2017 IEEE Symposium on Security and Privacy

Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security

Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky*, Yasemin Acar*, Michael Backes*, Sascha Fahl*
Fraunhofer Institute for Applied and Integrated Security; *CISPA, Saarland University

***Abstract*—Online programming discussion platforms such as Stack Overflow serve as a rich source of information for software developers. Available information include vibrant discussions and oftentimes ready-to-use code snippets. Previous research identified Stack Overflow as one of the most important information sources developers rely on. Anecdotes report that software developers copy and paste code snippets from those information sources for convenience reasons. Such behavior results in a constant flow of community-provided code snippets into production software. To date, the impact of this behaviour on code security is unknown.**

We answer this highly important question by quantifying the proliferation of security-related code snippets from Stack Overflow in Android applications available on Google Play. Access to the rich source of information available on Stack Overflow including ready-to-use code snippets provides huge benefits for software developers. However, when it comes to code security there are some caveats to bear in mind: Due to the complex nature of code security, it is very difficult to provide ready-to-use and secure solutions for every problem. Hence, integrating a security-related code snippet from Stack Overflow into production software requires caution and expertise. Unsurprisingly, we observed insecure code snippets being copied into Android applications millions of users install from Google Play every day.

To quantitatively evaluate the extent of this observation, we scanned Stack Overflow for code snippets and evaluated their security score using a stochastic gradient descent classifier. In order to identify code reuse in Android applications, we applied state-of-the-art static analysis. Our results are alarming: 15.4% of the 1.3 million Android applications we analyzed, contained security-related code snippets from Stack Overflow. Out of these 97.9% contain at least one insecure code snippet.

I. Introduction

Discussion platforms for software developers have grown in popularity. Especially inexperienced programmers treasure the direct help from the community providing easy guide and most often even ready-to-use code snippets. It is widely believed that copying such code snippets into production software is generally practiced not only by the novice but by large parts of the developer community. Access to the rich source of information given by public discussion platforms provides quick solutions. This allows fast prototyping and an efficient workflow. Further, the public discussions by sometimes experienced developers potentially promote distribution of best-practices and may improve code quality on a large basis.

However, when it comes to code security, we often observe the opposite. Android-related discussions on Stack Overflow for example include an impressive conglomeration of oddities: from requesting too many and unneeded permissions [1] to implementing insecure X.509 certificate validation [2] to misusing Android's cryptographic API [3], a developer who is seeking help can find solutions for almost any problem. While such solutions oftentimes provide functional code snippets, many of them threaten code security. Those insecure code snippets commonly have a rather solid life-cycle: provided by the community, copied and pasted by the developer, shipped to the customer, and exploited by the attacker. To date it is unknown to what extent software developers copy and paste code snippets from information sources into production software. Is this phenomenon limited to just occasional instances, or is it rather a general and dangerous trend threatening code security to a large extent?

We answer this highly important question by measuring the frequency 1,161 insecure code snippets posted on Stack Overflow were copied and pasted into 1,305,820 Android applications available on Google Play. We demonstrate that the proliferation of insecure code snippets within the Android ecosystem, and thus the impact of insecure code snippets posted on Stack Overflow, poses a major and dangerous problem for Android application security.

Our Contributions

We investigate the extent security-related code snippets posted on Stack Overflow were copied into Android applications available on Google Play. Our contributions can be summarized as follows:

- We identified all Android posts on Stack Overflow, extracted all (4,019) security-related code snippets and analyzed their security using a robust machine learning approach. As a result we provide a security analysis for all security-related Android code snippets available on Stack Overflow.
- We applied state-of-the-art static code analysis techniques to detect extracted code snippets from Stack Overflow in 1.3 million Android applications.
- We found that 15.4% of all 1.3 million Android applications contained security-related code snippets from Stack Overflow. Out of these 97.9% contain at least one insecure code snippet.
- We designed and implemented a fully automated large-scale processing pipeline for measuring the flow of security-related code snippets from Stack Overflow into Android applications.
- We make all data available on https://www.aisec.fraunhofer.de/stackoverflow.

DOI 10.1109/SP.2017.31

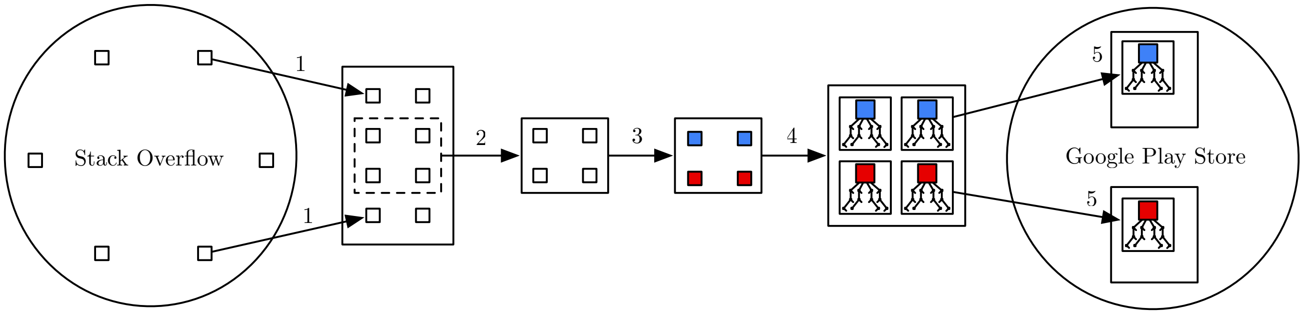

Fig. 1: Overall processing pipeline of code extraction (1), filtering (2), classification (3), program dependency graph generation (4), and clone detection (5).

Our processing pipeline is fully automated and designed to scale to extensive measurements of platforms other than Stack Overflow and software repositories other than Google Play.

II. Processing Pipeline Architecture

In this section, we discuss the architecture of our processing pipeline. The individual steps of the processing pipeline are described in detail in subsequent sections.

As depicted in Figure 1 the code originates in the Stack Overflow database (on the left) and flows into Google Play (on the right). To measure this flow we first crawl Stack Overflow and extract every single code snippet in the database (1). From this comprehensive snippet collection we filter those that are security-related (2). We discuss steps (1) and (2) in detail in Section III on code extraction and filtering. This provides us with a set of security-related snippets. In order to label each of them *secure* or *insecure* we define labeling rules as described in Section IV and apply machine learning classification (3) using support vector machines (cf. Section V). Next, we generate an abstract representation of each labeled code snippet (4) that allows us to detect their clones in Google Play (5) (cf. Section VI). Each step is fully automated and designed for large scale analysis. Only the training step for supervised machine learning classification (3) requires manual labeling of training data. However, this must be done only once for a small fraction of snippets, classification of very large sets of code snippets afterwards runs fully automated and is therefore just a matter of processing power and time. As we will show in the evaluation in Section VII-C our proposed approach is time-efficient and yields decent results.

III. Code Extraction and Filtering

First, we crawl discussion threads from a developer discussion platform for actual code snippets. Second, we extract all security-related snippets. We begin this section by defining the criteria for security-related code snippets and continue with describing our implementation for Stack Overflow.

A. Security-related Code Snippets

On Android, security operations include but are not limited to cryptographic operations, secure network communication and transmission, validation via PKI-based mechanisms, as well as authentication and access control. These operations are supported by different APIs. We define code elements of these APIs as an indicator for security-related code: A code snippet is considered security-related iff it makes calls to one of the following APIs: [4]

- Cryptography: Java Cryptography Architecture (JCA), Java Cryptography Extension (JCE)
- Secure network communications: Java Secure Socket Extension (JSSE), Java Generic Security Service (JGSS), Simple Authentication and Security Layer (SASL)
- Public key infrastructure: X.509 and Certificate Revocation Lists (CRL) in java.security.cert, Java certification path API, PKCS#11, OCSP
- Authentication and access control: Java Authentication and Authorization Service (JAAS)

Additionally, we included code snippets with reference to the following security libraries, which were specially designed for Android: BouncyCastle (BC) is the default, pre-installed cryptographic service provider on Android and is widely used [3]. SpongyCastle[1] (SC) gives a repackaged version of BC which provides additional functionality. We looked for code snippets containing both BC and SC API calls.

Furthermore, we extracted code snippets for the Apache TLS/SSL package as part of the `HttpClient` library which is one of the most used libraries on GitHub [5].

We also included code snippets that reference security libraries specifically designed with usability in mind [6], e.g. keyczar [7] and jasypt [8], which were designed to simplify the safe use of cryptography for developers.

To contrast Android's default providers and the usable security libraries with a more inconvenient alternative, we included GNU Crypto. Although this library also implements a JCA provider, it is challenging to integrate into Android [9], which makes it interesting to see whether it is being discussed on Stack Overflow and used by developers.

Table I lists the considered security libraries and gives an overview of their supported features.

[1] cf. https://rtyley.github.io/spongycastle/

	TLS	Symmetric Cryptography	Asymmetric Cryptography	Secure Random Number Generation	Message Digests	Digital Signatures	Authentication	Usability by Design
Standard API	●	●	●	●	●	●	●	○
BouncyCastle	●	●	●	●	●	●	●	○
SpongyCastle	●	●	●	●	●	●	●	○
Apache TLS/SSL	●	○	○	○	○	○	○	○
keyczar	○	●	●	○	○	●	●	●
jasypt	○	●	○	○	●	○	○	●
GNU Crypto	○	●	●	●	●	●	○	○

● = fully applies;
○ = does not apply at all

TABLE I: Cryptographic libraries and their supported features.

