An Empirical Study of Cryptographic Misuse in Android Applications

Manuel Egele, David Brumley
Carnegie Mellon University
{megele,dbrumley}@cmu.edu

Yanick Fratantonio, Christopher Kruegel
University of California, Santa Barbara
{yanick,chris}@cs.ucsb.edu

ABSTRACT

Developers use cryptographic APIs in Android with the intent of securing data such as passwords and personal information on mobile devices. In this paper, we ask whether developers use the cryptographic APIs in a fashion that provides typical cryptographic notions of security, e.g., IND-CPA security. We develop program analysis techniques to automatically check programs on the Google Play marketplace, and find that 10,327 out of 11,748 applications that use cryptographic APIs – 88% overall – make at least one mistake. These numbers show that applications do not use cryptographic APIs in a fashion that maximizes overall security. We then suggest specific remediations based on our analysis towards improving overall cryptographic security in Android applications.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Android program slicing, Misuse of cryptographic primitives

Keywords

Software Security, Program Analysis

1 Introduction

Developers use cryptographic primitives like block ciphers and message authenticate codes (MACs) to secure data and communications. Cryptographers know there is a right way and a wrong way to use these primitives, where the right way provides strong security guarantees and the wrong way invariably leads to trouble.

In this paper, we ask whether developers know how to use cryptographic APIs in a cryptographically correct fashion. In particular, given code that type-checks and compiles, does the implemented code use cryptographic primitives correctly to achieve typical definitions of security? We assume that developers who use cryptography in their applications make this choice consciously. After all, a developer would not likely try to encrypt or authenticate data that they did not believe needed securing.

We focus on two well-known security standards: security against chosen plaintext attacks (IND-CPA) and cracking resistance. For each definition of security, there is a generally accepted right and wrong way to do things. For example, electronic code book (ECB) mode should only be used by cryptographic experts. This is because identical plaintext blocks encrypt to identical ciphertext blocks, thus rendering ECB non-IND-CPA secure. When creating a password hash, a unique salt should be chosen to make password cracking more computationally expensive.

We focus on the Android platform, which is attractive for three reasons. First, Android applications run on smart phones, and smart phones manage a tremendous amount of personal information such as passwords, location, and social network data. Second, Android is closely related to Java, and Java's cryptographic API is stable. For example, the `Cipher` API which provides access to various encryption schemes has been unmodified since Java 1.4 was released in 2002. Third, the large number of available Android applications allows us to perform our analysis on a large dataset, thus gaining insight into how application developers use cryptographic primitives.

One approach for checking cryptographic implementations would be to adapt verification-based tools like the Microsoft Crypto Verification Kit [7], Murφ [22], and others. The main advantage of verification-based approaches is that they provide strong guarantees. However, they are also heavy-weight, require significant expertise, and require manual effort. The sum of these three limitations make the tools inappropriate for large-scale experiments, or for use by day-to-day developers who are not cryptographers.

Instead, we adopt a light-weight static analysis approach that checks for common flaws. Our tool, called CryptoLint, is based upon the Androguard Android program analysis framework [12]. The main new idea in CryptoLint is to use static program slicing to identify flows between cryptographic keys, initialization vectors, and similar cryptographic material and the cryptographic operations themselves. CryptoLint takes a raw Android binary, disassembles it, and checks for typical cryptographic misuses quickly and accurately. These characteristics make CryptoLint appropriate for use by developers, app store operators, and security-conscious users.

Using CryptoLint, we performed a study on crypto-

CCS'13, November 04 - 08 2013, Berlin, Germany
Copyright 2013 ACM 978-1-4503-2477-9/13/11 $15.00
http://dx.doi.org/10.1145/2508859.2516693.

graphic implementations in 11,748 Android applications. Overall we find that 10,327 programs – 88% in total – use cryptography inappropriately. The raw scale of misuse indicates a widespread misunderstanding of how to properly use cryptography in Android development.

We find there are exacerbating factors, and suggest remediations. First, while current developer tools can check a number of security properties, using cryptography correctly is not one of them. Adding light-weight checks, such as in CryptoLint, would improve security. Second, implementations abstract away semantic assumptions about the correct use of cryptographic primitives. For example, the documentation for CBC encryption does not state that the initialization vector should not be a constant. Adding a security discussion to cryptographic API documentation would address this problem. Third, the default behavior in cryptographic libraries is often not a recommended practice. For example, the predominant Android Java security provider API defaults to using the ECB block cipher mode for AES encryption. To remedy this problem, we suggest changing the default behavior to a more secure variant.

Contributions: Overall, our contributions are:

- We propose light-weight static analysis techniques and tools that can catch common cryptographic misuses (§5). Application developers and app store maintainers can use the tools to identify likely misuses in cryptography before an end-user uses the application.
- We perform a large-scale experiment to measure cryptographic misuse in Android (§6). To the best of our knowledge, we are the first to perform such a study at scale, demonstrate a widespread problem, and identify the likely culprits.
- We suggest remediation measures to help address the widespread issues identified (§7).

2 Definitions

This section presents common cryptographic definitions for symmetric encryption, password-based encryption, and the respective notions of security. We adopt the notation used by Bellare and Rogaway [6].

Block Ciphers and Symmetric Encryption Schemes A block cipher is a function:

$$E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$$

where k is the key size and n is the block size. We call the input the plaintext, and the output the ciphertext. For each $K \in \{0,1\}^k$, let $E_K : \{0,1\}^n \to \{0,1\}^n$ be defined as $E_K(M) = E(K, M)$. A block cipher $E_k(\cdot)$ is a permutation, with E_k^{-1} as its inverse. Thus, $E_K^{-1}(E_K(M)) = M$ and $E_K(E_K^{-1}(C)) = C$ for all $M, C \in \{0,1\}^n$.

While block ciphers encrypt fixed-length messages, an encryption scheme encrypts messages of arbitrary length. A symmetric encryption scheme $\mathcal{SE}$ is a triple of algorithms $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, where:

- $\mathcal{K}$ is a key generation algorithm producing a key K. We denote picking a key uniformly at random from the key space $KEYS(\mathcal{SE})$ as $K \xleftarrow{\$} \mathcal{K}$.
- An encryption algorithm $\mathcal{E}$, which might be randomized or stateful, takes a plaintext $\{0,1\}^*$, a key K returned by the key generation algorithm, and outputs a ciphertext $C \in \{0,1\}^* \cup \{\perp\}$.
- A deterministic decryption algorithm $\mathcal{D}$ which takes a ciphertext $C \in \{0,1\}^*$, a key K, and outputs $M \in \{0,1\}^* \cup \{\perp\}$. That is, $M \leftarrow \mathcal{D}_K(C)$.
- For correctness, we should be able to decrypt messages:

$$D_k(\mathcal{E}_k(M)) = M$$

We give two examples of encryption schemes built from block ciphers: ECB mode and CBC mode encryption.

