).

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,[14] an update to the widely-accepted recommendation of a 1024-bit minimum since at least 2002. 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.

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 **??**.

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 accept all certificates. This is same as using empty methods as discussed in **??**. This time developers can just use ALLOW_ALL_HOSTNAME_VERIFIER static field to do this. 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.

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.

3 METHODOLOGY

In this section, we will discuss the pipeline we follow to detect insecure patterns in code snippets as shown in Figure 1. 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.[FIXME: Give a statistic on the avg. length of the code snippets of the dataset] 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

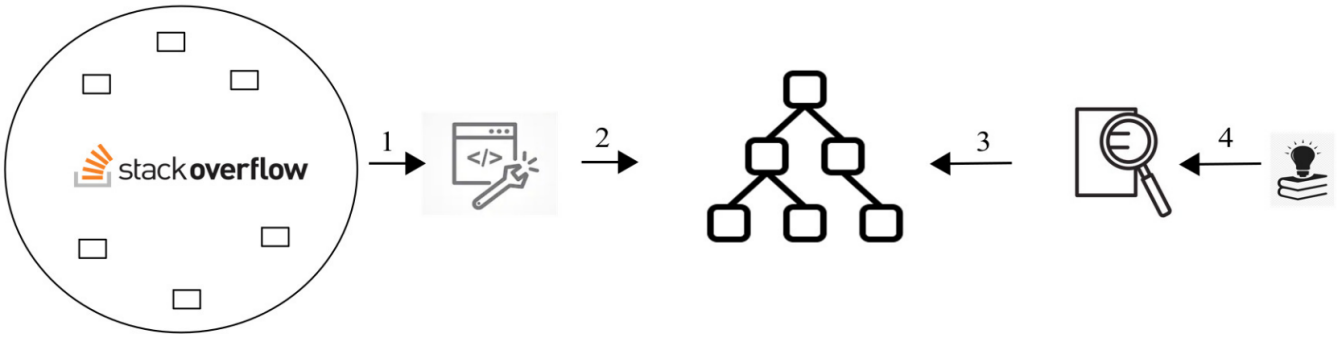

Figure 1

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., &gt, &lt, &amp, 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 [9] 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 [1] 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 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 [7]. 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 8	`SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER` `*.csrf.disable()`

Table 2

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

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. [6] 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.

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. [5] in their CryptoGurd Project. They introduced specialize def-use analysis [10] based on program slicing techniques [2] 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 **??** for rule 2 – detecting insecure broken cryptographic hashes MD5, MD4, MD2, SHA1. We can use keyword searching based on the name of broken hashes, and successsfully detect. However it will introduce FP for the code snippets shown in listing **??**. 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.getInstanc` and has a value of equal to name of the broken hash.

≪**Mazharul 3.0:** A formal proof is given in the Appendix.≫

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.

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.

4.1 Data collection

We used dataset from two previous sources [3, 4]. Both of these dataset contain code snippets posted on Stackoverflow. Fisher et al. crawled 1,161 code snippets posted on Stackoverflow related to Andrioid Securuity [3]. 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 [4]. 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.

≪**Mazharul 4.0:** add some more info and some statistics≫

To make the analysis more clear, we try to categorize the code snippets into one of the 8 rules. This is achieved by manually inspecting randomly sampled 200 code snippets from 1.6K available code snippets. We assign the code snippets into one of the 8 insecure patterns, and record the common keywords appear in the code snippets. By doing this we have list of common keyword for each of the 8 insecure patterns. We can then categorize all 1.6K code snippets using these common keywords.

4.2 Code repair

We called a code snippet successfully repaired if we can convert it to a Jimple IR. Figure 2 illustrates the percentage of code snippets for

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

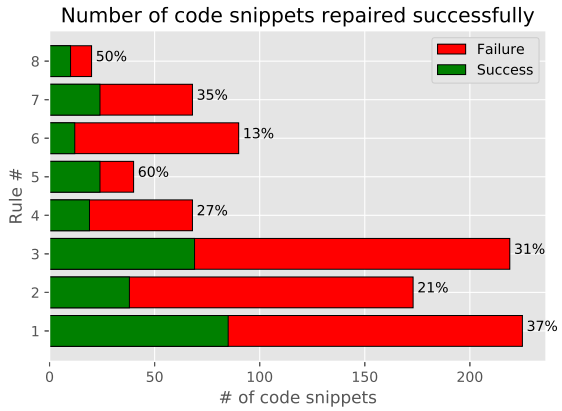

Figure 2: The percentage of code snippets successfully parsed for each insecure pattern rules.

each categorize, we have been able to parsed (i.e., convert to Jimple IR), using the code repair techniques discussed in subsection 3.1.

4.3 Insecure pattern detection accuracy.

After converting each successfully repaired code snippets to Jimple IR, we can give the Jimple IR to our tool. Out tool is built on top of CryptoGurd which uses Soot as its program analysis enginee. Using Soot's `name of the API` API, we enable backword flow analysis given the slicing criteria. The idea is track the special method invokations parameter and.

5 LIMITATIONS

- We did not analyze other code snippets other than Stack Overflow
- Program repairs are basically simple parsing.
- Limitations due to PPA/Jimple grammer. Could have used Eclispe JDT
- The categoriezation is based on keyword searching.

6 DISCUSSIONS

6.1 ML based detection techniques

6.2 Code Repair

6.3 Limitations for failing to construct a CFG:

program repair are basically simple edits PPA is very old

The tension between soundness and completeness: Our tool is the analysis part but not complete. but keyword based analysis also should introduce no FP. Synthesis Other Sources of vulnerability: Lack of training /Knowledge: Vulnerable Online Turorials Lack of screening tools: Being short and focusing on a working solution while writing code snippets.

7 RELATED WORK

Code snippets on StackOverflow. Subramanian, et al. [8] used Eclipse Java Development Tools (JDT) [2] 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. [3] 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. [4] 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. [6] did a study similar to Meng et al. [4], 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.

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

8 FUTURE WORK AND CONCLUSIONS

- Suggestion part

REFERENCES

[1] 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.
[2] 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
[3] 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.
[4] 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.
[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* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2455–2472. https://doi.org/10.1145/3319535.3345659
[6] 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
[7] Soot. 1999. - A framework for analyzing and transforming Java and Android applications. https://soot-oss.github.io/soot/.
[8] 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.
[9] Raja Vallee-Rai and Laurie J Hendren. 1998. Jimple: Simplifying Java bytecode for analyses and transformations. (1998).
[10] 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

9 APPENDIX

≪**Mazharul 9.0:** This is working..≫

```
    private byte[] encrypt(byte[] raw, byte[] clear) {
      ...
      Cipher cipher = Cipher.getInstance("AES");}
      // Cipher cipher = Cipher.getInstance("AES/CBC/
     PKCS5Padding");
      ....
      return encrypted
    }

```

Listing 1: A real code snippet taken from Stackoverflow. I want to build a tool which after analyzing the code snippet will highlight the part of the code that is insecure and suggest an alternative secure implementation as showed in the figure.