B. Finding Security-related Code Snippets on Stack Overflow

Code snippets on Stack Overflow are surrounded by `<code>` tags and can therefore easily be separated from accompanying text and extracted.

In order to decide which API is used by a code snippet, we need Fully Qualified Names (FQN) (i.e. package names in Java) of code elements in the snippet. Since Partially Qualified Names (PQN) (i.e. class and method names) are not unique, different APIs can contain classes (e.g. `android.util.Base64`, `java.util.Base64`) and methods (e.g. `java.security.Cipher.getInstance`, `java.security.Signature.getInstance`) which share the same name. FQNs allow us to distinguish non-unique class and method names.

Code snippets posted on Stack Overflow are often incomplete or erroneous and therefore only PQNs are available. Since disambiguating partial Java programs is an undecidable problem [10], we used an oracle called JavaBaker [11] to decide to which API a code element belongs. The oracle consists of a user-defined set of APIs which is used to apply a constraint-based approach to disambiguate types of given code elements. Given a code snippet JavaBaker returns the FQN for each element in the code, if it belongs to one of the initially given libraries. The JavaBaker oracle has a precision of 0.97 and a recall of 0.83 [11]. It is not restricted to specific libraries. With JavaBaker, using the security libraries explained in Section III-A, we were able to determine to which of the given security APIs a type reference, method call, or field access in a code snippet belongs. A code snippet is therefore considered security-related if the returned result of the oracle is not empty. We apply this to separate security-related code snippets from Stack Overflow from snippets that are not security-related.

Since the security APIs might contain packages whose usage does not indicate implementation of security code (e. g. *util* or *math* packages), our snippet filter includes a blacklist to ignore those non-security-related packages. We compiled this blacklist manually by inspecting each package individually.

Code snippets may contain sparsely used code elements. For instance, an object can be declared and initialized, but not used subsequently in the snippet. In this case, the oracle only has the PQN of the element and the call to the constructor as information to decide the FQN. This can lead to false positives because the oracle has insufficient information to narrow down possible candidates. To give an example, the oracle reported `java.security.auth.login.Configuration` as the FQN for a code element with type `Configuration` whose true FQN was `android.content.res.Configuration`. The related object only made a call to the constructor, hence it was impossible to disambiguate the given type `Configuration`. Luckily, these false positives are easily detectable by filtering out snippets for which the oracle reports the `<init>` method only or no methods at all. We do not worry about true positives we might sort out this way, as we are not interested in code snippets that contain security elements which are not used after initialization.

C. Limitations

The main purpose of the the oracle-based filter is to decide whether a given snippet is security-related. As it does this by examining the snippet for utilization of the defined security libraries, it might label a snippet as security-related, even though it does not belong to a security context. This is the case if an API element which is heavily used for security purposes can also be used in a non-security context. For instance, in a security context snippets would use hashing algorithms for verifying data integrity. In a non-security context hashes may be used for data management purposes only. In both cases the snippet would reference elements of one of the given security APIs which causes the filter to label the snippets as security-related.

IV. CODE LABELING

Now that we have extracted security-related code snippets (cf. Figure 1, (1) and (2)), we need to classify them as such. Therefore, we first provide the label definition and labeling rules and give details on the actual machine learning based classification in Section V. We apply supervised learning and therefore need to manually label a small fraction of extracted code snippets to train the support vector machine. Therefore, a pair of two reviewers inspected the set of 1,360 security-related snippets extracted from answer posts from Stack Overflow. We assume that answer snippets are more likely to be copied and pasted as they are intended to solve a given problem. Question snippets are not included in the training set as they might introduce unpredictable noise, which would compromise the classifier.

In case of conflicts, a third reviewer was consulted and the conflict was resolved (by explaining the reasoning of the reviews).

To better understand which topics were discussed (in combination with code snippets) on Stack Overflow, we categorized each code snippet into one or multiple of the following categories: *SSL/TLS*, *Symmetric cryptography*, *Asymmetric cryptography*, *One way hash functions*, *(Secure) Random number generation*.

A. Security Labels

We checked whether or not code snippets were security risks when pasted into Android application code and labeled them either secure or insecure:

Secure

- Snippets that contain up-to-date and strong algorithms for symmetric cryptography [12], [13], sufficiently large keys for RSA or elliptic curve cryptography [14], [15] or secure random number generation [3].
- Snippets that contain code that does not adhere to security best practices, but does not result in easily exploitable vulnerabilities either, e.g. usage of RSA with no or PKCS1 padding [16], SHA1 or outdated versions of SSL/TLS [12].
- Snippets that contain code whose security depended on additional developer input, e.g. the symmetric cryptography algorithm or key size is a parameter, which is configurable by the developer.

Insecure

- Snippets that contained obviously insecure code, e.g. using outdated algorithms or static initialization vectors and keys for symmetric cryptography, weak RSA keys for asymmetric cryptography, insecure random number generation [3], or insecure SSL/TLS implementations [2].

This labeling is very conservative as it classifies only the definitely vulnerable code snippets as insecure.

B. Labeling Rules

Code security was investigated for the category specific parameters, which are introduced in this section. Based on these parameters we state a security metric which provides the rules for labeling the code snippets. Our stated security metric does not intend to be an exhaustive metric for each security category, but only considers security parameters which were actually used in the snippets of our corpus. In the following, we provide tables for each category which depict secure and insecure parameters for quick lookup. Additionally, we give details on parameters that were ambiguous or need further explanation. We defined the following labeling rules for security classification:

1) SSL/TLS: Table II illustrates the TLS parameters we investigated [2]. The `HostnameVerifier` checks whether a given certificate's common name matches the server's hostname. `TrustManager` implementations allow developers to implement custom certificate (chain) validation strategies. Insecure hostname verifier or trust manager implementations make an application vulnerable to Man-In-The-Middle attacks. According to [2] we labeled `TrustManager` and `HostnameVerifier` implementing insecure validation strategies as insecure. `TrustManagers` that implement public key or certificate pinning are considered secure. However, we label pinning as insecure if the pinset contains ambiguous values, e. g. serial number of the certificate [18], [17]. We also investigated TLS security of `WebViews`. Developers can implement their own `OnReceivedSSLError` method to handle certificate validation errors while loading content via HTTPS and can ignore validation errors by proceeding the TLS handshake.

Parameter	Secure	Insecure
Hostname Verifier	browser compatible, strict	allow all hosts [17]
Trust Manager	default, secure pinning	trust all [2], bad pinning [18], [17], validity only
Version	>=TLSv1.1 [12]	<TLSv1.1 [19], [12], [20], [21]
Cipher Suite	DHE_RSA, ECDHE AES>=128, GCM SHA>=256 [12]	RC4,3DES, AES-CBC MD5, MD2 [12], [22]
OnReceived-SSLError	cancel	proceed

TABLE II: Secure and insecure TLS parameters.

Parameter	Secure	Insecure
Cipher/Mode	AES/GCM [12] AES/CFB [12] AES/CBC*	RC2 [23], RC4 [24], DES [23], 3DES [25], AES/ECB [3], AES/CBC** [22] Blowfish [26], [27]
Key	provider generated	static [3], bad derivation [3]
Initialization Vector (IV)	provider generated	zeroed [3], static [3], bad derivation [3],
Password Based Encryption (PBE)	>=1k iterations [13], >=64-bit salt [13], non-static salt [13]	<1k iterations [13], <64-bit salt [13] static salt [3]

TABLE III: Secure and insecure symmetric cryptography parameters.

2) Symmetric Cryptography: We investigated snippets for symmetric cryptography parameters as illustrated in Table III. We labeled `Ciphers` and `Modes` of operation which are known security best practices as secure. Ciphers and modes with known practical attacks were labeled insecure. The AES encryption mode CBC is depicted in both columns secure and insecure because known padding oracle attacks are only feasible in a client/server environment. If this encryption mode is used in a different scenario, we consider it as secure [22]. We labeled cryptographic `Keys` and `IV` which were statically assigned, zeroed or directly derived from text (such as passwords) as insecure [3].

Parameter	Secure	Insecure
Cipher/Mode	RSA RSA/ECB RSA/None	
Padding	PKCS1*, PKCS8, OAEPWithSHA-256 AndMGF1Padding,	PKCS1**
Key	RSA >= 2048 bit ECC >= 224 bit	RSA < 2048 bit [14] ECC < 224 bit [15]

TABLE IV: Secure and insecure asymmetric cryptography parameters.

3) Asymmetric Cryptography: We investigated snippets for asymmetric cryptography parameters as illustrated in table IV. The JCE API provides different `Cipher` and `Mode` transformation strings for RSA which include the definition of a block mode, e. g. RSA/ECB. However, these modes are ignored by the underlying provider and have no implication on security [28]. For RSA, we consider the used `Padding` and `Key` length to evaluate security [14]. We distinguish between a client/server and a non-client/server scenario. Only in the first scenario PKCS1 padding is vulnerable to padding oracle attacks and seen as a secure padding otherwise [16]. Secure and insecure key lengths for RSA and Eliptic curve cryptography [14], [15] are shown in table IV.

Parameter	Secure	Insecure
PBKDF	[PBKDF2](Hmac) >=SHA224 [29]	[PBKDF2](Hmac) MD2, MD5 [29]
Digital Signature	>SHA1	MD2, MD5
Credentials	>SHA1	MD2, MD5

TABLE V: Secure and insecure hash function parameters

4) One Way Hash Functions: We investigated snippets for one way hash function parameters, as illustrated in Table V, in the context of password-based key derivation, digital signatures, and authentication/authorization. These were the only categories where code snippets from our analysis corpus made explicit use of hash functions. In the context of OAuth and SASL (authentication and authorization), attacks are mainly possible through flaws in website implementations [30]. Therefore, we only analyzed which hashing schemes were used for hashing credentials.