Electronic codebook (ECB) mode is a stateless, deterministic encryption scheme defined over a block cipher. The encryption function (ECB) is:

```
$\mathtt{ECB}_K(M)$
    $M[1]\ldots M[m] \leftarrow M$
    for $i \leftarrow 1$ to $m$ do
        $C[i] \leftarrow E_K(M[i])$
    $C \leftarrow C[1]\ldots C[m]$
    return $C$
```

Algorithm 1: ECB Mode

Ciphertext Block Chaining (CBC) is an encryption algorithm built from a block cipher where each block of plaintext is XORed with the previous block of ciphertext. The first block of plaintext is XORed with an initialization vector (IV). As we will see, one insecure way to initialize the IV is by using a constant. A secure version, called CBC$, initializes the IV with a random number upon each invocation of the algorithm, as shown below:

```
$\mathtt{CBC\$}_K(M)$
    $M[1]\ldots M[m] \leftarrow M$
    $C[0] \xleftarrow{\$} \{0,1\}^n$
    for $i \leftarrow 1$ to $m$ do
        $C[i] \leftarrow E_K(M[i] \oplus C[i-1])$
    $C \leftarrow C[0]\ldots C[m]$
    return $C$
```

Algorithm 2: CBC$ Mode

Encryption and IND-CPA Security The goal of an encryption scheme is to provide privacy. Informally, privacy means that an adversary should have a hard time discerning even a single bit of information about the plaintext given the ciphertext. This intuition is formalized in the notion of indistinguishability under a chosen plaintext attack (IND-CPA). We should only consider an encryption scheme to be secure if and only if it is IND-CPA secure.

IND-CPA security can be formalized in a game where:

1. An oracle flips a fair coin $b = \{0,1\}$.
2. The adversary picks a pair of messages of equal length (M_0, M_1). The adversary, who does not have access to the secret key, gives the pair to the encryption oracle
3. The oracle for all encryption calls returns $C_b = \mathcal{E}_K(M_b)$ to the attacker.
4. The attacker executes steps 2 and 3 q times.
5. The attacker outputs a guess b'. The attacker wins if $b' = b$, else the attacker looses.

An encryption scheme is considered IND-CPA secure if the probability that the attacker, after seeing the encryption of q messages, cannot do better than guessing b.

We state as fact a well-known theorem (proven in [6]):

Theorem 1. *An encryption scheme must be either probabilistic or stateful to be indistinguishable under chosen plaintext attacks (IND-CPA).*

For instance, by Theorem 1 ECB mode cannot be IND-CPA secure. In particular, the attacker can learn b using only two queries to the oracle. Let the underlying block cipher length be n. The attacker constructs $M_1 = 0^{2n}$ and $M_0 = 0^n 1^n$. The attacker receives back a $2n$-bit ciphertext consisting of blocks $C[0]$ and $C[1]$. If $C[0] = C[1]$, then message M_1 was encrypted, else message M_0 was encrypted. Thus, the attacker can tell whether $b = 1$ or $b = 0$. CBC$, on the other hand, can be proven IND-CPA secure [6].

Password-based Encryption User-chosen passwords are often vulnerable to dictionary brute-force attacks. Password-based encryption schemes make such brute force attacks more expensive. PBE is defined by RFC 2898 (PKCS#5) [19], where encrypting a message M using a password pw and salt sa is defined as (as described in [5]):

```
PBE$(pw, M)$
    $sa \xleftarrow{\$} \{0,1\}^s$
    $L \leftarrow \mathtt{KD}(pw||sa)$
    return $\mathcal{E}_k(L, M)||sa$
```

Algorithm 3: Password-based encryption

In PBE, $\mathcal{E}$ should be a IND-CPA secure encryption scheme, and KD is the key derivation algorithm. The key derivation algorithm is a c-fold iteration of a cryptographically secure hash function H.

While the c-fold iteration makes brute force attacks more expensive a random salt sa effectively thwarts brute force attacks that rely on pre-computed information, such as rainbow tables. Without any salt, a brute force attack with a dictionary of size N using PBE takes at least an additional cN iterations of H. Assuming $s = |sa|$ is sufficiently large that salts are unique, the complexity rises to scN. RFC 2898 recommends using no less than 1,000 iterations and a 64-bit salt. For example, Apple's iOS Data Protection Layer choses an iteration count so that generating a single key from a password takes roughly 80ms [3]. This delay is hardly noticeable by the user, but significantly slows down brute-forcing attacks.

Abadi and Warinschi [2] provide a computational analysis of password based encryption schemes. Bellare et al. [5] propose a theory of multi-instance security, where they show the key-derivation functions proposed in PKCS#5 and prove that per password salts amplify multi-instance security.

3 Common Rules in Cryptography

While cryptographic security is precisely defined, this paper asks the question whether developers who use cryptographic APIs achieve this notion of security. Using cryptographic primitives correctly can be challenging. In particular, any application that violates one of the following six rules cannot be secure.

Rule 1: Do not use ECB mode for encryption. [6]
Rule 2: Do not use a non-random IV for CBC encryption. [6, 23]
Rule 3: Do not use constant encryption keys.
Rule 4: Do not use constant salts for PBE. [2, 5]
Rule 5: Do not use fewer than 1,000 iterations for PBE. [2, 5]
Rule 6: Do not use static seeds to seed `SecureRandom(·)`.

Rule 1 forbids the use of ECB mode because a symmetric encryption scheme in ECB mode does not provide a general notion of privacy (i.e., it is not IND-CPA secure). Recall that ECB mode is deterministic and not stateful, thus cannot be IND-CPA secure by Theorem 1. A significant problem with ECB mode is that identical messages encrypt to identical ciphertexts. Such a leak of information is often intolerable. One commonly stated exception is that ECB mode is secure if the message is smaller than the underlying block cipher block size *and* all messages are unique. However, even in such cases an IND-CPA secure scheme would also work while providing greater theoretic security, and would thus be a more robust choice.

