Cardpliance: PCI DSS Compliance of Android Applications

Samin Yaseer Mahmud,* Akhil Acharya,* Benjamin Andow,† William Enck,* and Brad Reaves*
*North Carolina State University
†IBM T.J. Watson Research Center

Abstract

Smartphones and their applications have become a predominant way of computing, and it is only natural that they have become an important part of financial transaction technology. However, applications asking users to enter credit card numbers have been largely overlooked by prior studies, which frequently report pervasive security and privacy concerns in the general mobile application ecosystem. Such applications are particularly security-sensitive, and they are subject to the Payment Card Industry Data Security Standard (PCI DSS). In this paper, we design a tool called *Cardpliance*, which bridges the semantics of the graphical user interface with static program analysis to capture relevant requirements from PCI DSS. We use Cardpliance to study 358 popular applications from the Google Play that ask the user to enter a credit card number. Overall, we found that *1.67%* of the 358 applications are not compliant with PCI DSS, with vulnerabilities including improperly storing credit card numbers and card verification codes. These findings paint a largely positive picture of the state of PCI DSS compliance of popular Android applications.

1 Introduction

Mobile devices have become a primary way for users to access technology, and for many users, it is the *only* way. The most wide-spread mobile device platforms, namely Android and iOS, are known for their vast application stores providing applications that offer a wide variety of functionality. An important subset of these applications takes payment information from consumers, including those providing entertainment, transportation, and food-related services.

The casual observer might expect that mobile apps offering paid services and goods will always leverage the established and centralized payment platforms provided by the mobile OS (e.g., Google Pay and Apple Pay). These payment platforms provide users a secure and trusted way to manage their payment information (e.g., credit card numbers) without unnecessarily exposing it to third parties. They do so by a) using a virtual token that is linked to the actual credit card, and b) handling both payment information and authorization outside of the third-party application [3]. However, recent work [8] reported that 4,433 of a random sample of 50,162 applications from the Google Play were asking the user to enter credit card information via text fields in the application UI. There are many reasons why this may occur. For example, an application developer may wish to offer an alternative if the user does not want to use the Google or Apple payment system. Alternatively, the application developer may wish to avoid overhead charges from Google and Apple [39, 40]. Whatever the reason, the fact remains: applications are asking users to enter credit card information.

The use of payment information makes these applications distinct from the majority of mobile applications. Specifically, the PCI DSS [6] financial industry-standard mandates that software systems protect payment information in specific ways. While it is well known that mobile applications leak privacy-sensitive information [9, 15, 16, 23], fail to verify SSL certificates [17, 18, 21, 26, 31, 38], and misuse cryptographic primitives [14, 27], doing so while processing payment information represents a significant violation.

Our work is motivated by the research question: *do mobile applications mishandle payment information?* Answering this question introduces several technical research challenges. First, which PCI DSS requirements apply to mobile applications? PCI DSS v3.2.1 (May 2018) is 139 pages and applies to a broad variety of payment systems. Second, how can those requirements be translated into static program analysis tasks? The analysis should avoid false negatives while minimizing false positives. Third, how can the use of credit card information be programmatically identified? Distinguishing credit card text values requires understanding the semantics of widgets in the user interface.

In this paper, we design a static program analysis tool called *Cardpliance* that captures key requirements from PCI DSS that are applicable to mobile applications. Cardpliance combines recent work on static program analysis of Android applications (i.e., Amandroid [19]) and UI semantic inference

(i.e., UiRef [8]) to create novel checks for PCI DSS requirements. We use Cardpliance to study a set of 17,500 popular free applications selected across all categories of Google Play. Using the UI semantic inference of UiRef [8], Cardpliance reduces this sample to 358 applications known to ask for credit card information from the user. Cardpliance then identifies 40 applications with potential PCI DSS violations. After manual decompilation and source code review, we confirmed 6 non-compliant applications.

Broadly, our empirical study leads to the following takeaways. Overall, *98.32%* of the 358 Android applications that we analyzed passed Cardpliance's PCI DSS tests, which shows that the risk of financial loss due to insecure behaviors in mobile applications may not be as wide-spread as predicted. In particular, we did not find any evidence of applications sending payment information over the network in plaintext, over vulnerable SSL connections, or insecurely exposing the payment information via inter-component communication channels. However, we identified 6 applications that combined have nearly *1.5* million downloads on Google Play violating PCI DSS requirements by storing or logging credit card numbers in plaintext (5/6), persisting credit card verification codes (3/6), and not masking credit card numbers when displaying (2/6). These applications are placing the users and potentially their customers at unnecessary risk for fraud due to their non-complying behaviors.

This paper makes the following contributions:

- *We encode PCI DSS requirements for mobile applications into static program analysis rules.* These rules are largely captured using data flow analysis, but the existence of method calls on the corresponding control flow paths play a key role.
- *We study a set of 358 applications known to prompt the user for credit card information.* We find 6 applications that violate PCI DSS requirements.
- *We propose a set of best practices for mobile application developers processing payment information.* These suggestions distill hundreds of pages to PCI DSS standards specification into key areas relevant to mobile apps.

We note that an entire industry of products exists to enable developers to identify individual PCI DSS violations in their own code [9, 19, 22, 25, 30]. By contrast, Cardpliance is to our knowledge the first system to identify violations across *a significant portion of an entire industry* with no prior knowledge of which apps might even handle credit card information. In addition to helping Android application developers aware of unintentional PCI DSS violations, Cardpliance can also be used by Google to triage and investigate flaws in applications as they are submitted to the Play Store. Google could also show the output of Cardpliance in the Play Store's developer console.

The remainder of this paper proceeds as follows: Section 2 describes relevant security requirements from PCI DSS. Section 3 overviews our approach to testing compliance with these requirements. Section 4 describes the design and implementation of Cardpliance. Section 5 uses Cardpliance to study popular applications accepting credit card information. Section 5.7 discusses threats to validity. Section 6 presents a set of best practices for mobile application developers processing payment information. Section 7 describes related work. Section 8 concludes.

2 PCI Data Security Standard

In the early 2000s, major credit card companies faced a crisis of payment fraud that was enabled by the widespread adoption of online financial transactions. As a result, the Payment Card Industry (PCI) released the first version of its Data Security Standard (DSS) in December 2004. PCI DSS [6] now regulates all financial systems seeking to do business with PCI members, which includes all major credit card companies. This standard applies to all computing systems that accept card payment, as well as those that store and process sensitive cardholder data. It defines a series of security measures that must be taken for such systems, including the use of firewalls and anti-virus software.

Not all PCI DSS security measures apply to mobile applications installed on consumer devices. Based on our expertise in mobile application security, we systematically reviewed the 139 pages of PCI DSS version 3.2.1 to determine which regulations apply. For example, mobile applications are payment terminals where a consumer may enter a credit card information into either their own device or the device of a merchant. In contrast, mobile applications are not used as back-end payment processing systems. We then looked for the different types of sensitive information referenced within the standard. We found that PCI DSS distinguishes between cardholder data (CHD) and sensitive account data (SAD), which impacts software processing, as shown in Table 1.