Parameter	Secure	Insecure
Type	SecureRandom	Random
Seeding	nextBytes, nextBytes->setSeed	setSeed->nextBytes, setSeed with static values [3]

TABLE VI: Secure and insecure parameters for (secure) random number generation.

5) (Secure) Random Number Generation: We investigated snippets for (secure) random number generation parameters shown in table VI. The main problem which can lead to security problems lies in provider specific implementation and ambiguous documentation of manual seeds [31]. We conclude that besides calling `nextBytes` only, which lets `SecureRandom` seed itself, calling `nextBytes` followed by `setSeed` is a secure sequence because `SecureRandom` is still self-seeded. The latter call to `setSeed` just supplements the seed and does not replace it [31]. Without calling `nextBytes` first, a call to `setSeed` may completely replace the seed. This behavior differs between several providers and is often ill-described in official documentation [31]. Therefore, we consider this call sequence as insecure if an insufficient seed is given.

C. Limitations

Our code snippet reviews might be limited in multiple ways in this step. Although we based our review decisions on widely accepted best practices and previous research results and let multiple reviewers review all snippets we cannot entirely eliminate incorrect labeling. The security of most code snippets depends on input values (e. g. initialization parameters) that were not given in all code snippets. Therefore, our results might under- or overreport the prevalence of insecure APIs in Android applications.

V. Code Classification

In this section, we present our method for large-scale code snippet classification, which corresponds to (3) in the overall processing pipeline (cf. Figure 1).

Manual snippet analysis allows profound insight into security problems specifically raised from crowd-sourced code snippets. Further, it allows the creation of a rich data set that annotates crowd-sourced code snippets from Stack Overflow. This opens the doors for machine learning based classification. To the best of our knowledge, we are the first to contribute such a data set to the machine learning community.

The security scoring of code snippets can be seen as a classification problem, which we can effectively solve by a variety of classifiers, e. g. feed-forward neural networks, decision trees, support vector machines, and many more. By manually labeling a subset of the collected snippets as secure and insecure (cf. Section IV), we are able to produce a training data set for binary classifiers. The trained model is then applied to classify unknown code snippets. We apply the binary classifier on all security-related snippets extracted by the oracle-based filter to provide an automatic procedure of security assessment.

It is arguable that machine learning based methods deliver more benefits than rule-based methods on solving security problems. Our binary classifier can efficiently extract discriminative information from the data set, which might be overlooked by rule based methods.

A. Support Vector Machine

We employ the binary classifier Support Vector Machine (SVM) as our learning model. In our scenario, the labeled training data set contains two classes, namely, insecure and secure code snippets. The collected code snippets can be

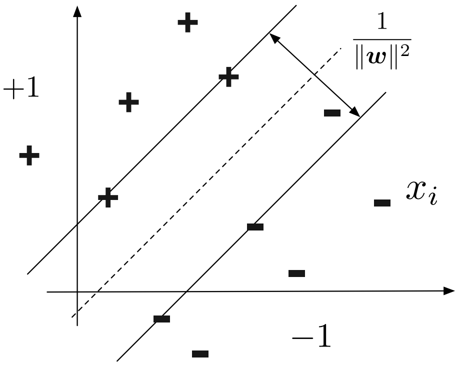

Fig. 2: Illustration of SVM binary classifier. It maximizes a margin $\frac{1}{\|\mathbf{w}\|^2}$ to separate positive and negative samples in its correct side. Note that a small portion of data samples are allowed within the margin, which can be controlled by a set of slack variables ξ.

regarded as documents. We argue that discriminative patterns can be discovered by examining the tokens in code snippets. These can be any combination of alphabets and symbols, e.g., *while*, *return*. Therefore, in our setting the learning problem is a document classification problem with binary classes from a set of tokens.

Given a training dataset of n samples $\mathcal{X} = \{x_i\}_{i=1}^n$, and its corresponding labels $\{y_i\}_{i=1}^n \in [+1, -1]$, a SVM classifier learns a margin that maximally separates training samples into two classes as illustrated in Figure 2. The objective function can be formulated as follows,

$$\begin{aligned} &\min_{w,b,\xi} \tfrac{1}{2} w^T w + C \textstyle\sum_{i=1}^n \xi_i \\ \text{s.t.} \quad &y_i(w^T \phi(x_i) + b) \geq 1 - \xi_i \\ &\xi_i \geq 0, i = 1, \ldots, n \end{aligned} \tag{1}$$

In (1) we note that minimizing w equals maximizing a margin. SVM introduces a set of slack variables $\{\xi_i\}$ to soften the margin, such that a small portion of training samples are allowed to be misclassified. Importantly, we also note that the feature mapping $\phi(x_i)$ defined over $\mathcal{X}$ can intrinsically handle non-linear cases by the so called 'kernel trick'. For more details, we refer to [32].

B. Feature Extraction

Since the learning problem of detecting the security level of code snippets can be viewed as a document classification problem, we employ a common feature extraction method named *tf-idf* vectorizer [33]. The *tf-idf* vectorizer transforms the whole set of code snippets into a numeric matrix. Each of the code snippets is considered as a document, namely an input data sample. We compute term frequency (*tf*) and inverse document frequency (*idf*) with respect to the total number of snippets.

For each snippet, the term frequency is computed by counting each token within its document. For the inverse document frequency, we compute the inverse of the number of documents where each token appears in. Then the *tf-idf* score is simply a multiplication of term frequency and inverse document frequency. In the end, we maintain a vocabulary of code tokens parsed from the snippets. This vocabulary will be converted into a numeric vector of a fixed length containing all possible tokens' frequency in this snippet. Normally, *tf-idf* vectorizer will form a high dimensional sparse data set with many entries being set to zero, if all the individual tokens are taken into account. Some tokens, e.g., randomly generated numbers, variable and class names, only appear in particular documents and therefore their document frequency is quite low. Document frequency can be very high for other tokens, e.g., common language terms such as *return*, *public*. The *tf-idf* scores for these tokens will be automatically justified by the inverse document frequency, such that their contribution to the discriminative function will also be reweighed. Finally, the sparse data set is then fed to SVM as the training data set. We expect the tokens found in each snippet to represent an encoding of how secure the code snippet will be.

VI. PDG Generation and Code Detection

Our processing pipeline has now filtered security-related code snippets from Stack Overflow and classified them either as secure or insecure (cf. Figure 1, (1) to (3)). Next, we aim to detect these code snippets in compiled Android applications from Google Play, (cf. Figure 1, (4) and (5)).

Snippets are given as source code and Android applications are only available as high-level binaries (i. e. DEX files). To be able to apply static code analysis, code snippets and Android applications must be transformed into the same (intermediate) representation (IR). In this section, we first describe this transformation step (4) and then give a detailed explanation of the method we apply (5).

A. Code Snippet Compiling

Commonly, static code analysis techniques require complete programs or source code [10]. Most code snippets from Stack Overflow however are not complete programs. They mostly do not compile without error since required method or class information is missing [11]. A snippet may be a subset of a larger program which is not accessible or additional dependencies (e. g. external libraries) might me unknown [10].

For incomplete code snippets creating a typed and complete IR is difficult. To overcome this, we use Partial Program Analysis (PPA) [10]. It was specifically designed to create complete and typed abstract syntax trees (AST) from source code of partial Java programs. PPA is able to resolve syntactic ambiguities. For example, the statement `SSLSocketFactory.getDefault()` does not allow to decide if `SSLSocketFactory` is a class or field name. In this case, `SSLSocketFactory` is a missing class and therefore `getDefault()` should be resolved to a static method call. PPA is also able to disambiguate possible typing problems which arise in case not all declared types are available. This is done by reconstructing data types from snippets without having access to source files, binaries or libraries. For data types that cannot be resolved applying PPA, a generic data type `UNKNOWNP.UNKNOWN` is used. This ensures that the created AST remains complete.

To transform snippets and applications into the same IR, we use WALA[2]. Since WALA operates on JVM bytecode, we transform Android applications to JVM bytecode using enjarify [34]. To be able to operate on Stack Overflow code snippets, we modified WALA by integrating PPA. This allows us to transform incomplete code snippets into WALA's IR. Before transformation, we make sure code snippets represent a complete Java class by adding missing class and method headers. Based on these snippets, we create the complete and typed AST using PPA. We were able to successfully process 1,293 answer (85.2%) and 1,668 question snippets (66.6%). Snippets which could not be compiled mostly had a too erroneous syntax and were therefore rejected by WALA. Furthermore, a lot of snippets contained a mixture of Java code and non-commented text (e. g code blocks were replaced with '(...)'). We ignored those snippets for further analyses [11].

B. Code Snippets in Apps

Code snippet containment is given if an application contains code that is very similar to the code snippet. However, a full match is not necessary. Instead we use a detection algorithm which is robust to *fractional* and *non-malicious* modifications[3].

We base code snippet detection on finding similar Program Dependency Graphs (PDG) which store data dependencies by applying a modified approach of Crussel et al. [35]. They create PDGs for each method and define the independent subgraphs of a PDG as the basic code features that are considered for reuse detection. A method's PDG may contain several data independent subgraphs which are called semantic blocks. Code similarity is defined on the amount of similar semantic blocks that are shared among the compared code. Following this approach provides robustness to high-level modifications and trivial control-flow alterations, as well as non-malicious code insertions/deletion, code reordering, constants modifications and method restructurings as described in [35].