Rule 2 states that the CBC-mode construction (in Alg. 2) should always use a random IV. In essence, CBC$ should always be used. Unfortunately, it is common to initialize the IV with a constant, e.g., all zeros (i.e., setting line 3 of Algorithm 2 to a constant). A constant IV results in a deterministic, stateless cipher, which by Theorem 1 cannot be IND-CPA secure. One can fix the situation by requiring that the first message block is a random number (essentially taking on the role of a randomized IV). We note that such exceptions to CBC$ are often historically a band-aide patch for implementations that do not follow Rule 2 initially, e.g., as in TLS [23] and SSH [4].

Rule 3 states that an encryption scheme should not use a constant key. Intuitively, a constant key hard-coded in publicly available software is not a secret key, thus the resulting encryption does not provide privacy. Symmetric encryption schemes commonly include a notion of a randomized key generation algorithm $\mathcal{K}$ (see Section 2).

Rule 4 and Rule 5 are both recommended best practices for PBE schemes. Recall from §2 that the iteration count and salt entail a multiplicative increase in work for a brute force dictionary attack. An application that does not follow Rule 4 and uses a constant salt reduces to a program, for cryptographic purposes, with no salt at all. We chose the threshold for the iteration count at 1,000 because this minimum value is suggested in PKCS#5.

Finally, Rule 6 states that the Android `SecureRandom(·)` function should not be seeded with a constant. Android's `SecureRandom` is a *pseudo*-random number generator (PRNG) that is seeded. A PRNG seeded with a constant seed will produce a constant, known output across all implementations. Since PRNG's are often used to create key material, the resulting keys are not random thus not secure. As the name of the API – `SecureRandom` – suggests its use for security relevant tasks, we flag applications that seed the `SecureRandom` PRNG with static values.

4 Crypto in Android

Android applications are authored as Java source code and then compiled to Dalvik bytecode. This bytecode is packaged with additional resources, such as images and configuration files into an application package (apk) file. When the user installs an application from Google Play, the apk file is downloaded and installed on the user's system. Although, the application's source is Java, the Dalvik virtual machine (DVM) considerably differs from the Java virtual machine.

For example, while the Oracle Java virtual machine is stack-based, the DVM is entirely register based. Furthermore, Android provides a rich execution framework. This framework offers access to a variety of subsystems such as graphical user interfaces, networking, or the telephony and messaging sub-systems.

The sub-system relevant to this paper is the Java Cryptography Architecture (JCA). This architecture standardizes how application developers can make use of cryptographic algorithms. To this end, so-called cryptographic service providers (CSP) are registered with the JCA. A CSP is a package providing implementations of cryptographic algorithms, such as message authentication codes, encryption schemes, or key generation algorithms. This modularized architecture enables distributors and developers to seamlessly install and use different CSPs in parallel, or substitute one for the other, as long as they provide implementations for the same algorithms. For example, while Oracle Java contains the SunJCE as the default CSP, Android (since version 2.1) uses BouncyCastle [1] as its default cryptographic service provider.

Block ciphers, symmetric, and asymmetric encryption schemes are accessible to an application through the `Cipher` API. To obtain an instance of a specific encryption scheme, the developer calls the `Cipher.getInstance` factory method and provides a *transformation* as the argument. A transformation is a string that specifies the name of the algorithm, the cipher mode, and padding scheme to use. For example, to obtain a cipher object that uses AES in CBC mode with PKCS5 padding the developer would specify the transformation as: `AES/CBC/PKCS5Padding`. Only the algorithm part is mandatory. The security provider maintains default values for the cipher mode as well as the padding scheme should the developer choose to omit these values. Unfortunately, Java as well as Android chose ECB and PKCS7Padding as default values in case only the AES encryption algorithm is specified. Thus, a developer who only specifies the arguably secure AES block cipher ends up in a potentially insecure situation where the ECB block cipher mode is used.

5 System Design and Implementation

At a high level we observe that the rules specified in Section 3 are temporal properties. We have built an automated analysis tool to evaluate these rules on real-world Android applications. More precisely, we compute static program slices that terminate in calls to cryptographic APIs, and then extract the necessary information from these slices.

In this section we first discuss how we extract the control flow and super control flow graphs from real-world Android applications. We then detail our static program slicing approach, and how we evaluate the rules from Section 3 based on a program slice.

Control flow graphs Our approach targets the Dalvik bytecode of Android applications directly. We build our analysis on top of the Androguard [12] Android application analysis platform. Androguard disassembles an application into classes, methods, basic blocks, and individual instructions.

CryptoLint then first translates this low-level representation into an intermediate representation (IR). In this IR we combine the more than 200 Dalvik instructions into 19 semantically similar *instruction groups* (e.g., arithmetic instructions, invoke instructions). CryptoLint also extracts the intra-procedural control flow graphs for all methods in the application.

An application's use of cryptographic functionality might not be limited to a single method. For example, a cipher object could be instantiated in an object constructor and then used in two different methods (e.g., `encrypt` and `decrypt`) to encrypt or decrypt data respectively. If these two methods are analyzed in isolation, we would not be able to extract the encryption scheme that was used when the cipher object was instantiated. Thus, our approach implements inter-procedural analysis based on the application's super control flow graph to correlate the use of the cipher object for encryption and decryption with the cipher's instantiation. However, before CryptoLint reconstructs the super control flow graph two additional steps are performed.

First, CryptoLint translates all methods into single static assignment (SSA) form as described in [10]. Second, CryptoLint also extracts the class hierarchy of all classes implemented in the Application. Because in Android it is common and often necessary to extend classes that are defined in the Android framework, we also include all classes that are defined by the Android framework into this analysis. For example, any class implementing a user interface component has to extend the class `View` which is defined in the Android framework. This analysis yields the class hierarchy tree rooted at the `Object` class, and contains all inheritance relationships between classes in the application and the Android framework. Furthermore, this data structure also contains information pertaining to Java interfaces and the classes that implement them. In our current implementation CryptoLint targets API version 16 of the Android framework (i.e., Android Jelly Bean). Of course, CryptoLint can be used with any other version of the framework too.

5.1 Extracting the super control flow graph

A super control flow graph (sCFG) consists of the call graph of an application superimposed over the control flow graphs of the individual functions. Call edges are added between call instructions and function entry points, and function exit points are connected with exit edges back to the call site. CryptoLint reconstructs an over-approximation the sCFG of an application by executing the following steps.