Next, we reviewed the standard for requirements relating to mobile applications. We identified the following six relevant PCI DSS requirements:

Requirement 1 (Limit CHD storage and retention time): PCI DSS Section 3.1 states:

> *Limit cardholder data storage and retention time to that which is required for business, legal, and/or regulatory purposes, as documented in your data retention policy. Purge unnecessarily stored data at least quarterly.*

Therefore, mobile applications should minimize the situations when the credit card number and other CHD values are written to persistent storage. Ideally, CHD is never written, but if it is, the applications need a method to remove it. CHD should also never be written to shared storage locations, e.g., SDcard in Android, as it may be read by other applications.

Table 1: Types of payment information relevant to credit cards

Information	Type	Storage Permitted	Description
PAN	CHD	Yes	Primary Account Number, 16 digits, on front of card.
Cardholder Name	CHD	Yes	Cardholder's name, on front of card
Expiry Date	CHD	Yes	Card expiration date, displayed as MM/YY
Service Code	CHD	Yes	3 digit code, each digit has own service code assignment
Full Track Data	SAD	No	Sensitive data stored on magnetic strip or on a chip
CAV2, CVC2, CVV2, CID	SAD	No	Three or four digit code on back of card
PIN	SAD	No	Pass code that verifies the user during transactions

CHD = Card Holder Data; SAD = Sensitive Account Data

Applications also do not have the ability to delete contents written to Android's logcat logging infrastructure.

Requirement 2 (Restrict SAD storage): PCI DSS Section 3.2 states:

> *Do not store sensitive authentication data after authorization (even if encrypted). If sensitive authentication data is received, render all data unrecoverable upon completion of the authorization process.*

Therefore, SAD values such as full track data (magnetic-stripe data or equivalent on a chip), card security codes (e.g., CAV2/CVC2/CVV2/CID), PINs and PIN blocks should never be written to persistent storage, even if it is encrypted or in a location only accessible to the application.

The standard states that data sources such as incoming transaction data, logs, history files, trace files, database schemes, and database contents should not contain SAD. While we expect few mobile applications ask for full track data, subsets of SAD are relevant. Furthermore, mobile applications should be careful not to include SAD in debugging logs and crash dumps.

Requirement 3 (Mask PAN when displaying): PCI DSS Section 3.3 states:

> *Mask PAN when displayed (the first six and last four digits are the maximum number of digits you may display), so that only authorized people with a legitimate business need can see more than the first six/last four digits of the PAN. This does not supersede stricter requirements that may be in place for displays of cardholder data, such as on a point-of-sale receipt.*

The standard warns that the display of the full PAN on computer screens, mobile UI, payment card receipts, faxes, or paper reports can aid unauthorized individuals in performing unwanted activities. Therefore, after the user enters the credit card number, the application should mask it before displaying (e.g., on a subsequent UI screen).

Requirement 4 (Protect PAN when Storing): PCI DSS Section 3.4 states:

> *Render PAN unreadable anywhere it is stored – including on portable digital media, backup media, in logs, and data received from or stored by wireless networks. Technology solutions for this requirement may include strong one-way hash functions of the entire PAN, truncation, index tokens with securely stored pads, or strong cryptography.*

This requirement supplements Requirement 1 with restrictions specifically for the credit card number (PAN). If it is written at all, some sort of protection is required.

Requirement 5 (Use secure communication): PCI DSS Section 4.1 states:

> *Use strong cryptography and security protocols to safeguard sensitive cardholder data during transmission over open, public networks (e.g. Internet, wireless technologies, cellular technologies, General Packet Radio Service [GPRS], satellite communications). Ensure wireless networks transmitting cardholder data or connected to the cardholder data environment use industry best practices to implement strong encryption for authentication and transmission.*

From the perspective of mobile applications, all network connections should use TLS/SSL. Furthermore, the application should not remove the server authentication checks, which prior work [17] has identified is a common vulnerability in mobile applications.

Requirement 6 (Secure transmission of PAN through messaging technologies): PCI DSS Section 4.2 states:

> *Never send unprotected PANs by end-user messaging technologies (for example, e-mail, instant messaging, SMS, chat, etc.).*

Again, specific additional restrictions are made for the credit card number (PAN). That is, mobile applications should not pass the PAN to APIs for sending SMS messages. Additionally, Android allows sharing data with other messaging applications using its Intent message-based inter-component communication (ICC). Such messages should be protected.

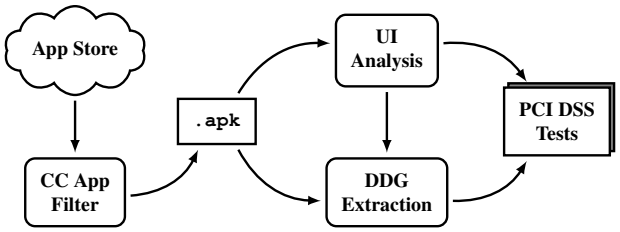

Figure 1: Overview of Cardpliance

3 Overview

While many studies have investigated vulnerabilities in mobile applications, we are unaware of studies focused on credit card information. Such vulnerabilities represent PCI DSS violations and hence are of significant importance. However, programmatically investigating the relevant PCI DSS requirements is nontrivial, presenting the following key challenges.

- *Credit card information is often collected via text input.* There is no clearly-defined API that identifies when the user enters a credit card number. These inputs must be identified and linked to control and data flow graphs.
- *The relevant PCI DSS requirements are context-sensitive.* Simple data-flow analysis is insufficient. For example, some types of credit card information can be stored or transmitted if it is obfuscated.
- *The relevant PCI DSS requirements are imprecise.* The requirements often refer to broad approaches to information protection such as rending the PAN "unreadable." There are many ways in which developers can achieve these goals.

Cardpliance addresses these challenges using a collection of tailored static program analysis tests. Where possible, we leverage existing open source projects that embody knowledge gained from a decade of mobile application analysis. Specifically, we build upon UiRef [8] to infer the semantics of text input and Amandroid [19] (also called Argus-SAF) to perform static data flow analysis. Our analysis also leverages concepts from MalloDroid [17] to identify SSL vulnerabilities and StringDroid [41] for identifying the URL string used to make network connections. Combining these existing techniques to create specific PCI DSS checks requires careful construction and represents a unique contribution.

Figure 1 provides a high-level overview of Cardpliance's approach to identifying PCI DSS violations in mobile applications. The first step is to identify which applications ask users to enter credit card information. While we build upon UiRef for user interface analysis, the analysis requires injecting a code executing the repackaged application. This process is too heavyweight for application discovery. Therefore, we use a two-phase application filter, first using a lightweight keyword-based search of the strings used by the application, then using UiRef to confirm that the application actually asks the user to enter credit card information (e.g., the terms could have been used in some other context).

The next phase is the Data Dependence Graph (DDG) extraction. A key feature of Amandroid is to produce graphs upon which different static analysis tasks can be performed. This approach encapsulates traditional static program analysis within the core Amandroid tool and allows users of Amandroid to focus on their goals as graph traversal algorithms. However, we found that the latest version of Amandroid did not include all of the program contexts that were needed for our PCI DSS tests. First, we use information from UiRef to annotate UI input widgets as being related to credit card information. Second, we enhance how Amandroid handles `OnClickListener` callbacks to correctly track data flows from UI input.