Although, this approach allows the detection of reused code that has been subject to the defined modifications, we consider some of the given robustness features as inappropriate for our use case. It has several drawbacks when applied on detecting reuse of code snippets in large Android applications. Many snippets are quite small in terms of lines of code and therefore result in small PDGs. In this scenario, different code might result in identical PDGs. Therefore, we apply a more strict approach which additionally compares constants and method names that belong to a semantic block. This is reasonable because constants are critical for initializing Android security APIs. For instance cryptographic ciphers or TLS sockets are selected by using a transformation String (e. g. AES, TLS). Critical information like cryptographic keys, key lengths, initialization vectors, passwords and salts can be statically assigned in the code.

To be able to detect reused constants they must not have been modified. Additionally, we compare method names that are part of a semantic block and belong to APIs of our predefined set of security libraries. This allows us to distinguish security-related parts of the code, in case of different code with identical semantic blocks and empty or identical constant sets. Finally, we disallow class and method restructuring. This is necessary because we have have to ensure that detected semantic blocks are contained in classes and methods that have the same structure as the snippet. We compare semantic blocks, constants and method names on a per method base and ensure (nested) class membership by analyzing path names of all detected methods.

To avoid computational overhead, we limit the number of classes to search for code snippets to classes that contain security-related API calls as defined in Section III-B.

Finding subgraph isomorphisms in PDGs is NP-hard [36]. Therefore, we follow the approach of embedding graphs in vector spaces in order to reduce the problem of finding similar graphs to the problem of finding similar vectors [37]. We apply the embedding algorithm provided by Crussel et al. [36] which assigns a semantic vector to each semantic block. The semantic vector stores information about nodes and edges, i. e. the overall structure of a semantic block. Nodes represent instructions, edges represent data dependencies between instructions.

Each instruction type as provided by WALA's IR has two corresponding fields in the vector. One field stores node and the other stores edge information. The count of nodes for each instruction type (e. g. *invokevirtual*, *getfield*, *new* or *return*) in the semantic block is stored in the related nodes field of the instruction type in the vector. The maximum out node degree for each instruction type is used to store information about PDG edges. It holds the maximum count of outgoing edges over all nodes in a semantic block for a given instruction type and is stored in the related edges field of an instruction type in the vector.

To decide if two semantic vectors are similar, we calculate their Jaccard similarity [38], [39] which describes the similarity ratio of two sets. Jaccard similarity for sets represented as binary vectors X, Y is defined as $J_s(X,Y) = \frac{\sum_i (X_i \wedge Y_i)}{\sum_i (X_i \vee Y_i)}$. However, since the semantic vector stores count information of nodes and edges belonging to a semantic block, we define Jaccard similarity as $J_s(X,Y) = \frac{\sum_i min(X_i,Y_i)}{\sum_i max(X_i,Y_i)}$. Hence, two statements of the same instruction type in the semantic block represent different elements in the set representation of the semantic block. This is also true for outgoing edges which belong to the maximum out node degree. Therefore, two outgoing edges of a single node are different elements in the set representation. Furthermore, this definition ensures that only elements of the same instruction type are compared.

PPA is able to create an IR of an incomplete code snippet with an average correctness of 91% [10]. This gives us a threshold for Jaccard similarity of 0.91. To decide if method names and constants of a semantic block are contained in

[2] cf. http://wala.sourceforge.net

[3] Code obfuscation is not intended to be covered by our approach

another semantic block, we calculate their Jaccard containment. Jaccard containment depicts the containment ratio of an arbitrary set X in another set Y and is defined as $J_c(X, Y) = \frac{|X \wedge Y|}{|X|}$. We calculate both Jaccard containment of two method name and constant sets to evaluate whether all methods or constants are contained. We rely on a Jaccard containment value of 1.0 to satisfy the requirements of VI-B. We define containment of a code snippet in an app iff the following holds for each method in the snippet:

- For all given semantic blocks we find semantic blocks that satisfy Jaccard similarity and are contained in a single method contained in the callgraph of a given application.
- The method name set is fully contained in the same method.
- The constants set is fully contained in the same method.
- They belong to the same (nested) class.

C. Exotic Case

Empty TrustManager implementations require special treatment. They exclusively consist of overwritten methods (e. g. cf. Listing 4). These methods are mostly empty which means their PDG and methods and constants sets are also empty. Therefore, our approach cannot distinguish these methods. To avoid false positives, the TrustManager's methods `checkClientTrusted`, `checkServerTrusted` and `getAcceptedIssuers` receive special treatment. In case an empty method has been detected in the call graph of an application, we compare the method's fully qualified name with the method names given above. This way, we can successfully identify empty TrustManager implementations without false positives.

VII. Evaluation

In this section we present a detailed evaluation of our approach. We discuss benchmarks and numbers for each step of our processing pipeline (cf. Figure 1). Further, we compare our results with feedback from the Stack Overflow community, provided in the respective code threads of copied insecure snippets.

A. Evaluation of Code Extraction and Filtering

To systematically investigate the occurrence and quality of Android related code snippets on Stack Overflow, we downloaded[4] a dataset of all Stack Overflow posts in March 2016, which gave us a dataset of 29,499,660 posts. We extracted all posts which were tagged with the `android` tag - this resulted in 818,572 question threads with 1,165,350 answers. Questions in our data set had 1.4 answers on average. The oldest post in the dataset was from August, 2008. 559,933 (68.4%) of the questions and 744,166 (63.9%) of the answers contained at least one code snippet. Posts had 1,639.4 views on average. The most popular post in our dataset had 794,592 visitors.

With the oracle-based parser (as described in Section III) we filtered the 818,572 questions and the 1,165,350 answer posts from Stack Overflow which revealed 2,504 (2,474 distinct) security-related snippets from question posts and 1,517 (1,360 distinct) security-related code snippets from answer posts, respectively. In summary, using the JavaBaker oracle, we could successfully identify security-related snippets as shown in Table VII.

The majority of snippets (2,841, i.e. 70.7%) were related to the *java.security* API which implements access control, generation/storage of public key pairs, message digest, signature and secure random number generation. Most snippets were related to cryptographic key initialization, storage (e. g. *java.security.Key*, *java.security.KeyPairGenerator* or *java.security.KeyStore* – 44.9%) and message digests (*java.security.MessageDigest* – 30.4%). This attunes to our intuition, as almost all cryptographic implementations require key management and hash functions are cryptographic primitives.

Code containing Android's cryptographic API was second most prevalent and present in 1,286 (31.9%) code snippets. 1,088 (84.6%) of these code snippets applied the *javax.crypto.Cipher* API and hence, contained code for symmetric encryption/decryption. Interestingly, many snippets employ user-chosen raw keys for encryption (701 snippets with *SecretKeySpec*) instead of generating secure random keys by using the API (207 snippets with *KeyGenerator*). This indicates that most of the keys are hard-coded into the snippet, which states a high risk of key leakage if reused in an application due to reverse engineering.

The TLS/SSL package *javax.net.ssl* was used in 28.9% of the code snippets. The majority of these code snippets (545, i.e. 46.7%), contained custom TrustManagers to implement X.509 certificate validation. Optimistically, by implementing a custom trust manager, developers might aim at higher security by only trusting their own infrastructure. Practically, we observe that custom trust managers basically ignore authentication at all [2]. 17.1% of the code snippets contained custom hostname verifiers. Apache's SSL library was mainly used for enabling deprecated hostname verifiers that turned off effective hostname verification.

Code snippets containing code for BouncyCastle, SpongyCastle and SUN were rarely found. This could be due to the fact that those libraries are mostly called directly by only changing the security provider. Interestingly, nearly no (0.3%) snippets contained code for the easy-to-use jasypt and keyzcar libraries. Possible reasons could be their low popularity or good usability. Similarly, the GNU cryptographic API was rarely used. This might be due to the difficulty to integrate it in an Android application [9].

B. Evaluation of Code Classification

Altogether, we classified 1,360 distinct security-related code snippets to provide a training set for our the machine learning based classification model. We then applied the trained classifier on the complete set of 3,834 distinct security-related code snippets found in Android posts, including both questions (64.53%) and answers (35.47%). The security classification

[4] archive.org offers the option to download an archive of all Stack Overflow posts from their website, cf. https://archive.org/details/stackexchange

Namespace	Snippets	Namespace	Snippets
javax.crypto	1,286	android.security	5
Cipher	1,088	com.sun.security	5
KeyGenerator	207	gnu.crypto	47
spec.SecretKeySpec	701	java.security	2,841
spec.PBEKeySpec	69	javax.security	44
spec.DESedeKeySpec	6	javax.xml.crypto	3
spec.DESKeySpec	21	org.bouncycastle	48
spec.IvParameterSpec	338	org.spongycastle	44
spec.RC2ParameterSpec	1	org.jasypt	11
Mac	85	org.apache.http.conn.ssl	241
Sealed	8	AllowAllHostnameVerifier	184
javax.net.ssl	1,166	StrictHostnameVerifier	51
TrustManager	545	BrowserCompatHostnameVerifier	8
HostnameVerifier	200	TrustSelfSignedStrategy	1
SSLSocket	533	SSLSocketFactory	105
org.keyczar	2		

TABLE VII: Snippet counts per library.

results of the training set are presented first and are described as follows:

The qualitative description of the snippets is divided into the security categories TLS/SSL, symmetric cryptography, asymmetric cryptography, random number generation, message digests, digital signatures, authentication, and storage. For each category, we describe why we consider the respective code snippets to be insecure, what has been done wrong and why it (supposedly) has been done wrong. Whenever possible, we give counts for security mistakes and examples for the security mistakes we found.