First, CryptoLint computes the possible types each register can hold at each program point. Initially, CryptoLint assumes that each register can hold values of any type. CryptoLint then analyzes how registers are used and refines the set of possible types of values accordingly. CryptoLint leverages the static type information that is present in the application's byte code. For example, the types of arguments and return values are listed in the datastructures that describe methods. CryptoLint propagates types to registers that receive arguments or return values from method calls. CryptoLint also leverages additional type information, such as `check-cast` instructions that assure that a register contains (a subtype of) the specified static type. The `new-instance` instruction is used to instantiate a new objects of the given type.

Dalvik bytecode only contains information regarding the static type of the objects used. However, the dynamic type of an object at runtime can be any subtype of the static type. An exception to this rule is the `new-instance` instruction, the dynamic and static type for operands used with a `new-instance` instruction are always identical. Thus, a `new-instance` instruction precisely defines what type of object the operand register contains. This refinement is performed until a fixed-point is reached.

Second, CryptoLint leverages the information from the type refinement step to resolve targets of the `invoke` family of instructions. These instructions consist of `invoke-virtual`, `-super`, `-direct`, `-static`, `-interface`, and their respective `-range` variants. To this end, we follow the approach presented by Dean et al. [11], and combine the information from the class hierarchy analysis with the possible types of the registers to identify the possible targets of invoke instructions.

5.2 Static program slicing

Static program slicing [27] is specified with respect to a slicing criterion. A slicing criterion is defined as a program point p and a variable x. The slicing algorithm then determines all program instructions that might affect the value of x at point p.

The slicing algorithm iteratively traverses the program backwards starting at program point p. During execution, the algorithm keeps a working set of registers whose definitions need to be determined. Initially, this working set contains the registers from the slicing criterion (i.e., x). A register is removed from the working set if the algorithm encounters an instruction that statically defines that register. For an instruction that defines a register in the working set but uses other registers, the algorithm performs the following two steps: First, the defined register is removed from the working set. Second, all registers used by the instruction are added to the working set.

The slicing algorithm terminates successfully, once the working set is empty. This means that all operands in the working set were statically defined. The slicing algorithm terminates unsuccessfully, if it reaches the beginning of a method, but the sCFG does not contain any incoming edges to the currently analyzed method.

Additionally, our implementation of the slicing algorithm is also field sensitive. To this end, we keep a list of fields that are accessed by instructions in the `get` family of instructions (i.e., `iget`, `aget`, `sget`) during the execution of the slicing algorithm. The slicing algorithm is then recursively applied at each program location where the corresponding fields are defined, with the slicing criterion set to the location and the register that defines the field.

5.3 Evaluating security properties

We now describe how CryptoLint evaluates the security rules outlined above on real-world Android applications.

Rule 1: Do not use ECB mode for encryption. An application will use the ECB block cipher mode under one of two conditions. First, if the developer explicitly specifies that she wants to use ECB this is reflected in the transformation string (e.g., `AES/ECB`). The second, and arguably more subtle instance of using ECB, occurs if the developer only specifies a block cipher to use in the transformation. For example, if the developer only specifies `AES` as the transformation, BouncyCastle will automatically choose ECB as the default block cipher mode.

Thus, to identify applications that make use of the ECB block cipher mode, CryptoLint resolves for each call to the `Cipher.getInstance` factory method what transformation string is specified by the developer. To this end, CryptoLint calculates the backwards slice for the slicing criterion consisting of the invoke statement to the factory method and the register that specifies the encryption scheme to be used. CryptoLint raises a warning if either only a block cipher is used as the transformation string or, the transformation explicitly lists a block cipher and the ECB mode.

Rule 2: Do not use a non-random IV for CBC encryption. Block ciphers in feedback mode (e.g., AES/CBC) require an initialization vector. While the CBC$ algorithm (see Algorithm 2) specifies the random selection of a fresh IV for each invocation of the algorithm, the Java API allows the developer to override this random selection and specify an IV herself. If the developer does not specify an IV, the BouncyCastle implementation of CBC will follow the CBC$ algorithm and generate an IV at random using the `SecureRandom` API.

To evaluate this property CryptoLint computes the backwards slice for all calls to the `Cipher.init` method and uses as the slicing criterion the `ParameterSpec` argument of the method call. CryptoLint will flag an application as using constant initialization vectors if the following two conditions hold: (1) The slice includes an object of type `IvParameterSpec`, and (2) all values that are used for the constructor for that `IvParameterSpec` are static. The first condition is necessary because the `ParameterSpec` argument can hold types other than `IvParameterSpec`, such as `PBEParameterSpec` or `DSAParameterSpec`.

The second requirement allows us to identify whether the used IV consists of constant values. That is, if the slice does only depend on constant values, this implies that the IV is constant too.

Although the security prerequisites for IVs in block cipher modes require non-predictable and unique IVs, CryptoLint only identifies the use of static IV values. Static IVs are a subset of predictable IVs (i.e., they are trivially predictable). More precisely, for IVs that are not static, CryptoLint cannot distinguish between predictable and unpredictable IVs. The reason is that a non-static IV might still be predictable due to information that is not available to our analysis.

Rule 3: Do not use constant encryption keys. To identify the use of static symmetric encryption keys, CryptoLint calculates backwards slices for the *key* argument to all invocations of the `SecretKeySpec` constructors. Because the Java security provider API is generic for symmetric and asymmetric encryption, CryptoLint only reports the violation of this property if a static key is used in a symmetric encryption scheme. In an asymmetric encryption scheme it is perfectly legitimate that an encryption key (e.g., the public key of a keypair) is statically included in the application.

Rule 4: Do not use constant salts for PBE. CryptoLint identifies applications that use static salt values for password based encryption by computing the backwards slice for all calls to constructors of the `PBEParameterSpec` and `PBEKeySpec` APIs. The slicing criterion is specified as the call-site of the API call and the register that specifies the salt. If all instructions in the slice exclusively depend on static values, the salt has to be static too, and CryptoLint alerts respectively.

Rule 5: Do not use fewer than 1,000 iterations for PBE. To identify applications that violate this rule, CryptoLint computes a backwards slice for the register that specifies the iteration count at each call to the `PBEParameterSpec` and `PBEKeySpec` constructors. If CryptoLint identifies that the iteration count is below 1,000 this use of password-based encryption is flagged as insecure. We chose this threshold value of 1,000 because RFC 2898 recommends using at least an iteration count of 1,000.