The six PCI DSS tests capture the relevant requirements described in Section 2. Described in detail in Section 4, these tests consider the different uses of cardholder data (CHD) and sensitive account data (SAD) listed in Table 1. Each test defines sets of sources and sinks for Amandroid's taint analysis; however, the tests require context beyond traditional taint analysis. First, Amandroid uses method signatures as sources and sinks, whereas Cardpliance only considers a subset of method calls that are parameterized with specific concrete values (e.g., UI widget references from UIRef). Second, three of the six tests are designed to not raise an alarm if *all* paths from a specific source to a specific sink invoke a method that makes the data flow acceptable (e.g., masking or obfuscating the credit card number). Therefore, Cardpliance includes additional traversal of the DDG.

Finally, due to the imprecision of PCI DSS, several of the tests are inherently heuristic. In such cases, we erred on the side of being security conservative, preferring false positives over false negatives and invalidating the false positives through manual inspection. Therefore, Cardpliance serves as a tool to drastically reduce the effort of a manual auditor, providing key information necessary to make a certification determination. Section 5 describes our experiences manually reviewing flagged applications with the JEB decompiler. Note that we did not perform dynamic analysis of the flagged applications because many of them required social security numbers to register for accounts or for us to be in a physical location to test (e.g., road toll applications).

4 Cardpliance

Android application analysis is a well-studied problem. Open-source analysis tools such as FlowDroid [9], Amandroid [19], and DroidSafe [22] capture much of Android's runtime complexity, including application lifecycles and callbacks from code executing system processes. We chose to build on top

of Amandroid, also called Argus-SAF,[1] because it a) is being actively maintained, b) has a design that is easy to extend, and c) outputs convenient graphs for use by novel analysis. This section is split into two parts: First, we explain key concepts in Amandroid and how we configured it for our analysis. Second, we describe our tests that capture the relevant PCI DSS requirements described in Section 2. This second part captures a key technical contribution of this paper.

4.1 DDG Extraction

The Cardpliance tests are graph queries on Amandroid's Data Dependence Graph (DDG). Amandroid performs flow- and context-sensitive static program analysis on `.apk` files. It analyzes each Android component (e.g., Activity component) separately and then combines the per-component analysis to handle inter-component communication (ICC). As such, program analysis timeouts are defined at the component level (as we discuss in Section 5, we use a timeout of 60 minutes).

Amandroid is primarily focused on data flow analysis. It calculates points-to information for each instruction in the control flow graph, storing it in a Points-to Analysis Results (PTAResult) hash map. It also keeps track of ICC invocations in a summary table (ST). Amandroid then produces an Interprocedural Data Flow Graph (IDFG) for each component, which combines the Interprocedural Control Flow Graph (ICFG) with the PTAResult for that component. It then generates an Interprocedural Data Dependency Graph (IDDG), which contains the same nodes as the IDFG, but the edges are the dependencies between each object's definition to its use. Finally, a DDG for the entire application is created by combining each component's IDDG and the ST.

Amandroid uses the DDG to perform taint analysis. Given a set of taint sources and taint sinks, Amandroid marks the sources and sinks in the DDG and computes the set of all paths between them. The list of paths from sources to sinks is stored in a Taint Analysis Result (TAR) structure. Amandroid allows the user to define sources and sinks via text strings of method signatures in a configuration file.

Cardpliance analyzes how applications handle credit card information entered by the user into text fields. Applications access this text via the `TextView.getText()` method. However, Cardpliance needs to determine which `TextView` objects correspond to the UI widgets that collect different types of credit card information. To acquire a `TextView` object, the application calls `Activity.findViewById(R.id.widget_name)`, where `R.id.widget_name` is a unique integer managed by the application's resource `R` class. Therefore, Cardpliance uses `Activity.findViewById(int)` as a taint source. The analysis will taint the returned `TextView` and the subsequent string from `TextView.getText()`. Furthermore, since the DDG contains points-to information, the PCI DSS tests can use Amandroid's `ExplicitValueFinder.findExplicitLiteralForArgs()` method to determine the integer value passed to the taint source. It then uses the resource IDs of credit card information widgets identified by UiRef [8] to determine the types of information flowing to each sink.

However, applications frequently call `Activity.findViewById()` to assess many different UI widgets. Therefore, simply defining it as a taint source will cause Amandroid's taint analysis to needlessly compute taint paths for many irrelevant sources. To address this problem, Cardpliance implements a custom source and sink manager that refines the taint sources to just those `Activity.findViewById(int)` instructions that are passed an integer in a list precomputed by UiRef. This process involves using the PTAResult hash map while marking taint sources. In doing so, we significantly reduce the time to analyze applications.

Additionally, since one of Cardpliance's tests uses `View.setText()` as a taint sink, we perform a similar optimization in the custom source and sink manager. In this case, we backtrack in the DDG to the definition site of the `View` object and identify the corresponding call to `Activity.findViewById(int)`. We then similarly resolve the integer resource ID. If the ID is in a predefined list (defined via a heuristic for the test), the call to `View.setText()` is defined as a taint sink.

Finally, we had to patch Amandroid's control flow analysis to properly track the use of `View` objects obtained in `OnClickListener` callbacks. We found that many applications declare the `OnClickListener` of a `View` as an anonymous inner class. In such cases, Amandroid did not capture the data flow initiated by the button click. We fixed this issue by adding a dummy edge from the point where the `OnClickListener` was registered to the entry point of the corresponding `OnClickListener.onClick()` method.

4.2 PCI DSS Tests

At a high level, Cardpliance uses Amandroid's taint analysis result (TAR) to identify potential PCI DSS violations. However, the TAR does not consider context at the sources and sinks, or all different paths between the sources and sinks. Cardpliance uses the DDG to identify specific instructions as sources and sinks based on constant values available from the PTAResult hash map. It then calculates all paths between those specific source and sink instructions, determining if specific conditions occur (e.g., calling an obfuscation method).

4.2.1 Analysis Approach

The DDG is a directed acyclic graph (V,E) where the set of vertices V are program instructions and the set of edges E represent def-use dependencies between vertices (v_i, v_j). We say there exists a path between v_s and v_k (denoted $v_s \rightsquigarrow v_k$) if there is a sequence of edges (v_s, v_{s+1}), (v_{s+1}, v_{s+2}), $\ldots (v_{k-1}, v_k)$. We refer to a *specific* path p from v_s to v_k as $v_s \overset{p}{\rightsquigarrow} v_k$.

[1] http://pag.arguslab.org/argus-saf

Each PCI DSS test is defined with respect to instructions invoking three sets of methods: source methods (S), sink methods (K), and required methods (R). S and K are traditional sources and sinks for taint analysis. Whereas Amandroid's sources and sinks are method signatures, some of Cardpliance's sources and sinks are context-sensitive. For example, an instruction that calls `Activity.findViewById(int)` is only a source if the argument is an integer from a list of resource IDs identified by UiRef as requesting credit card information.

In contrast to S and K, R places *requirements* on the data flow path. Informally, R defines a set of methods that should be called on the data flow path (e.g., a string manipulation method that could mask characters). If no methods from R exist on the path, then a potential violation is raised.