Second, we demonstrate the feasibility of our SVM approach by discussing the overall quality of our classification model regarding precision, recall, and accuracy. Finally, we present the results for the large scale security classification of all security-related code snippets found on Stack Overflow.

1) Labeling of Training Set: As described above, the training set consists of code snippets that have been identified by the oracle-based filter to include security-related properties (cf. Section III). We classified a subset manually in order to provide supervision for the SVM.

a) TLS/SSL: We found 431 (31.48%) of all snippets in the training set to be TLS related, among these we rated 277 (20.23%) as insecure. In other words, almost one third of security-related discussions seem to target communication security and more than half of the related snippets would introduce a potential risk in real-world applications. The majority of TLS snippets are insecure because of using a default hostname verifier or overriding the default TrustManager of *java.net.ssl* when initializing custom TLS sockets. Every single custom TrustManager implementation we found consists of empty methods that disable certificate validation checks completely, while none of the custom TrustManager are used to implement custom certificate pinning, which is the reasonable and secure use case for creating custom TrustManagers. This correlates to our assumption stated in Section VII-A. An empty TrustManager is implemented by 156 snippets, while 6 snippets use the *AllowAllHostNameVerifier* - and 2 implemented both. We found 42 snippets that override the verification method of *HostnameVerifier* of *java.net.ssl* by returning `true` unconditionally, which ultimately disables hostname verification completely (cf. Listing 1). This change to the HostnameVerifier implements the same behavior as *AllowAllHostNameVerifier*.

We found several snippets that modify the list of supported ciphers. In all cases, insecure ciphers were added to the list. We assume this is caused by reasons of either legacy or compatibility.

b) Symmetric Cryptography: We found 189 (13.80%) of all snippets in the training set to be related to symmetric cryptography, among these we rated 159 (11.61%) of the snippets as insecure. For example, we found snippets containing encryption/decryption methods with less than 5 lines of code, implementing the minimum of code needed to accomplish an encryption operation. These snippets were insecure by using the cipher transformation string ”AES” which uses ECB as default mode of operation (cf. Section IV-B2). Developers might be unaware of this default behavior or of ECB being insecure.

Another example are snippets that create raw keys and raw IVs using empty byte arrays (i. e. byte arrays which consist of zeros only), derive raw IVs directly from static strings, or by using the array indexes as actual field values as shown in Listing 2. Other snippets derive raw keys directly from strings that were mostly simple and insecure passphrases, e.g. *”ThisIsSecretEncryptionKey”*, *”MyDifficultPassw”*. We also found snippets that initialized the IV using the secret key.

c) Asymmetric Cryptography: We found 59 (4.3%) of all code snippets in the training set to include asymmetric cryptography API calls, among these 13 (0.94%) of the snippets were classified as insecure. Considering the importance of public key cryptography in key distribution and establishing secure communication channels, 4.3% is quite low and corresponds to our assumption in Section VII-A. All insecure snippets used weak key lengths which varied between 256 and 1024 bits for RSA keys. Obviously, recommendations from public authorities (e.g. the NIST) regarding secure cryptographic parameters are not fully taken into consideration.

d) (Secure) Random Number Generation: We found 30 (2.19%) of all code snippets in the training set to include (secure) random number generation API calls, among these 29 (2.11%) of the snippets were classified as insecure. All insecure snippets explicitly seeded the random number generator with static strings (cf. Listing 3). Replacing the random number generator's seed this way and not supplementing it results in low entropy [31].

e) Digital Signatures and Message Digests: Overall, 279 (20.37%) snippets contain digital signatures related API calls. We classified none of them as insecure. This is a remarkable and unexpected observation, especially compared to the high number of insecure snippets in discussions regarding symmetric cryptography. To explain this we had a closer look at the relevant code snippets: calls to the digital signatures API are most often related to extracting existing signatures, not to validate them or generate new ones. Such an interactive query of existing signatures is not very error-prone regarding security.

Further, we found 392 (28.63%) snippets to contain message digest related API calls, among these 14 (1.02%) were classified as insecure due to usage of weak hash algorithms. Again, compared to the quantity of insecure snippets of other categories this is quite a low percentage. In generating a message digest the biggest pitfall is choosing a weak hash function. We assume that state-of-the-art hash functions are relatively established in the Stack Overflow community.

f) Remaining: 19 (1.38%) of the snippets contained authentication code, where one snippet was classified as insecure. Eight (0.58%) contained secure storage code, where three snippets were classified as insecure.

g) Not Security-Relevant: We classified 342 snippets as not security-relevant as defined in III-C.

2) Model Evaluation of the SVM Code Classifier: Overall, after removing some duplicates, the training data set consisted of 1,360 samples, out of which 420 code snippets were identified as insecure. As introduced in Section V-B, we use a *tf-idf* vectorizer to convert code snippets into numeric vectors for training.

To illustrate how our approach works, Figure 3 illustrates the projection of our training samples in $2d$ space by a common dimensionality reduction method, *i.e.,* Principle Component Analysis (PCA) [40]. We leverage a RBF (Radial Basis Function) kernel function to tackle the non-linearity hidden in the projected training samples. The RBF kernel is a well known type of kernel to model non-linearity of data. It maps the non-linear input data to a high dimensional linear feature space, such that the data becomes linearly separable. We can see that even in $2d$ space where some relevant information might be lost, the SVM classifier produces a good class boundary for both the secure (blue dots) and insecure (red dots) code samples.

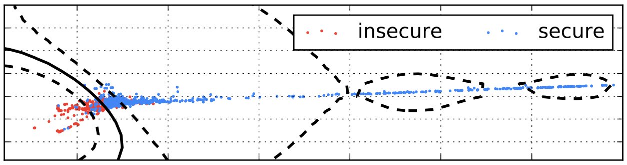

Fig. 3: SVM with RBF kernel is trained on the training dataset, where the high dimensional training samples are projected on 2-dimensional using PCA. Solid contour line represents the classification boundary and dashed lines indicate the maximal margin learned by SVM. Insecure code snippets are marked as red circles, and secure ones are marked as blue circles.

Next, we evaluate our SVM model quantitatively by cross validating the training data set. First, we conduct a grid search on SVM to estimate the optimal penalty term C (cf. (1)) with respect to classification accuracy. Since the training data contains very high dimensional features, we use a linear kernel for SVM instead of RBF kernel in previous $2d$ demonstration. The optimal parameter C is determined to be 0.644. We evaluate the model on various training sizes with respect to precision, recall and accuracy. A discussion on these evaluation metrics can be found in [41]. In our setup, we consider insecure samples as positive and secure ones as negative. Therefore, the precision score measures how many predicted insecure snippets are indeed insecure, the recall score evaluates how many real insecure snippets are retrieved from all insecure snippets, and finally the accuracy score measures an overall classification performance taking both positive and negative samples into account.

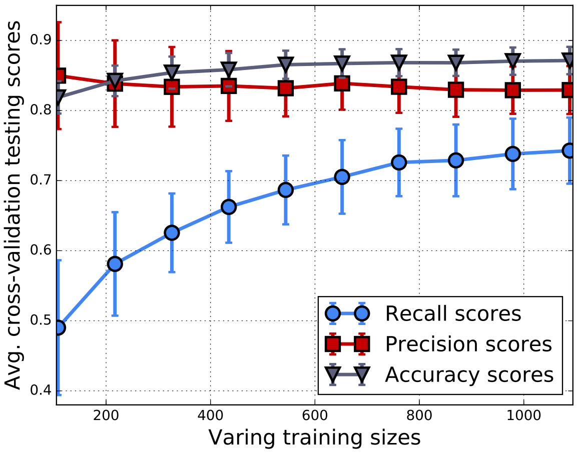

Fig. 4: Binary SVM with linear kernel is trained over varying training sizes. Cross validation is performed on each of these subsets of the training data set and evaluated with respect to precision, recall and accuracy scores.

In Figure 4, we report the learning curve of our model with respect to varying training sizes. For each training size, a subset of the training data set is classified by the model with a 50-repetition cross validation. In each repetition, we randomly hold out 20% of the training samples as testing set, and train on the remaining samples. Finally, we average the testing scores on all repetitions and plot the mean scores with standard deviation as the error bar. The results present a good precision and accuracy on varying training sizes, as the mean scores are approximately all above 0.8. The constantly developing precision curve illustrates that our model performs very well on detecting real insecure snippets instead of introducing too many false positives, even on a small training size. On the other hand, we see the recall curve is relatively poor on small training size. However, it reaches nearly 0.75 when we have more than $1,000$ training samples. Accuracy also improves with increasing training samples. The variance of the accuracy is canceled by combining both precision and recall.

For completeness, we conduct a 5-fold cross validation on the whole training data set with optimal penalty term $C = 0.644$. We report the confusion matrix of the best fold in Table VIII. Note that the test size for each fold is 272.