Rule 6: Do not use static seeds to seed `SecureRandom(·)`. CryptoLint flags applications that do seed `SecureRandom` with static values. To this end, CryptoLint computes a backwards slice from all call-sites to the constructor of `SecureRandom` for the seed value specified. We were positively surprised to see that the documentation on `SecureRandom` does include a discussion about its secure usage[1] and possible pitfalls when seeding `SecureRandom`. This is the only crypto related API investigated in this work whose documentation contains such useful security relevant discussions.

To identify applications that do seed `SecureRandom` with static values, CryptoLint computes a backwards slice from all constructors of `SecureRandom` that accept a seed argument. If all instructions in the slice depend exclusively on static values, the seed is considered to be static too.

6 Evaluation

Dataset The goal of this evaluation is twofold. First, we want to demonstrate that CryptoLint is indeed useful to identify violations of the specified rules. Second, by applying CryptoLint on a large number of real-world applications, we gain an insight into the prevalence of the misuse of cryptographic functionality in Android applications.

For this evaluation we downloaded 145,095 applications from the official Google Play marketplace. This dataset was collected between May and July 2012. The security rules that CryptoLint evaluates are related to functionality that resides in the `javax/crypto` and `java/security` name-spaces. Thus, CryptoLint first assesses whether an application makes use of functionality in these name-spaces. 15,134 or 10.4% of all applications in our dataset use crypto functionality.

Name	Description
scoreloop	Cross platform social gaming
vending	Google License verification library
urbanairship	Mobile marketing solutions
openfeint	Social gaming platform
google/ads	Google Advertising
phonegap	Cross platform application development
vpon	Mobile advertising
unity3d	Mobile game engine
apache/james	Internet messaging
microad	Advertising
amazonaws	Libraries for Amazon AWS

Table 1: White-listed libraries

Similar to existing research (e.g., [25]), we observed the pervasive use of third-party libraries for advertisement and statistics purposes. In order to prevent over-counting, we whitelisted common libraries that use cryptography, as listed in Table 1. CryptoLint discards applications if their only use of cryptographic functionality is confined to these libraries. We assume that the applications we analyze do not actively try to disguise the use of these libraries, and thus, identify these libraries by matching their package names.

# apps	violated rule
5,656	Uses ECB (BouncyCastle default) (R1)
3,644	Uses constant symmetric key (R3)
2,000	Uses ECB (Explicit use) (R1)
1,932	Uses constant IV (R2)
1,636	Used iteration count < 1,000 for PBE(R5)
1,629	Seeds SecureRandom with static (R6)
1,574	Uses static salt for PBE (R4)
1,421	No violation

Table 2: Violations of cryptographic security rules

6.1 Results

In total, CryptoLint successfully analyzed 11,748 applications. The analysis of the remaining applications did either not terminate within a timeout of 30 minutes (2,614), or the analysis infrastructure ran out of memory (765). All numbers reported from here on are in reference to the 11,748 successfully analyzed applications.

Table 2 lists the number of distinct applications that violated the rules from §3. Only 1,421 applications in our data set did not violate any of the rules. We discuss the remaining rule violations in order of prevalence below.

Rule 1: Do not use ECB mode for encryption. This was the most frequent rule violated, with 7,656 total apps violating this rule at least once. The primary cause of ECB mode was developers using the default values in the BouncyCastle security provider. More precisely, in 5,656 applications the developer only specifies a block cipher (e.g., `AES`, `DES`, `DESede`) and the BouncyCastle provider configures the resulting cipher in ECB mode. These results indicate that developers are not aware that the API default does not provide the IND-CPA strong notion of privacy.

Additionally, 2,000 applications explicitly request a block cipher in ECB mode from the security provider. While there are legitimate uses for ECB mode (see §3), we manually inspected several applications and verified that they a) were trying to achieve privacy, and b) were using ECB mode on data that was over a block length or non-random. Examples of misuse of ECB mode include:

- One game uses DES/ECB to encrypt personal identifiers. The identifiers exceed the DES block size, thus an adversary can learn which parts of the encrypted ID are the same.
- One anti-virus product encrypts the MD5 hash of viruses found in DES3/ECB mode. An adversary can learn if multiple instances of the same virus are found on the device.
- One password manager stores passwords encrypted using AES in ECB mode. A more detailed discussion of this application will be presented in the case studies (below).

Rule 3: Do not use constant encryption keys. The second most violated cryptographic security property in our dataset is the use of a constant symmetric encryption key. As mentioned in Section 2, the security of symmetric encryption schemes depends on the secrecy of the shared key. Thus, embedding such secret keys into an application negates

[1] Seeding SecureRandom may be insecure at http://developer.android.com/reference/java/security/SecureRandom.html

the security benefits of symmetric encryption. For example, AdMob encrypts the phone's location data using a constant key and sends it over the network. Another example is an application that stores the user's Google credentials on disk encrypted using a static key.

Rule 2: Do not use a non-random IV for CBC encryption. CryptoLint identified 1,932 applications that make use of constant initialization vectors in CBC mode encryption.

Rule 4: Do not use constant salts for PBE. CryptoLint identified 1,574 applications that use a static value for the salt used with the key derivation function in PBE. Using a static salt allows an attacker to pre-compute a dictionary based on the known salt, negating much of the benefit of using a salt at all. While the use of a static salt is better than using the password directly as encryption key, this choice negates the advantages in multi-instance security [5].

Rule 5: Do not use fewer than 1,000 iterations for PBE. The Java `PBEKeySpec` API implements password based encryption based on the PKCS#5 standard. The RFC for PKCS#5 recommends an iteration count of at least 1,000. CryptoLint identified 1,636 applications that use fewer iterations. Applications that use a low iteration count and a static salt for password-based encryption are exposed to trivial dictionary-based off-line attacks, exactly the type of attacks that password-based encryption schemes were designed to protect against.

Applications violating multiple rules We next investigated the number of applications that violate multiple rules. These results are illustrated in Figure 1. Interestingly, it was more common for applications to violate two rules than only violating a single rule. Of the applications violating a single rule, rule 1 was violated the most (3,033 times). 511 applications violated rule 2 and used constant IVs. For 246 applications CryptoLint identified the use of a static symmetric encryption key (violation of rule 3). 29 applications were flagged for using a low iteration count, and 13 applications use static salt values for password-based encryption schemes. CryptoLint identified 6 applications that only violated rule 6 by seeding `SecureRandom` with a static seed.