We now describe the general template used by each test to generate sets of potential violations. For simplicity, we say that instruction $v \in V$ is in S, K, or R if the instruction v calls a method in one of those sets, potentially parameterized with the correct constant values. Then, for v_s, v_k, v_r in V, the test produces *paths* as potential violations as follows:

$$\{(v_s \overset{p}{\rightsquigarrow} v_k) | v_s \in S, v_k \in K, v_s \overset{p}{\rightsquigarrow} v_k \ \wedge (\nexists v_r \in p | v_r \in R)\}$$

That is, even if $v_s \rightsquigarrow v_k$, it is not a violation if *all* paths include an instruction v_r that is in R. Note that not all tests use R and therefore the logic for these tests skips the second term in the conjunction. However, this is logically equivalent to $R = \emptyset$, which will cause the term to always be true.

4.2.2 Test Implementation

The remainder of this section describes our six PCI DSS tests with respect to S, K, and R. In doing so, we reflect on the relevant requirements described in Section 2. We also describe implementation-specific considerations for each test. An overview of the tests is provided in Table 2.

Test T1 (Storing CHD): Requirement 1 in Section 2 states that storage of cardholder data (CHD) should be limited, and if it is stored, there should be a mechanism to delete if after a period of time. Determining all of the ways in which persistent data can be deleted is not practical to detect using static program analysis. Therefore, Test T1 takes a security-conservative approach and identifies whenever CHD is written to persistent storage. As such, Test T1 is more of a warning than a violation of PCI DSS. However, it is useful as a coarse metric and can bring potentially dangerous situations to the attention of a security analyst.

Test T1 captures a key program analysis primitive that is needed by the other tests: data flow analysis from *specific* UI inputs. Amandroid provides a Taint Analysis Result (TAR) structure that contains a superset of all of the paths identified in *all* of the tests. Test T1 filters the TAR based on the sources and sinks listed in Table 2. Note that Test T1 only considers the sources that call `Activity.findViewById(int)` with resource IDs corresponding to CHD. We further reduce the text input source to just the credit card number (PAN), as there is the potential for ambiguity when identifying the other fields (e.g., cardholder name vs. another name field). The custom source and sink manager described in Section 4.1 only limits the analysis to credit card related data, which includes both CHD and SAD. Therefore, we again use Amandroid's `ExplicitValueFinder`, but within a different phase of the analysis. The data persistence method (DPM) sink methods listed in the table do not require special consideration. Once these concrete sources and sinks are identified, we traverse the DDG to identify all paths between them.

Test T2 (Storing SAD): Requirement 2 in Section 2 states that sensitive account data (SAD) should never be written to persistent storage, including logs. From the mobile application perspective, the only SAD that users will enter into text fields is the three or four digit CVC code written on the physical card. Therefore, Test T2 only needs to consider `Activity.findViewById(int)` sources that are passed resource IDs corresponding to CVC-related fields. The remainder of the analysis is identical to Test T1. Note that unlike Test T1, the existence of a data flow path directly represents a PCI DSS violation.

Test T3 (Masking Credit Card Number): Requirement 3 in Section 2 states that the only time the application should display the full credit card number (PAN) is when the user is entering it in the text field. All other times the credit card number is displayed, it should be masked, showing at most the first six and last four digits of the number.

Test T3 requires additional sophistication in the static program analysis algorithm. First, it includes R, the set of required methods. Recall that a violation does *not* occur if *all* paths from the sources to the sinks include an instruction that invokes a method in R. In this case, we define a set of PAN masking methods (PMM), listed in Table 2, that represent different ways in which the application developer may have masked the credit card number. While the developer may choose to use other string manipulation methods, this set is conservative and will raise an alarm for manual review by a security analyst. Of course, this set can be easily expanded as additional string manipulation methods are discovered.

Second, Test T3 considers not only textual user input as taint sources, but also input from the network. For example, an application may retrieve the credit card number from the server and display it for the user. Such cases should also be masked. However, in this case, it is nontrivial to detect which input data is the credit card number. While the semantics of JSON key-value fields could potentially be used [28, 34], we elected to use a simpler heuristic that filters tainted paths at the sink. Specifically, we extract a list of all resource IDs of UI widgets that exist on a UI screen that also contains the text "Credit Card." Our intuition is that mobile application UI screens are generally purpose-specific and the other displayed information is likely related. This classification allows

Table 2: PCI DSS tests defined by source (S), sink (K), and required (R) methods on data flow paths in the DDG.

Test	Identifies	S	K	R
T1	Storing CHD	`Activity.findViewById(ID_CC)`	DPM	-
T2	Storing SAD	`Activity.findViewById(ID_CVC)`	DPM	-
T3	Not Masking Credit Card Number	`Activity.findViewById(ID_CC)`, `URLConnection.getInputStream()`	`View.setText()`	PMM
T4	Storing Non-Obfuscated Credit Card Number	`Activity.findViewById(ID_CC)`	DPM	OM
T5	Insecure Transmission	`Activity.findViewById(ID_CC)`	`OutputStreamWriter.write()`, `OutputStream.write()`	-
T6	Sharing Non-Obfuscated Credit Card Number	`Activity.findViewById(ID_CC)`	`Intent.putExtra()`, `SmsManager.sendTextMessage()`	OM

Data Persistence Methods (DPM) = `java.io.OutputStream.write()`, `java.io.FileOutputSream.write()`, `java.io.Writer.write()`, `java.lang.System.out.println()`, `android.content.SharedPreferencesEditor.putString()`, `android.util.Log.i()`, `android.util.Log.d()`
PAN Masking Methods (PMM) = `java.lang.String.replace()`, `java.lang.String.substring()`, `java.lang.String.concat()`, `java.lang.StringBuilder.append()`
Obfuscation Methods (OM) = `javax.crypto.Cipher.update()`, `javax.crypto.Cipher.updateAAD()`, `javax.crypto.Cipher.doFinal()`, `java.security.MessageDigest.digest()`, `java.security.MessageDigest.update()`

the static program analysis to only consider `View.setText()` methods as taint sinks if they correspond to objects that were retrieved using `findViewById()` and a resource ID from that set. As mentioned in Section 4.1, we leverage the `ExplicitValueFinder` within the custom source and sink manager to perform this refinement. We, therefore, leverage the `View.setText()` sinks in Amandroid's TAR structure, knowing that they have been refined as such.

Once Test T3 has filtered the TAR with respect to the sources and sinks described above, it computes all paths between them using the DDG. We then remove paths that contain a method from R. The resulting set of paths are potential violations of the PCI DSS and are made available for manual review.

Test T4 (Storing Non-Obfuscated Credit Card Number): Requirement 4 in Section 2 states that the credit card number (PAN) should always be protected if it is stored by the mobile application. The PCI DSS standard has some flexibility in how the number is protected, but it offers suggestions including one-way hash functions and cryptography. Requirement 4 refines Requirement 1 specifically for the credit card number, and since our Test T1 only considers the credit card number, and not the other CHD values, it might seem that both Test T1 and Test T4 are not needed. However, we wanted to include both, because Test T1 will capture all cases when the credit card number is written to persistent storage, whereas Test T4 only raises an alarm when there is not an obfuscation method on the data flow path. Put another way, Test T1 is designed to be a warning for closer inspection, whereas Test T4 is designed to detect violations.