True/Predicted	Secure (-1)	Insecure (+1)
Secure (-1)	181	7
Insecure (+1)	19	65
Summary	accuracy: 0.904	precision: 0.903

TABLE VIII: Confusion matrix

To conclude our model evaluation, we argue that our SVM model could be improved by a more exhaustive feature engineering phase and by increasing the size of the training data set. In our experiment, we only remove comments in code snippets as a preprocessing step. In practice, this will be enhanced by applying a more complex token parser, e.g., static code parser, to generate better quality of features. Moreover, we did not leverage control flow information, which is considered informative of predicting security level, to enhance the model. A possible refinement would be the encoding of the relative position of each token in the snippet into the features. However, this could double the size of input feature dimension. Due to the complexity of model pruning and the limit of the training sample size, we decided to leave it for future work. Given the fact that the performance of the SVM model already achieves a level of practicability, we think that machine learning based approaches have the potential to support security code analysis.

3) Large Scale Classification: We applied our SVM code classifier on the complete set of 3,834 distinct security-related snippets from Stack Overflow, including question and answer snippets. Overall, we found 1,161 (30.28%) insecure snippets and 2,673 (69.72%) secure snippets. Out of the 1,360 distinct snippets found in answer posts, 420 (30.88%) snippets were classified as insecure and and 940 (69.12%) as secure. For the 2,474 distinct snippets we detected in questions posts, 741 (29.95%) snippets were classified as insecure and 1,733 (70.05%) as secure.

C. Evaluation of Code Detection

We applied our pipeline (cf. Section VI) to a large corpus of free Android applications from Google Play. Beginning in October 2015, we successfully downloaded 1,305,820 free Android applications from Google Play[5]. We re-downloaded new versions until May 2016. The majority of apps received their newest update within the last 12 months.

1) Apps with Copied and Pasted Code Snippets: Overall, we detected copied and pasted snippets in 200,672 (15.4%) apps. Of these apps, 198,347 (15.2%) contain a question snippet and 40,786 (3.1%) apps contain an answer snippet. An overwhelming amount of apps contain an insecure code snippet: 196,403 (15%) apps contain at least one. The top offending snippet has been found in 180,388 (13.81%) apps and is presented in Listing 4. The remaining insecure snippets were found in 43,941 (3,37%) distinct apps.

We found 506,922 (38.82%) apps that contain a secure snippet. The most frequent secure snippet was detected in 408,011 (31.24%) apps while the remaining snippets were contained in less then 73,839 (5.65%) apps. On average, an insecure snippet is found in 4,539.96 apps, while a secure code snippet is found in 10,719.83 apps.

To investigate insecure snippets that were detected by our fully automated processing pipeline in detail, we performed a manual post-analysis of the categories described in Section IV. To be more precise, we examined all security-related snippets that were detected in applications and sorted them by category. In the following, we give counts for affected applications for each security category. The given percentage values are related to applications that contain a snippet from Stack Overflow. Further, we discuss the most offending snippets and estimate their practical exploitability.

2) SSL/TLS: The highest number of apps that implemented an insecure code snippet used this snippet to handle TLS. 183,268 (14.03%) apps were affected by insecure TLS handling through a copied and pasted insecure code snippet. Conversely, only 441 (0.03%) of all apps contained a secure code snippet related to TLS. For the large majority of 182,659 (13.98%) apps with an insecure TLS snippet, their code snippet matches a question code snippet on Stack Overflow, while only 22,040 (1.68%) apps contain an insecure TLS snippet that was present in an answer on Stack Overflow. A high risk example in this category is given by the top offending snippet as presented in Listing 4, which uses an insecure custom TrustManager as described in Section VII-A. Missing server verification enables Man-In-The-Middle attacks by presenting malicious certificates during the TLS handshake. This snippet is a real threat with high risk of exploitation in the wild, as shown in [2].

3) Symmetric Cryptography: The second highest number of insecure code snippets in the wild were used for symmetric cryptography in 21,239 (1.62%) apps. 19,452 (1.48%) of the apps with a code snippet that was related to symmetric cryptography had integrated a secure snippet. With a count of 19,189 apps, slightly more apps contain an insecure question snippet than an insecure answer snippet, which happened in 15,125 apps. The insecure snippet with the highest copy and paste count (found in 18,000 apps) within this category proposes AES in ECB mode. According to [3] this is vulnerable to chosen-plaintext attacks. Further, applications that include snippets with hard-coded cryptographic keys can most often be reverse-engineered without much effort. This leads leads to key leakage and therefore states a high risk (at least in the case where the key is not obfuscated).

[5] cf. https://play.google.com

4) Asymmetric Cryptography: We found only 114 (0.01%) apps that contained an insecure code snippet related to asymmetric cryptography, 698 (0.05%) apps contained a secure asymmetric cryptography related snippet. 114 apps with insecure snippets contain an insecure question snippet. 29 apps implemented a secure answer snippet, 688 a secure question snippet.

5) Secure Random Number Generation: 8,228 (0.63%) apps contain an insecure code snippet related to random number generation, while 4,100 (0.31%) apps contained a secure snippet. Most insecurities in this category come from question snippets (this was true for 8,227 apps, while 7,991 apps contain an insecure answer snippet).

6) Hashes: For hash functions, the majority of apps containing code snippets from Stack Overflow contained secure code snippets: This was true in 4,012 (0.3%), 14 apps contained an insecure one.

7) Signatures: 15 apps contained a secure signature related snippet, while no insecure snippet was found in apps in this category. All of those snippets could be found in questions on Stack Overflow.

8) Not Security-Related: Some of the snippets that were detected in apps could not be assigned to one of the categories above because they were not security-related as described in III-C. 498,046 (38.1%) apps contained a snippet that was not security-related and therefore classified as secure.

The most frequent secure snippet found in 408,011 apps was also not security-related. Therefore, considering security-related snippets only, we can state that significantly more Android applications contain an insecure snippet (196,403) than a secure one (73,839) (cf. Section VII-C1).

9) Sensitive App Categories: The largest number of sensitive apps that use insecure copied and pasted code snippets are 14,944 business apps, 4,707 shopping apps, and 4,243 finance apps. We find this result rather surprising, as we would have expected that security receives special consideration for these types of applications. Especially, finance apps have access to bank account information and therefore we would have expected them to be developed with extra care. Security and privacy is especially important in apps that handle medical data, as leaked sensitive data can have a severe impact on users. We found 2,000 medical and 4,222 health&fitness apps that copied and pasted vulnerable Stack Overflow code. Apps that are used for communication (3,745 apps) and social media (4,459 apps) are also widely affected.

10) Download Counts: In order to prove that we did not only inspect the long tail of unpopular apps provided by Google Play we provide download counts for apps that contain insecure snippets in Figure 6.

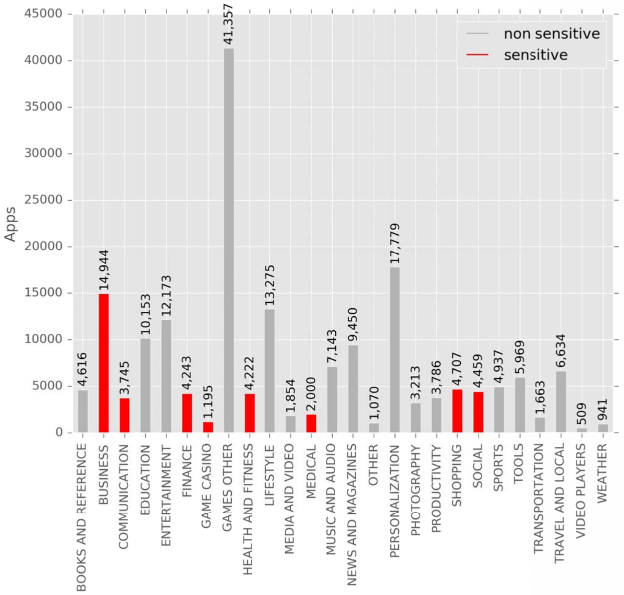

Fig. 5: Distributions of insecure snippets found in Apps

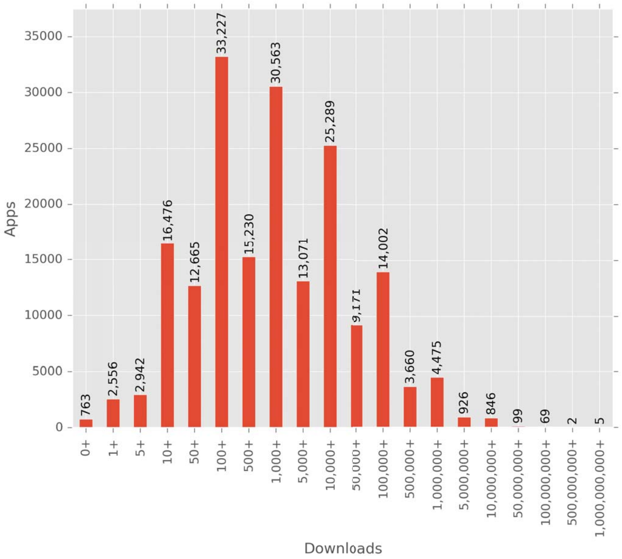

Fig. 6: Download counts for Apps with insecure snippets

D. Evaluation of Community Feedback

For those insecure snippets that we detected in Android applications, we analyzed the community feedback on Stack Overflow as this represents the current public evaluation of posted code snippets. The available feedback mechanisms allow a very general evaluation of code snippets, e. g. with the presentation of view counters and the individual post score which results from up-/down-votes by the community. In addition to this, code snippets can be commented which is used to provide a more detailed feedback.

We analyzed if the existing feedback system provided by

SO is capable of informing the user about insecure snippets in an adequate way. At that, we analyze if the currently given feedback by the community is preventing or contributing to copy and paste of insecure code into Android applications.