The numbers of applications that violated exactly two rules are listed in Table 3. Additionally, our dataset contained exactly one application that violated all six rules that CryptoLint evaluates.

# apps	rules violated
1,905	Rule 1 & Rule 3
1,588	Rule 1 & Rule 6
1,247	Rule 4 & Rule 5
866	Rule 2 & Rule 3
109	Rule 1 & Rule 2
24	Rule 1 & Rule 5
11	Rule 3 & Rule 5
5	Rule 2 & Rule 5
2	Rule 1 & Rule 4
2	Rule 3 & Rule 4

Table 3: Applications violating two rules

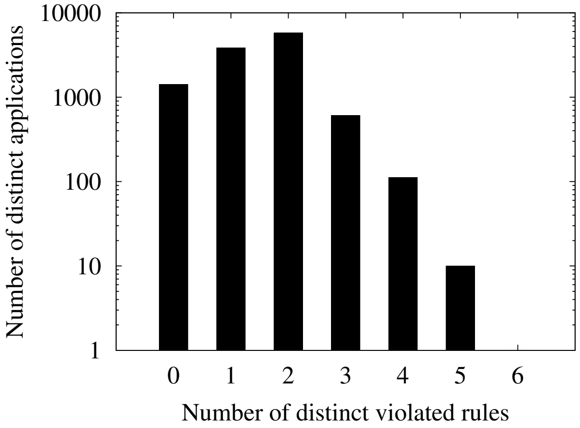

Figure 1: Number of applications violating 0, 1, 2, . . . 6 rules.

6.2 Case Studies

Social gaming platform To estimate the impact of applying cryptographic primitives incorrectly, we manually examined a popular game that CryptoLint reported as misusing crypto. This game is from a development studio that released a series of popular games, all containing a social platform for connecting and interacting with friends. This social platform is used to track high-scores on a leader board. According to Google play, the application we analyzed has between 50,000,000 and 100,000,000 installations. The application communicates with the back-end servers of the social components over http. However, data that is transmitted between the server and the client is encrypted. This application got flagged by CryptoLint for two reasons. First, it uses the DES blockcipher in ECB mode. The developers explicitly specified the ECB block cipher mode as the used transformation string is `DES/ECB`). Furthermore, CryptoLint also complains that the application uses a static key with this encryption scheme. We evaluated the correctness of these results by interacting with the game and exercising the social network functionality while at the same time recording all network traffic sent by the application. With the key material retrieved by CryptoLint, it was trivially possible to decrypt the encrypted network traffic.

Bookmark Manager We also investigated a bookmark manager application in more detail (install base between 1,000,000 and 5,000,000). This application allows the user to synchronize bookmarks between different browsers installed on the mobile device. Furthermore, it provides the functionality to synchronize browser bookmarks with Google's web services. To make use of this functionality, the user has to provide her Google credentials. The application stores these credentials in a regular Java property file. While the Google user-name is stored in the clear, the password is encrypted. CryptoLint flagged this application because it uses the DES blockcipher in ECB mode to store that information in the property file. Furthermore, the application also uses a constant key for the encryption. Again we verified that decryption of the password is trivially possible. We agree that safe storage of access credentials is challenging to get right. To this end, Android provides a `KeyStore` facility that is designed for exactly the purpose of storing access credentials and is accessible through the API.

Password management application Users entrust their passwords to password manager applications for safe keeping and easy management. Because password information is important to protect securely, we investigated one application in this category closer. Although this application only has between 100,000 and 500,000 installations, the fact that the application is open source with a publicly available GIT repository warrants a closer analysis. In the earliest versions of the application the developer used the AES block cipher in ECB mode. However, before encryption, the application prepends two bytes of random data to the password. After decryption the initial two bytes are discarded. Furthermore, the application derives the key by calculating an HMAC over the master password the user supplies.

Several design decisions reduce the security of this implementation and render it non IND-CPA secure. Although prepending the password with two random bytes prevents two identical passwords from being encrypted to identical ciphertexts, this measure only protects the first 14 bytes of a password. Because individual blocks are encrypted independently, all plaintext blocks after the initial 14 bytes would be encrypted to the same ciphertext blocks. Furthermore, the developer chose to use a single HMAC operation with a static key to derive key material from the master password. Instead, the author should have used existing password-based encryption schemes to protect the key database against dictionary attacks.

In a subsequent version the author substituted the ECB mode for an encryption scheme based on AES/CBC. However, the author also hard-coded a static IV into the application. Similar to before, the author prepended the password to store in the database with two random bytes of data before performing the encryption. While the use of CBC and the two random bytes constitute a significant security improvement over earlier versions of the application, the application is still not IND-CPA secure. The reason is that two random bytes at the beginning of the plain text is not enough to preserve the IND-CPA security of the CBC$ algorithm. More precisely, two passwords are encrypted into the same cipher texts with probability of $1/2^{16} = 1/65536$, which is considered non-negligible.

Finally, in more recent versions the author relies on AES/CBC and generates IVs at random. However, the author uses the regular random number generator instead of the `SecureRandom` API which should be used in cryptographic contexts.

This development history spans two years of development on a system that is arguably designed to keep personal data secure. Our analysis shows that it is non-trivial even for well-intended developers to apply cryptographic primitives correctly. Thus, we propose a series of mitigations in Section 7 to make it easier for developers to use cryptographic algorithms correctly.

Popular libraries In the following we discuss our findings regarding how popular libraries apply cryptographic algorithms.

AdMob[2]**.** Google's AdMob advertising library is one of the most popular libraries included in Android applications. In fact we found that 36 % of the applications in our dataset make use of this library. AdMob uses the AES block cipher in CBC$ mode to encrypt device location and identifiers before transmitting that information to the adserver. This library correctly uses the default behavior of the BouncyCastle provider to generate a random IV through the `SecureRandom` API. Thus, AdMob makes correct use of the cryptographic functionality provided in Android. However, AdMob also uses a constant encryption key for this operation. Thus, the security provided by the symmetric encryption scheme is undermined.

Scoreloop[3]**.** The Scoreloop library provides functionality to integrate social capabilities to mobile applications. The platform allows the developer to add virtual currencies and game items to her application and supports multiple payment options. When analyzing an example application that makes use of the Scoreloop library, CryptoLint correctly alerts that the library is using AES/CBC with constant initialization vectors. More precisely, the library derives the used IVs deterministically from the hard-coded URL endpoints of the Scoreloop backend servers. This result illustrates that not all developers of high profile libraries are capable of using cryptography correctly.