Given the similarity to Test T1, Test T4 follows the same implementation pattern. However, Test T4 includes a set R of required obfuscation methods (OM), as listed in Table 2. These methods include calls to common encryption and message digest functionality in Java, as listed on the Android developer's website [2]. Similar to Test T3's PAN masking methods, we do not seek to enumerate all possible cryptography libraries. Nonstandard libraries should be reviewed and can potentially be added to the list in the future. For the `Cipher.doFinal()` method, we validate that the `Cipher` object is initialized with an `ENCRYPT_MODE`. In the future, additional cryptography checks [14,27] could be incorporated. Note that false negatives resulting from this limitation of Test T4 would still raise a warning for Test T1, which reports any write to storage, obfuscated or not. Finally, Test T4 uses the same strategy as Test T3 for ensuring all paths from the filtered sources and sinks contain a method from R.

Test T5 (Insecure Transmission): Requirement 5 in Section 2 states that mobile applications should always use TLS/SSL when transmitting cardholder data. There are two ways in which an application can fail to properly use TLS/SSL: (1) send data via HTTP URLs, (2) invalidate certificate checks when sending data via HTTPS URLs.

As shown in Table 2, Test T5 uses `OutputStreamWriter.write()` and `OutputStream.write()` as taint sinks to filter the TAR. However, these sinks may also be used for file writes. Unfortunately, the `URLConnection` object used to create the output stream will not be on the tainted path for the credit card number (so R cannot be used). Therefore, we separately walk backward on the DDG from the taint sink to find the `URLConnection` object used to create the output stream object. We then use Amandroid's `ExplicitValueFinder` to determine the argument passed to the corresponding URL initialization method (`URL.init(String)`). We then determine if the string is an HTTP or HTTPS URL. If an HTTP URL is used, an alarm is raised.

If an application has an HTTPS URL as a taint sink, we also check if the application contains a vulnerable TLS/SSL configuration. To do so, we leverage Amandroid's existing API Misuse module, which has a configuration option for

COMMUNICATION_LEAKAGE. Specifically, this check looks for insecure implementations of SSLSocketFactory and a TrustManager that uses the ALLOW_All_HOSTNAME_VERIFIER flag. Note that this analysis is not context-sensitive to a specific taint sink, as these options are often set globally for an application. Therefore, there is a possibility for false positives if an application uses different SSL configurations for different network connections.

Test T6 (Sharing Non-Obfuscated Credit Card Number): Requirement 6 in Section 2 states that credit card numbers should be protected if they are shared outside of the application. Therefore, we consider both SMS APIs and Android's inter-component communication (ICC) mechanism that allows execution to span applications. Similar to Test T4, this test determines if all paths from sources and sinks include a call to an obfuscation method, as shown in Table 2.

Identifying taint sinks for SMS is straightforward due to Android's runtime API SmsManager.sendTextMessage(). Identifying ICC taint sink is more complex. First, ICC is commonly used within an application. To simplify the analysis, we assume that Intent messages with explicit destinations (i.e., specify the exact target component name) are used for ICC within an application, and implicit destinations (i.e., use "action" strings) are used for ICC between applications. Second, the Intent objects used for ICC are populated in steps. We use Intent.putExtra() as a taint sink filter for the TAR. We then backtrack the DDG to find the Intent object creation and use Amandroid's ExplicitValueFinder to identify if it is an implicit or explicit Intent. If it is an implicit Intent and the action value is ACTION_SEND, we use the Intent.putExtra() call as a taint sink, as this is the action string used to share information between applications. Finally, we follow a similar process as Test T4 to ensure that all paths between the sources and sinks include a required obfuscation method from *R*. Paths failing this requirement will raise an alarm.

5 PCI DSS Compliance Study

Our primary motivation for creating Cardpliance was to analyze whether mobile applications are mishandling payment information. The goal of this study is to gauge the impact of PCI DSS non-compliance on real-world users. In this section, we use Cardpliance to analyze popular applications from Google Play for potential PCI DSS violations and present case studies based on our findings.

As Cardpliance uses static analysis to vet application's compliance of PCI DSS requirements, it is subject to the same limitations as static analysis. In particular, static analysis may provide an over-approximation of application behaviors that may result in false alarms. Therefore, we manually validate data flows that Cardpliance flags as potential PCI DSS violations to determine whether the application is actually violating PCI DSS requirements. Note that the goal of validation is to determine whether the application is violating PCI DSS requirements, not to comprehensively determine whether every data flow identified by static analysis is a true positive or false positive. Therefore, a true positive denotes that the application contains a PCI DSS violation while a false positive denotes that none of the data flows flagged by static analysis were valid due to errors in the underlying tooling.

5.1 Dataset Characteristics

To select our dataset, we downloaded the top 500 free applications ("top_selling_free" collection) across Google Play's 35 application categories in May 2019, which resulted in an initial dataset of 17,500 applications. To determine which applications request payment information, we disassembled the dataset and performed a keyword-search on the resource files for terms that describe payment card numbers (e.g., credit card number, debit card number, card number). The list of terms was obtained from the synonym list in UiRef [8] for "credit card number."[2] This keyword-based triaging flagged 1,868 applications as potentially requesting credit card information, which reduced the dataset by 89.3% (15,632/17,500). Note that this triaging may provide an under-approximation of the total number of applications requesting credit card numbers due to the comprehensiveness of the keyword-based list. However, since this keyword list was used by prior work [8] to identify 4,433/50,162 (8.83%) applications in Google Play were asking users for credit card information, we believe it is suitable for our study. We leave it as future work to construct a comprehensive multi-language vocabulary of terms that refer to credit card numbers.

As discussed previously, simply containing a string that matches a credit-card related keyword does not imply the application accepts credit card numbers from the user. Therefore, we use UiRef to determine when an application takes credit card numbers as input. We ran UiRef on the refined dataset and found that 807 applications failed during reassembly due to errors in ApkTool.[3] UiRef failed to extract layouts from an additional 110 applications. Of the remaining 951 applications, UiRef identified that 442 applications containing input widgets that request credit card numbers.

We ran Cardpliance on the 442 application that request payment information. We performed the analysis on a virtual machine running Ubuntu 18.04 on the VMware ESXi 6.4 hypervisor with an Intel(R) Xeon(R) Gold 6130 2.10GHz machine with 320 GB RAM and 28 physical cores. We con-

[2] *Keyword list:* credit card number, card number, cardnumber, credit / debit card number, credit or debit card number, payment card number, credit card number on our order form, credit card number on our registration form, credit-card number, credit / debit card number, credit or debit card number, customer credit card number, credit card / debit card number, credit card account number, credit and debit card number, debit card number, valid credit / debit card number, digit card number, cc number, credit card, debit card, master card, mastercard

[3] https://ibotpeaches.github.io/Apktool/

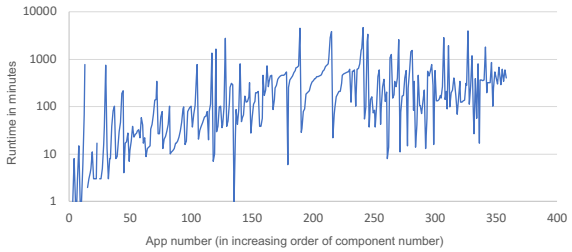

Figure 2: Runtime increases gradually for 358 apps sorted according to increasing component number, it saturates near the 170th app which had 40 components

figured Cardpliance to run 15 applications in parallel and set a 60-minute timeout per application component. If a timeout occurred when analyzing a component, we discard the results for the entire application to avoid partial results. In total, Cardpliance successfully analyzed 80.99% (358/442) of the triaged application dataset. Of the 19.01% (84/442) applications that failed analysis, 3.84% (17/442) applications contained components that exceeded the timeout and 15.15% (67/442) applications could not run due to errors in the underlying static analysis framework Amandroid.