1) Scoring: According to Stack Overflow, a question snippet is supposed to be up-voted if it shows reasonable research effort in order to motivate the community to reply to it. Therefore, with a pure focus on security aspects we expect insecure question snippets to be up-voted, as insecure code snippets intuitively demand more community research than secure ones. In contrast to question scores, answers are up-voted (according to Stack Overflow) if they are estimated useful. Regarding the score of insecure answer snippets we expect a lower score as these do not provide a useful answer considering security-related snippets.

The results in table IX show that the scoring of insecure question snippets contradict to our assumption, because the secure ones have a higher score. In other words, the community assigns a higher needed research effort to questions with secure snippets. This is counter-intuitive from the security perspective and therefore leads to the conclusion that question scoring is not an adequate way of evaluating security. Of course, the community estimates needed research effort on the basis of a diversity of aspects, which outweigh security considerations. However, regarding answers, the low scoring of insecure snippets (as depicted in Table IX) correspond to our expectations. Again, this positive correlation might be caused by a variety of aspects, but from the security point of view it reveals the desired community behavior. However, aspects that are currently taken account for answer scoring are likely to be weighted differently in the future.

Next, we additionally include security warnings in our evaluation. Here, the scoring for questions given by the community are consistent with our intuition: Insecure questions including a warning are assigned with a higher scoring (corresponding to higher estimated research effort) than questions without such a warning. However, the scoring estimation regarding answer posts contradicts the desired community behavior: Insecure answers with security warning are scored significantly higher compared to the ones without warnings (cf. Table IX). Therefore, the influence of warnings (in answers) on the community score is highly questionable. A high scored answer with assigned security warning might confuse the reader.

The following considerations try an explanation of this result. Though a security warning should have a strong impact on the evaluation, the author of the warning can down-vote the score only once. On the one hand, this gives the community the ability to review the warning and to further reduce the score or to comment disagreement. On the other hand, the results show that the scoring of insecure snippets is partly contradicting and we did not find a single warning that has been questioned in a subsequent comment. Acar et al. [42] have shown that developers prefer functional snippets over secure snippets when implementing security related tasks in Android. This preference might also influence the scoring of security-related snippets which can result into a score that mostly considers functionality as the definition of a useful answer.

When solely taking security considerations into account, we conclude that the currently deployed feedback system is insufficient for providing reliable and precise security estimation to the user.

Metadata	Secure	Insecure	Insecure+Warning	Insecure-Warning
Avg. Score Q/A	3.4/4.8	1.7/4.4	2.4/15.5	2.3/6.2
Avg. Viewcount Q/A	1467/4341	2254/8117	4081/16534	2812/10001

TABLE IX: Community feedback for security-related snippets regarding questions (Q) and answers (A)

2) Impact on Copy and Paste: Next, we investigated if view count, warnings, and score of insecure snippets have an impact on the extent they are copied into applications. We first ordered all snippets according to their detection rate (the amount of applications that contained that snippet). For the snippets that ranked highest and lowest on this list (respectively 25% – we refer to these as top and bottom tier) we extracted the corresponding metadata from Stack Overflow. This allowed us to observe possible correlations between view counts, warnings, and scoring to the actual copy and paste rate.

For both score and view count we found a positive correlation: A higher score or view count corresponds to an increased copy and paste count, as depicted in Table X. This yields for both, questions and answers.

Interestingly, we see the opposite behavior with regard to warnings: Snippets that have been commented with security warnings are copied more often into applications than those without. An exceptionally striking example for this observation is the top offending snippet which was copied 180,388 times despite of being commented with warnings (cf. Listing 4).

Metadata	Questions	Answers
Avg. Score (top/bottom tier)	1.87/1.27	7.21/6.37
Avg. Viewcount (top/bottom tier)	2,792/1,373	11,915/7805

TABLE X: Correlation of community feedback with copy and paste count of insecure code snippets

E. Limitations

Besides the limitations of the intermediate steps discussed in Sections III-C and IV-C, our processing pipeline does not fully prove that copied snippets originate from Stack Overflow. To illustrate this objection, there theoretically could exist a third platform where snippets are copied and inserted to both, Stack Overflow and Android applications. However, Stack Overflow is the most popular platform for developer discussions[6]. Further, [42] et al. showed that developers most often rely on Stack Overflow when solving security-related programming problems at hand. And finally, we found a positive correlation of the snippets view counts with their detected presence in applications (as discussed in Section VII-D2). Therefore, it is very likely that snippets originate from Stack Overflow.

[6] cf. http://www.alexa.com/topsites

VIII. Related Work

We focus on related work in four key areas, i.e. security of mobile apps, developer studies, investigation of Stack Overflow, and detection of code reusage in apps.

A. Security of Mobile Apps

Fahl et al. analyzed the security of TLS code in 13,500 popular, free Android applications [2]. They found that 8% were vulnerable to Man-In-The-Middle attacks. In follow-up work, they extended their investigation to iOS and found similar results: 20% of the analyzed apps were vulnerable to Man-In-The-Middle attacks [17]. Oltrogge et al. [18] investigated the applicability of public key pinning in Android applications and came to the conclusion that pinning was not as widely applicable as commonly believed. Egele et al. [3] investigated the secure use of cryptography APIs in Android applications and found more than 10,000 apps misusing cryptographic primitives in insecure ways. Enck et al. [43] presented TaintDroid, a tool that applies dynamic taint tracking to reveal how Android applications actually use permission-protected data. They found a number of questionable privacy practices in apps and suggested modifications of the Android permission model and access control mechanism for inter-component communication. Chin et al [44] characterized errors in inter-application communications (*intents*) that can lead to interception of private data, service hijacking, and control-flow attacks. Enck et al. [45] analyzed 1,100 Android applications and reported widespread security problems, including the use of fine-grained location information in potentially unexpected ways, using device IDs for fingerprinting and tracking and transmitting device and location in plaintext. Poeplau et al. [46] reported that many apps load application code via insecure channels allowing attackers to inject malicious code into benign apps.

B. Developer Studies

The bulk of identified security issues are attributed to developers that are poorly skilled in security-related programming. Core reasons for these issues were identified in a developer study conducted by Fahl et al. [17]: developers that customized TLS code disabled TLS functionality during testing and forgot to re-enable it for production, and they did not understand the security guarantees provided by and the security consequences imposed by improper TLS use. Similar root causes were reported by Georgiev et al. [47], showing that developers were confused by the many parameters, options and defaults of TLS APIs. Both papers explicitly mentioned Stack Overflow as a platform that provides various solutions for ”circumventing” TLS-related error messages by disabling TLS features. Acar et al. [42] conducted a laboratory study to investigate the impact of information sources on code security and found that developers using Stack Overflow for looking up security-related issues produced the most functional but also the most insecure code, whereas participants using Android's official documentation produced more secure but less functional code.

C. Investigation of Stack Overflow

Treude et al. [48] report that developer discussion platforms like Stack Overflow are very effective at code reviews and conceptual questions. Vasilescu et al. [49] investigate the interplay of Stack Overflow activity and development process on GitHub. They conclude that knowledge of the GitHub community flows into Stack Overflow. In turn, this knowledge increases the number of commits of Stack Overflow users on GitHub. Vasquez et al. [50] created an algorithm to link Stack Overflow questions with Android classes detected in source code. They found that Android developer question counts peak on Stack Overflow immediately after APIs receive updates that modify their behavior.

D. Detection of Code Reusage in Apps

Jiang et al. [39] compared the similarity of abstract syntax trees to detect code duplicates in source code. Hanna et al. [38] created k-gram streams from bytecode basic blocks. Each k-gram defines a program feature. A code snippet and an application is represented by the binary feature vector that is created using universal hashing over k-grams. They decide if a code snippet is contained in an app by dividing the number of common features by the number of features of the code snippet. While their approach works in benign scenarios, it is not robust against trivial code modifications (e. g. reordering of instructions or renaming of variables). Crussell et al. [35], [36] detect code clones by searching for subgraph isomophisms of program dependency graphs (PDG). Their approach is able to detect code fragments that perform similar computations through different syntactic variants [51] and robust against trivial modifications, constant renaming and method/class restructuring. Chen et al. [51] use Control Flow Graphs (CFG) in combination with opcodes to detect code clones in Android applications. They define a geometry characteristic called centroid to embed a CFG into vector space.

IX. Countermeasures

On the one hand, there is a significant amount of secure code on Stack Overflow that finds its way into Android applications. The question is how we can reinforce this flow that surely is beneficial for the Android ecosystem. On the other hand, we also observed a vast amount of highly insecure code snippets. How can we prevent insecure code snippets from being copied?

In Section VII-D we showed that the deployed scoring system of Stack Overflow is not fine-grained enough to mirror security concerns provided by the community. This suggests a scoring system that is purely focused on security aspects. However, a fine-grained scoring system will possibly also include equitable aspects such as code stability, efficiency, or audibility. This might impact the overall usability of Stack Overflow and we fully understand the decision for just one score for each post.

Instead of expanding (and maybe complicating) the scoring system of posts, we propose another solution: Classification of code snippets into secure and insecure is fully automated in our approach, which allows us to implement a browser-plugin that directly indicates security issues by real-time classification of snippets. This includes both, snippets copied to the clipboard and snippets parsed on the actually watched discussion thread. Such a browser plugin is not limited to Stack Overflow, but would work without much effort for any source of snippets in the web. We are currently actively developing such a browser plugin for Mozilla Firefox and Chrome.