Android License Verification Library[4]**.** The LVL provides the developer with the necessary functionality to enforce a licensing policy on her applications. The documentation states that to keep licensing information persistent, this information has to be stored in an obfuscated manner on the device. To this end, the LVL provides an obfuscation scheme that is based on the AES block cipher in CBC mode. This default implementation however uses a constant initialization vector[5]. As the name of this API (i.e., `AESObfuscator`) suggests, the intended purpose here is to obfuscate instead of achieving IND-CPA security. However, modifying the source code to at least choose an IV at random would be straight forward, as the AdMob library illustrates.

6.3 Limitations

Our current system has a number of limitations. For example, Android applications can make use of native code. However, CryptoLint currently only targets Dalvik bytecode. Therefore, applications that invoke cryptographic primitives from native code cannot be analyzed. Furthermore, the property that initialization vectors have to be unique is not globally valid. For example, the Kerberos protocol uses a static initialization vector with the CBC block cipher mode. However, the protocol also specifies that the first block of a message is filled with random data and discarded at decryption. This basically transforms the first block into fulfilling the role of the initialization vector instead. Thus, the IV does not need to be transmitted along with cipher-text message. Inferring this valid use of a static IV in CBC mode would require CryptoLint to infer the implicit knowledge of the protocol designer which is clearly beyond the scope of this work.

CryptoLint only detects the use and misuse of cryptographic primitives if they are properly exposed through the intended interfaces (e.g., security providers, ciphers, and

[2] http://www.google.com/ads/admob/

[3] http://www.scoreloop.com/

[4] http://developer.android.com/google/play/licensing/index.html

[5] The source for the LVL and the AESObfuscator can be obtained at: http://code.google.com/p/marketlicensing/source/browse/library/src/com/android/vending/licensing/AESObfuscator.java

#Occurences	Symmetric encryption scheme
5878	AES/CBC/PKCS5Padding
4803	AES *
1151	DES/ECB/NoPadding
741	DES *
501	DESede *
473	DESede/ECB/PKCS5Padding
468	AES/CBC/NoPadding
443	AES/ECB/PKCS5Padding
235	AES/CBC/PKCS7Padding
221	DES/ECB/PKCS5Padding
220	AES/ECB/NoPadding
205	DES/CBC/PKCS5Padding
155	AES/ECB/PKCS7Padding
104	AES/CFB8/NoPadding

Table 4: Distribution of frequently used symmetric encryption schemes. Schemes marked with * are used in ECB mode by default.

MACs). CryptoLint cannot reason about applications that implement cryptographic primitives ad-hoc. To do so, CryptoLint would have to infer whether a particular piece of code implements cryptographic functionality. This is outside the scope of this work. Furthermore, history has shown that developers are rarely better off rolling their own cryptographic implementations as opposed to using well-tested library functionality. Thus, any such implementation is most likely less secure than the OS provided cryptographic algorithms.

Recent events demonstrated that the assumption that cryptographic primitives are implemented correctly can be violated [20]. However, CryptoLint's focus is to identify applications that use these primitives incorrectly and not the identification of flawed implementations of the primitives themselves.

7 Mitigations

We now discuss a set of possible countermeasures that would likely reduce the prevalence of misused cryptographic primitives in Android applications. As explained in the introduction three main issues cause the problems we see with applying cryptography in Android applications. (1) APIs are not expressive enough to enforce semantic contracts (e.g., IVs should be unique and non-predictable). (2) APIs ship with poor default configurations, and (3) the documentation insufficiently describes the APIs.

Semantic contracts in APIs . One approach is to use tools such as CryptoLint to vet software, e.g., as part of the Google Play marketplace. Additionally, compilers can provide safety warnings on typically insecure method calls to the crypto API. For example, a call to the ECB encryption mode could raise a warning similar to the way that the `strcpy` function is flagged by the Microsoft C compiler.

Poor default configurations in APIs . Switching default configurations in APIs is challenging, especially in the light of backwards compatibility. However, because of the negative characteristics of today's default values, we believe that choosing better defaults would mitigate many problems that lead to misused cryptography.

Table 4 lists the symmetric encryption schemes used by at least 100 applications in our dataset. The most popular API call is to CBC mode encryption, where CBC mode is explicitly picked in the name. The second (AES), fourth (DES), and fifth most popular (DESede) do not indicate which mode is being used. One possibility is to ban APIs that do not make the encryption mode explicit. Such an approach would require developers to investigate an appropriate encryption mode. Paired with appropriate documentation, this change would potentially make more developers aware of the cryptographic issues associated with block cipher encryption modes.

API documentation. The Java and Android API documentation contains a disclaimer that it is not designed to teach a developer the prerequisites of cryptography. However, the documentation could suggest sane defaults, e.g., CBC mode with a random IV because it is secure. Furthermore, we strongly advocate that the documentation for the cryptographic security provider explicitly state the default values. In some cases the default value is not mentioned at all. For example, the `Cipher` class states that a block cipher mode can be requested. However, it fails to mention if no mode is requested, ECB mode will be used by default. As it stands, the only way for a developer to determine default values, and whether they are secure, is via trial and error, Internet searches, or the inspection of the source code of the security provider.

Although Google regularly pulls up-to-date BouncyCastle revisions into the Android source tree, not all enhancements within the security provider are exposed to applications via the SDK. For example, BouncyCastle has supported Galois counter mode (GCM [6]) for authenticated encryption with associated data since 2008. Oracle Java has supported this mode and the necessary APIs since version 1.7 which was released in 2011. However, the latest Android version (at the time of writing Jelly Bean 4.3) does not expose the necessary APIs to use associated data with any authenticated encryption modes. Furthermore, the Android documentation does not mention authenticated encryption at all. Thus, developers who want to use these encryption modes have to gather their knowledge from other resources.

8 Related Work

The popularity of the Android operating system has attracted the attention of many researchers in the past. With TaintDroid, Enck et al. [13] track the propagation of sensitive information through Android applications. Hoffmann et al. [17] present SAAF, a static analysis framework that helps a human analyst to examine Android applications. Furthermore, the Android permission system has been at the core of many scientific publications [14, 16, 24]. While the permission system is the first line of defense in the Android security landscape, cryptographic primitives allow developers to add another line of defense by encrypting data before storing or transmitting it. Thus, to the best of our knowledge, CryptoLint is the first approach that investigates whether application developers make correct (i.e., secure) use of cryptographic primitives.