Finding 1: *At least 2.5% of popular free Android applications on Google Play directly request payment information*. As discussed above, we used a lightweight heuristic to identify which applications were mentioning credit card numbers and then used UiRef to resolve semantics. We found that 442 applications contain input widgets that directly request payment information from users (i.e., credit card numbers). This reduction in the scope of analysis makes deploying the deeper and more time-consuming static analysis checks provided by Cardpliance feasible at scale. Note that this is a conservative lower-bound estimate, as we could not analyze 917 applications due to errors in ApkTool and UiRef.

Finding 2: *Cardpliance can analyze an application with a mean and median runtime of 334 minutes and 179 minutes, respectively*. Figure 2 plots the runtime versus the number of components within an application. Note that the x-axis consists of the 358 applications sorted in ascending order based on the number of components within the application. The component counts within applications ranged from 0 to 315 components where 54 was the average number of components per application. As shown in Figure 2, an increased number of components within an application generally resulted in a longer runtime. Further, it saturates after the 170th application where there were 40 components. The mean and median runtime for applications was 334 and 179 minutes per application, respectively.

Cardpliance's runtime significantly increased over the stock version of Amandroid [19] due to the inclusion of frequently used user input sources and sinks, such as `Activity.findViewById(int)` and `View.setText()`. For example, an application may only have the source `TelephonyManager.getDeviceID()` once within the application, but it may likely have the source `Activity.findViewById(int)` multiple times throughout the application, which significantly increases the number of sources that require tracking. Therefore, in order to scale Cardpliance to an entire market, a lightweight keyword-based filter is required (as shown in Figure 1). Note that if the filter is not comprehensive, non-compliant applications may not be discovered. We discuss this limitation further in Section 5.7.

Finally, as discussed above, Cardpliance successfully analyzed 358 applications. Those 358 applications spanned 32 application categories with the majority coming from the FOOD_AND_DRINK (51), SHOPPING (43), FINANCE (39), and MAPS_AND_NAVIGATION (37). The average download count for these applications was 1.25 million downloads and an average rating of 3.8 stars out of 5. The most popular application Wish- Shopping Made Fun (`com.contextlogic.wish`) in the group had over 100 million downloads. The dataset consisted of other widely used applications, such as Lyft (`me.lyft.android`), CVS Caremark (`com.caremark.caremark`), and the WWE application (`com.wwe.universe`).

5.2 Validation Methodology

We opt for manual code review instead of manually running the application due to complexities of reaching screens that request payments (e.g., creating accounts that require disclosure of sensitive data, requiring referral codes, or relying on an existing balance/debt). The manual code review for validation was performed by one student author of this paper, who has more than 6 years of both academic and industrial experience programming Java and developing Android applications. For each candidate application flagged by Cardpliance, we begin by decompiling the application with the JEB decompiler [37] to obtain the source code. We then group the data flows that were marked as potential PCI DSS violations by the PCI DSS requirement that it violated from Section 4).

The goal of validation is to verify that the data flow actually occurs within the code and was not a false alarm due to the imprecision of the underlying tooling. Note that for all of the validation checks, we stop verification if we discover that the result is a false alarm and begin validating the next data flow within the PCI DSS requirement group. If all of the data flows within the PCI DSS requirement group are erroneous, we mark the application as a false positive for that PCI DSS requirement group. However, if we successfully validate the data flow, we mark the application as containing a PCI DSS violation and start analysis on the next PCI DSS requirement group for that application.

We begin by validating whether the semantics linked to the input widget of the data flow was correctly resolved by UiRef. We start at the source of the data flow (e.g., `Activity.findViewById(int)` method) and resolve the integer parameter of the method invocation to the resource identifier in the R.java file

of the source code. We identify in the input widget referenced by the resource identifier within the source code and validate that UiRef made the correct resolution of semantics (i.e., credit card number, CVC). If UiRef was incorrect, we mark the data flow as erroneous and begin validating the next data flow for that requirement group. If UiRef resolved the correct semantics, we continue the following validation process.

Next, we trace through the source code from the source of the data flow to the sink to determine that the data flow exists within the source code. For example, if the data flow denotes that non-obfuscated credit card numbers are being stored, we verify that the data retrieved from the input widget accepting credit card numbers is actually written to disk without being encrypted or through some other obfuscation library. If the data flow does not occur within the source code due to imprecisions of static analysis, we mark it as an error and continue analysis as discussed above. For example, we found that the Context object of the `Activity.findViewById(int,Context)` method was frequently tainted and lead to imprecision.

Finally, for validating potential SSL vulnerabilities that lead to insecure transmission, we searched for `SSLSocketFactory` and `TrustManager` classes within the source code and manually checked whether the implementation was performing improper certificate validation. We then searched for the use of those classes throughout the source code and determined whether payment information was sent over connections using these vulnerable classes.

5.3 Compliance: The Good

In this section, we report the positive findings from our analysis of the 358 applications analyzed by Cardpliance. We believe that these findings provide significant value and insight to the community.

Finding 3: *Around 98.32% of the 358 applications pass Cardpliance's PCI DSS compliance tests*. Out of the 358 applications, Cardpliance identified that 318 applications did not violate any of the PCI DSS compliance checks. After manual validation of Cardpliance's findings, we found that *352* applications in total were not violating any PCI DSS check that we modeled. This result in itself is surprising due to the vast amount of prior research that highlights the poor state of Android application security [9, 15, 16, 24]. The fact that our tool reporting *98.32%* of applications in our dataset handling payment information are maintaining these data security standards shows that the risk of financial loss due to insecure behaviors in mobile applications might not be as wide-spread. Further, as the majority of applications seem to be handling payment information correctly, it demonstrates that securely processing payment information and meeting PCI DSS requirements within a mobile application is largely an obtainable effort.

Finding 4: *Applications are correctly using HTTPS instead of HTTP to transmit payment information*. Cardpliance did not identify any applications that transmitted payment information insecurely in plaintext over HTTP in Test T5. The adoption of HTTPS over the insecure HTTP is a great move in the right direction, as a prior study [17] showed that 93.4% of URLs in Android applications were HTTP and another study showed poor SSL adoption in financial applications in developing countries [33]. The fact that we did not find any applications sending payment information over HTTP means that the effort to push HTTPS adoption has been working for transmitting sensitive information, such as payment information. Note that as Cardpliance is a static analysis-based approach, we cannot determine whether payment information is sent insecurely if the destination URLs are not present in the code or resource files. This limitation is shared by practically all prior work on this same problem [17, 33].