X. Conclusion

We present, implement and evaluate the first systematic and fully automated processing pipeline for measuring the flow of secure and insecure code snippets from Stack Overflow into Android applications available at Google Play. First, we scanned public discussion threads for code snippets filtering out the security-relevant subset using a robust oracle-based approach. Next, we applied machine learning classification to receive a security scoring for code snippets. By constructing an abstract representation in form of a program dependency graph for each snippet, we detect their presence in compiled Android applications.

We evaluate the performance of our approach by searching for security-related code snippets from Stack Overflow in 1,305,820 Android applications available at Google Play. We show that 196,403 (15%) of the 1.3 million Android applications contain vulnerable code snippets that were very likely copied from Stack Overflow (cf. Section VII-E). We detected 73,839 applications (cf. SectionVII-C8) using a secure code snippet from Stack Overflow. By analyzing metadata, we gain insight into developer behavior: From typical post upvoting trends and popularity of insecure code to favoured security libraries of specific domains (such as finance and gaming) we are able to draw interesting new conclusions on behavior of the Android developer community. We expect that a future systematic investigation augmenting metadata of security-related code snippets with metadata of affected Android applications will serve as a rich source of new and interesting research questions.

Answering the original research question whether Stack Overflow should be considered harmful is difficult. Although we cannot guarantee that code snippets we detected originate from Stack Overflow with 100% certainty, the fact that Stack Overflow is the most popular Q&A site for Android developers and many of them heavily rely on the posted discussions and solutions [42] suggests Stack Overflow's significant responsibility. Luckily, there is hope on the horizon. We presented a fully automated solution to detect security-related code snippets posted on Stack Overflow and analyze their security. Stack Overflow could implement our or a similar approach and either remove insecure code snippets entirely or add clear security warnings to prevent developers from copying and pasting those snippets into their applications. Identifying the most effective way of dealing with insecure code snippets and communicating risks to developers is an interesting and challenging field of work for future usable security and privacy researchers.

Acknowledgements

The authors would like to thank Siddharth Subramanian for his strong support with JavaBaker and the anonymous reviewers for their helpful comments. This work was supported in part by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) (FKZ: 16KIS0344,16KIS0656)

References

[1] A. Porter Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *Proceedings of the 18th ACM Conference on Computer and Communication Security*. ACM, 2011.

[2] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben, "Why Eve and Mallory love Android: An analysis of Android SSL (in) security," in *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)*. ACM, 2012.

[3] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An Empirical Study of Cryptographic Misuse in Android Applications," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. ACM, 2013.

[4] Oracle, "Java SE 8," http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html.

[5] A. Zhitnitsky, "Libraries on GitHub," http://blog.takipi.com/we-analyzed-60678-libraries-on-github-here-are-the-top-100, 2015.

[6] T. Duong and J. Rizzo, "Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET," in *2011 IEEE Symposium on Security and Privacy*, 2011.

[7] A. Dey and S. Weis, "Keyczar: A Cryptographic Toolkit," 2008.

[8] jasypt, "Java Simplified Encryption," http://www.jasypt.org, 2014.

[9] D. González, O. Esparza, J. L. Muñoz, J. Alins, and J. Mata, "Evaluation of Cryptographic Capabilities for the Android Platform," in *Future Network Systems and Security: First International Conference, FNSS 2015, Paris, France, June 11-13, 2015, Proceedings*, 2015.

[10] B. Dagenais and L. Hendren, "Enabling Static Analysis for Partial Java Programs," in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, ser. OOPSLA '08, 2008.

[11] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API Documentation," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE, 2014.

[12] Y. Sheffer and R. Holz, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)," Tech. Rep., 2015.

[13] B. Kaliski, "PKCS #5: Password-Based cryptography specification version 2.0," Internet Engineering Task Force, RFC, 2000.

[14] J. Manger, "A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) As Standardized in PKCS #1 V2.0," in *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO, 2001.

[15] E. Barker and A. Roginsky, "Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths," http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf.

[16] D. Bleichenbacher, "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1," in *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '98, 1998.

[17] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking ssl development in an appified world," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ser. CCS '13. ACM, 2013, pp. 49–60.

[18] M. Oltrogge, Y. Acar, S. Dechand, M. Smith, and S. Fahl, "To Pin or Not to Pin—Helping App Developers Bullet Proof Their TLS Connections," in *Proc. 24th USENIX Security Symposium (SEC'15)*. USENIX Association, 2015.

[19] S. Turner and T. Polk, "Prohibiting Secure Sockets Layer (SSL) Version 2.0," 2011.

[20] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," Tech. Rep., 2016.

[21] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," Tech. Rep., 2008.

[22] S. Vaudenay, "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ..." in *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology*, ser. EUROCRYPT '02, 2002.

[23] J. Kelsey, B. Schneier, and D. Wagner, "Related-key Cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA," in *Proceedings of the First International Conference on Information and Communication Security*, ser. ICICS '97, 1997.

[24] N. J. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. Schuldt, "On the Security of RC4 in TLS." in *Usenix Security*, 2013.

[25] S. Lucks, "Attacking Triple Encryption," in *Fast Software Encryption: 5th International Workshop, FSE' 98 Paris, France, March 23–25, 1998 Proceedings*, S. Vaudenay, Ed., 1998.

[26] S. Vaudenay, "On the Weak Keys of Blowfish," in *Fast Software Encryption*, 1996.

[27] O. Kara and C. Manap, "A New Class of Weak Keys for Blowfish," in *Fast Software Encryption*, 2007.

[28] "Java Security and Related Topics," http://armoredbarista.blogspot.de/2012/09/rsaecb-how-block-operation-modes-and.html.

[29] D. Giry, "Keylength," https://www.keylength.com/en/4/.

[30] SANS Institute, "Four Attacks on OAuth - How to Secure Your OAuth Implementation," https://www.sans.org/reading-room/whitepapers/application/attacks-oauth-secure-oauth-implementation-33644.

[31] "Android Developers," http://android-developers.blogspot.de/2013/02/using-cryptography-to-store-credentials.html.

[32] B. Schölkopf, "Statistical Learning and Kernel Methods," Microsoft Research, Tech. Rep., 2000.

[33] H. C. Wu, R. W. P. Luk, K. F. Wong, and K. L. Kwok, "Interpreting TF-IDF Term Weights As Making Relevance Decisions," *ACM Trans. Inf. Syst.*, 2008.

[34] Google Inc., "Enjarify," https://github.com/google/enjarify.

[35] J. Crussell, C. Gibler, and H. Chen, "Attack of the Clones: Detecting Cloned Applications on Android Markets," in *ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, 2012.

[36] ——, "AnDarwin: Scalable Detection of Semantically Similar Android Applications," in *ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, 2013.

[37] K. Riesen and H. Bunke, "Graph Classification Based on Vector Space Embedding," *International Journal of Pattern Recognition and Artificial Intelligence*, 2009.

[38] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications," in *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA'12, 2013.

[39] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07, 2007.

[40] L. I. Smith, "A Tutorial on Principal Components Analysis," Cornell University, USA, Tech. Rep., 2002.

[41] J. Makhoul, F. Kubala, R. Schwartz, and R. Weischedel, "Performance Measures For Information Extraction," in *In Proceedings of DARPA Broadcast News Workshop*, 1999.

[42] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You Get Where You're Looking For: The Impact of Information Sources on Code Security," *IEEE Symposium on Security and Privacy (SP)*, 2016.

[43] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proc. 9th Usenix Symposium on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, 2010.

[44] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing Inter-application Communication in Android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011.

[45] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in *Proceedings of the 20th USENIX Conference on Security*, Aug. 2011.

[46] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications," in *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. The Internet Society, 2014.

[47] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software," in *Proceedings of the 2012 ACM Conference on Computer and Communications security*. ACM Press, Oct. 2012.

[48] C. Treude, O. Barzilay, and M.-A. Storey, "How Do Programmers Ask and Answer Questions on the Web?: Nier track," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011.

[49] B. Vasilescu, V. Filkov, and A. Serebrenik, "StackOverflow and GitHub: Associations Between Software Development and Crowdsourced Knowledge," in *Social Computing (SocialCom), 2013 International Conference on*. IEEE, 2013.

[50] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How Do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK," in *proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014.

[51] K. Chen, P. Liu, and Y. Zhang, "Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. ACM, 2014.

APPENDIX

Listing 1: Empty HostnameVerifier - Accepts all hostnames

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

Listing 2: Sample of static IVs and Keys in Snippets

```
byte[] rawSecretKey = {0x00, 0x00, 0x00, 0x00, 0
    x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
    , 0x00, 0x00, 0x00, 0x00};
String iv = "00000000";
byte[] iv = new byte[] { 0x0, 0x1, 0x2, 0x3, 0x4,
     0x5, 0x6, 0x7, 0x8, 0x9, 0xA, 0xB, 0xC, 0xD,
     0xE, 0xF };
```

Listing 3: String used to replace the random number generators seed

```
byte[] keyStart = "this is a key".getBytes();
SecureRandom sr =
    SecureRandom.getInstance("SHA1PRNG");
sr.setSeed(keyStart);
```

Listing 4: Top offending snippet

```
TrustManager tm = new X509TrustManager() {
    public void checkClientTrusted(
        X509Certificate[] chain, String authType)
         throws CertificateException { }
    public void checkServerTrusted(
        X509Certificate[] chain, String authType)
         throws CertificateException { }
    public X509Certificate[] getAcceptedIssuers()
         { return null; }
};
```