Zhou [29] and Vidas [26] investigate malicious applications in the Android ecosystem. The focus of CryptoLint, however, is to identify benign applications that employ cryptographic primitives incorrectly.

Our work is similar to the Lint program checker [18]. That is, we identify a series of common programming mistakes

[6] Within the Android documentation GCM is used to refer to the Google Cloud Messaging API.

and automatically identify applications that contain such mistakes. Similarly, LCLint [21] uses source-code analysis and manual annotations to identify likely buffer overflows in C programs. The main difference between these approaches and CryptoLint is, however, that CryptoLint does not have access to source code and operates on compiled Android applications instead. To ensure wide applicability of CryptoLint, we have to operate on compiled Android applications instead. Chen and Wagner presented MOPS [8] to examine security properties of software at compile time. CryptoLint is similar to MOPS as its goal is also to evaluate security properties. However, CryptoLint operates on compiled applications. Furthermore, a major contribution of this work is the broad overview that we gained about the prevalence of misused cryptographic functionality in Android applications. Fahl et al. [15] presented MaloDroid, a system that identifies Android applications that do not perform the necessary validation on SSL certificates. MaloDroid is similar to CryptoLint as it targets Android applications. However, the single property that is evaluated by MaloDroid is whether applications adequately verify SSL certificates. CryptoLint checks for properties that are generally applicable to a cryptographic context, such as the proper use of initialization vectors, or random salt values for password based encryption schemes.

Mitchell et al. [22] present Murφ a tool that allows them to detect vulnerabilities in cryptographic and security-relevant protocols through state enumeration. Bhargavan et al. [7] illustrate how verification tools can be used to show the security of cryptographic protocol implementations. The verification of the TLS1.0 protocol required the authors to write an implementation from scratch that makes itself amenable to the suggested verification techniques. While such an approach is desirable, it seems unreasonable to require the large population of Android application developers to adapt such programming standards. Similar to these two approaches, we treat encryption primitives as black boxes. That is, we trust their security and implementations. However, developers using cryptographic primitives rarely implement well known protocols themselves.

Similar to Whitten et al. [28] and Clark and Goodspeed [9] we analyze the usability of cryptography. However, while the mentioned works analyze the usability for end-users, CryptoLint focuses on the usability of cryptographic APIs, and functionality for application developers. However, our findings are somewhat comparable in that end-users as well as developers seem to lack the proper knowledge or support to make correct decisions when applying cryptography.

9 Conclusions and Future work

CryptoLint checks real-world Android applications for the violation of the six security rules outlined in Section 3. With this automated approach we identified 10,327 applications (88% of our dataset) that violate at least one of these rules. We identified one of the contributing factors to be the undocumented insecure default configuration of the BouncyCastle cryptographic security provider used on the Android platform. Based on the insights we gained from the large-scale analysis of real-world Android applications, we also illustrated different mitigation approaches, we believe would be beneficial to the overall security of the Android ecosystem.

We are currently working on making CryptoLint a publicly accessible online service where developers and curious users can submit Android applications and have them evaluated with respect to the cryptographic security rules described herein. In the future we also plan to extend CryptoLint with security rules that capture the misuse of asymmetric cryptography.

Acknowledgements

This material is based on research sponsored by DARPA under CSSP #23143.2.1080246 and agreement number FA8750-12- 2-0101. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] The legion of the bouncy castle. http://bouncycastle.org/, 2013.

[2] M. Abadi and B. Warinschi. Password-Based Encryption Analyzed. In *Proceedings of the international colloquium of Automata, Languages and Programming*, pages 664–676. Springer, 2005.

[3] I. Apple. iOS Security Contents, 2012.

[4] M. Bellare, T. Kohno, and C. Namprempre. Authenticated encryption in SSH: Provably Fixing the SSH Binary Packet Protocol. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 1–11, 2002.

[5] M. Bellare, T. Ristenpart, and S. Tessaro. Multi-instance Security and Its Application to Password-Based Cryptography. In *Proceedings of the 32nd Annual Cryptology Conference*, pages 312–329. Springer, 2012.

[6] M. Bellare and P. Rogaway. Course notes for introduction to modern cryptography. cseweb.ucsd.edu/users/mihir/cse207/classnotes.html.

[7] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu. Cryptographically verified implementations for TLS. In *Proceedings of the 15th ACM conference on computer and Communications security*, pages 459–468, 2008.

[8] H. Chen and D. Wagner. MOPS: An Infrastructure for Examining Security Properties of Software. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244, 2002.

[9] S. Clark and T. Goodspeed. Why (special agent) Johnny (still) can't encrypt: a security analysis of the APCO project 25 two-way radio system. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[11] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101. Springer, 1995.

[12] A. Desnos. Androguard: Reverse engineering, malware and goodware analysis of android applications ... and more (ninja !). http://code.google.com/p/androguard/.

[13] 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 *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[14] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on computer and Communications security*, pages 235–245, 2009.

[15] 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 *Proceedings of the 19th ACM conference on Computer and communications security*, pages 50–61, 2012.

[16] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638, 2011.

[17] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth. Slicing droids: program slicing for smali code. In *In Proceedings of the 28th ACM Symposium on Applied Computing*, 2013.

[18] S. C. Johnson. Lint , a C Program Checker. Technical report, 1978.

[19] B. Kaliski. PKCS #5: Password-based cryptography specification version 2.0. http://tools.ietf.org/html/rfc2898.

[20] A. Klyubin. Some SecureRandom thoughts. http://android-developers.blogspot.co.uk/2013/08/some-securerandom-thoughts.html, 2013.

[21] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 177–190, 2001.

[22] J. C. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Murphi. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 141–151, 1997.

[23] B. Moeller. TLS insecurity (attack on CBC). http://www.openssl.org/~bodo/tls-cbc.txt, 2001.

[24] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332, 2010.

[25] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012.

[26] T. Vidas, D. Votipka, and N. Christin. All Your Droid Are Belong To Us: A Survey of Current Android Attacks. In *Proceedings of the 5th USENIX Workshop on Offensive Technologies*, 2011.

[27] M. Weiser. Program Slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449, 1981.

[28] A. Whitten and J. Tygar. Why Johnny Can't Encrypt : A Usability Evaluation of PGP 5.0. In *Proceedings of the 8th USENIX Security Symposium*, 1999.

[29] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012.