Finding 5: *Applications are correctly performing hostname and certificate verification when sending payment information over SSL connections*. Cardpliance identified 20 applications that were handling payment information and also contained vulnerable SSL implementations within their codebase. Out of these 20 applications, we did not find evidence that any payment information was sent over vulnerable SSL connections during manual verification. The majority of the code for the vulnerable SSL implementation was dead code or contained build flags that disabled that functionality. Overall, this finding demonstrates the positive impact on Android application security by prior research on SSL misconfigurations [17] and Google's efforts.[4]

Note that we did find that the Harris Teeter application (com.harristeeter.htmobile) sends profiling and usage data to Dynatrace over a vulnerable SSL connection, which results from a misconfiguration when interfacing with the Dynatrace library. This issue of sending non-payment information indicates that vulnerable SSL problems still exist. As recommended in Section 6, developers should never modify `SSLSocketFactory` or `TrustManager` within the application. Further, third-party libraries that applications are including should also be vetted, as they can override the `TrustManager` used by the default `SSLSocketFactory`, which could result in all SSL connections within the application becoming vulnerable.

Finding 6: *Applications are not insecurely sharing payment information via SMS or with other applications via ICC channels*. Cardpliance did not identify any applications transmitting payment information to other applications using SMS APIs or implicit intents without obfuscating the data in Test T6. Prior research [23, 24] highlighted that a wide range of private data was being leaked through ICC, such as location data and device identifiers. In this work, we demonstrate that credit card numbers are not being insecurely exposed through the use of implicit intents. One potential mitigating factor may have been that Android banned binding to services with implicit intents since Android 5.0 [1].

[4] https://support.google.com/faqs/answer/6346016?hl=en

Table 3: Applications with Validated PCI DSS Violations

App Name	Package Name	Downloads	T1	T2	T3	T4
Credit Card Reader	com.ics.creditcardreader	500K+	X			X
FastToll Illinois	com.pragmistic.fasttoll	10K+	X	X		X
Bens Soft Pretzels	com.rt7mobilereward.app.benspretzel	10K+	X	X	X	X
The Toll Roads	com.seta.tollroaddroid.app	100K+	X	X		X
ConnectNetwork by GTL	net.gtl.mobile_app	1M+			X	
Peach Pass GO!	com.srta.PeachPass	50K+	X			X

5.4 Non-Compliance: The Bad and the Ugly

After validation of the 40 applications that Cardpliance flagged as having potential PCI DSS violations, we found that 6 applications were non-compliant with PCI DSS requirements. Table 3 lists all of the applications that contain PCI DSS violations. While the fact that only *1.67%* of the 358 credit card number collecting applications are non-compliant with PCI DSS requirements does not seem surprising in itself, the fact that *any* applications are non-compliant is troublesome. The impact of non-compliance is substantial to both the end-users, app developers, payment processors and issuing banks. For end-users, non-compliance may result in significant financial loss due to fraud if payment information is insecurely exposed. For companies, non-compliance can result in damage to public perception and also significant financial loss up to $5,000 to $100,000 a month depending on the size of the business and degree of non-compliance [5].

While identifying 6 PCI DSS violations out of 40 applications is not ideal, we narrowed the scope of analyzing PCI DSS compliance from manually validating 17,500 applications to only requiring manual validation of 40 applications. Further, the main source of imprecision was due to the data flow analysis in Amandroid. For example, we found that the context object of the `Activity.findViewById(int,Context)` method was frequently tainted and became a large source of imprecision. Further, the context insensitive analysis of SSL vulnerabilities also contributed to the low precision. Future work can improve the precision of the data flow tracking in static analysis tooling to reduce false alarms. The remainder of this section highlights our findings on the PCI DSS violations that Cardpliance identified within applications and case studies from our analysis.

Finding 7: *Applications totaling over 1.5 million downloads are not complying with PCI DSS regulations*. After verification, we found that 6 applications were non-compliant with the PCI DSS requirements. These violations were distributed across applications from popular merchant applications, toll-paying apps, and communication networks. The impact of these violations even reached vulnerable populations of users, such as the application for ConnectNetwork, which is an application that allows users to call and send messages to family and friends incarcerated within a prison. In total, the download counts of these 6 applications reached around *1.5* million downloads. Therefore, up to *1.5* million users were potentially impacted by the PCI DSS violations that Cardpliance identified and may be at risk for potential fraud. Findings 8-10 discuss each of the PCI DSS violations in depth.

Finding 8: *Applications are storing credit card numbers without hashing or encrypting the data*. Figure 3 shows that Cardpliance identified that 20 applications were persisting credit card numbers in files, shared preferences, and device logs (T1) with 19 of those applications not hashing or encrypting the data (T4). After manual validation, we found 5 out of those 20 (25%) applications were actually persisting credit card numbers and none of them were providing adequate protection of the data as defined by PCI DSS requirements by hashing or encrypting it. While we did not verify whether the location that the data is being saved was accessible to external applications, the fact that data is being saved in plaintext is a security risk. For example, consider the case where a user's device is compromised by a malicious application that obtains root access to the device. Even if the application stores the data within its private directory that is traditionally protected by UNIX file system privileges, the malicious application can simply read it due to its escalated privileges. Therefore, all credit card numbers should be either hashed or encrypted before storing. If encrypting, the application should also use the Android Keystore to protect access to the cryptographic key.

Although PCI DSS requirements allow storing of credit card numbers, PCI-DSS guideline 3.4.d states that application logs should not contain credit card numbers in plaintext. We found 4 applications writing credit card numbers to logs in plaintext. Examples of applications persisting and logging credit card numbers in plaintext are discussed in Section 5.5.

Finding 9: *Applications are persisting card verification codes (CVCs)*. As shown in Figure 3, we validated that 3/8 (37.5%) applications were persisting card verification codes (CVCs) that Cardpliance identified. As discussed in Section 2, PCI DSS mandates that CVCs should never be stored even after authorization. One application called The Toll Roads (`com.seta.tollroaddroid.app`) has over 100k+ downloads on Google Play is used to estimate and pay tolls when traveling. This application was flagged by Cardpliance for outputting the payment request along with the CVC to the device logs. Similarly, another application for a franchise restaurant called

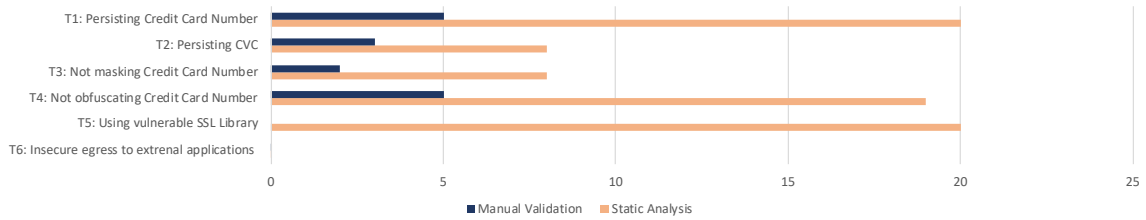

Figure 3: Number of PCI DSS non-compliant applications for different tests.

Ben's Soft Pretzels (`com.rt7mobilereward.app.benspretzel`) with over 10k+ downloads was also writing the CVC to the device logs. Another toll application called FastToll Illinois (`com.pragmistic.fasttoll`) is used to pay tolls acquired within Illinois and has over 10k+ downloads. Cardpliance identified that this application was persisting the CVC in the shared preferences of the application.

Finding 10: *Applications are not masking credit card numbers when displaying them in the user interface*. Figure 3 shows that Cardpliance identified that 8 applications were displaying credit card numbers without partial masking. After validation, we verified that that 2 (25%) applications were not partially masking credit card numbers and violating PCI DSS requirements. An application called ConnectNetwork by GTL (`net.gtl.mobile_app`) has over 1M+ downloads and allows friends and family members to send messages and call people incarcerated within a prison. This application takes the user's credit card number as input in one UI widget and then displays it in another UI widget for validation without partially masking the credit card number. Section 5.5 discusses the other application in detail. Other than directly violating PCI DSS compliance, all of these applications are putting users at risk of financial loss due to potential shoulder surfing including vulnerable population groups of users such as those using the ConnectNetwork by GTL application.

5.5 Case Studies

In this section, we discuss two interesting case studies that demonstrate how applications are potentially mishandling credit card information and thus violating PCI-DSS.

Case Study 1: A credit card reader application is mishandling hundreds-of-thousands of customer's credit card numbers: Credit Card Reader (`com.ics.creditcardreader`) is a popular Android application from Google Play store with 500k+ downloads. This application functions similarly to point-of-sale machines and allows the user to accept physical payments from customers. Cardpliance identified that this application was persisting credit card numbers without hashing or encrypting the information. A snippet of the source code for this application is shown in Listing 1. As shown in line 23, the application is obtaining the user's credit card number from the EditText widget in the user interface and directly logging it to LogCat.

```
1  @Override  // android.view.View\$OnClickListener
2  public void onClick(View v) {
3      switch(v.getId()) {
4          case 0x7F060002: {  // id:action_next
5              Intent i = new Intent(this, TipActivity.class);
6              if(this.cc_sales_tax.isChecked()) {
7                  i.putExtra("sale_amount", String.format("%.2f", Double.valueOf
       (Double.valueOf(this.sale_amt).doubleValue() + Double.valueOf(this.
       sale_amt).doubleValue() * this.sale_tax_per / 100)));
8              }
9              else {
10                 i.putExtra("sale_amount", this.sale_amt);
11             }
12
13             i.putExtra("cc_no", this.cc_no.getText().toString());
14             i.putExtra("cc_exp", this.cc_exp.getText().toString());
15             i.putExtra("cc_cvv2", this.cc_cvv2.getText().toString());
16             i.putExtra("cc_zip", this.cc_zip.getText().toString());
17             i.putExtra("cc_st_add", this.cc_st_add.getText().toString());
18             this.startActivity(i);
19             break;
20         }
21     }
22
23     Log.d("CCR - Payment", this.cc_no.getText().toString());
24 }
25 \vspace{-3em}
```

Listing 1: Code Snippet of Credit Card Reader logging customer's credit card number

Note that this scenario is substantially worse than other applications logging payment information, as it is exposing credit card numbers of unsuspecting customers. As the application has over 500k+ downloads and merchants may handle a wide range of customers, the amount of customers impacted is ultimately unbounded but likely in the hundreds-of-thousands. As discussed in Finding 8, this practice violates PCI-DSS guideline 3.4.d. Further, logging the credit card number also introduces additional risks of fraud. For example, if an adversary obtains physical access to the device, they can download all of the customers' credit card numbers in plaintext. In addition, if the user's device is compromised, a malicious application with escalated privileges could also potentially read all of the customers' credit card numbers in plaintext. We recommend developers completely avoid writing credit card numbers to logging mechanisms.

Case Study 2: An application for placing online orders at a restaurant franchise is persisting credit card numbers in plaintext along with CVCs: A franchise restaurant called Ben's Soft Pretzels has an application on Google Play (`com.rt7mobilereward.app.benspretzel`) with over 10K+ downloads. Based on the developer identifier and website on Google Play, the development of the application appears to have been outsourced to a company called RT7 Incorporated. The application allows users to place online orders from the restaurant and it accepts credit card payments via the application. Cardpliance identified that this app was persisting

```
1  private SecretKeySpec setthekey() {
2          SecretKeySpec v0_1;
3          try {
4              SecureRandom v0 = SecureRandom.getInstance("SHA1PRNG");
5              v0.setSeed(CreditCardEnterPage.userId.concat(CreditCardEnterPage.
       userCardNumber).getBytes());
6              KeyGenerator v1 = KeyGenerator.getInstance("AES");
7              v1.init(0x80, v0);
8              v0_1 = new SecretKeySpec(v1.generateKey().getEncoded(), "AES");
9          }
10         catch(Exception unused_ex) {
11             Log.e("AES Error", "AES secret key spec error");
12             v0_1 = null;
13         }
14
15         if(v0_1 != null) {
16             String v1_1 = Base64.encodeToString(v0_1.getEncoded(), 0);
17             SharedPreferences.Editor v2 = PreferenceManager.
       getDefaultSharedPreferences(this.getApplicationContext()).edit();
18             v2.putString("GetDataPoss".concat(CreditCardEnterPage.userId).concat
       (CreditCardEnterPage.userCardNumber), v1_1);
19             Log.d("ToChangedStores", v1_1);
20             v2.apply();
21         }
22
23         return v0_1;
24     }
25
```

Listing 2: Code Snippet of Ben's Soft Pretzels app insecurely generating and handlig encryption key.

credit card numbers without hashing or encrypting, persisting CVCs, and not masking credit card numbers when displaying.

Our validation of the application uncovered several concerning problems. In particular, we found that they were attempting to encrypt the credit card number before storing to `SharedPreferences`. However, the key in the key-value pair used to store the encrypted credit card number was the concatenation of a constant string and the user's credit card number and username. Therefore, the credit card number is still being persisted to disk in plaintext. Further, as shown in Listing 2, they use the bytes from the username and credit card number to seed the random number generator for generating the key. This encryption key is also written to the logs and `SharedPreferences` as a value under a key that contains both the card number and username. In addition, we found that when the user clicks on the pay button, the credit card number and CVC are both logged. If any of the fields that the user entered are empty when the button is clicked, the remaining payment information is also logged (e.g., expiration date, name, address, and zip code). Moreover, in the `CreditCardSaved2Page` Activity, the application saves the credit card number in plaintext and CVC code to `SharedPreferences` as values under the keys "CardNumTemp" and "CardCvcTemp," respectively. If the user traverses back to the page, both the credit card number and CVC are fetched from `SharedPreferences` and repopulated into the text fields. Note that re-displaying credit card numbers without masking is a violation of PCI DSS. In Section 6, we provide recommendations on how developers can securely handle credit card numbers and CVCs and generate and protect encryption keys.

5.6 Disclosure of Findings

Cardpliance identified 15 PCI DSS violations in 6 applications from Google Play Store which is enlisted in Table 3. For each of these applications, we tried to reach out to the developers through their email addresses mentioned in Google Play. All of the emails were successfully delivered to the corresponding email addresses enlisted in Google Play. In each email, we mentioned the application name, package name, timeline and the PCI DSS violations. For each PCI DSS violation, we reported why it was a violation with reference to the PCI DSS document and the source where the violation occurred.

As of 75 days after disclosure, only one developer responded to our message. A 16.6% response rate is not unexpected considering the fact that responding could raise liability concerns. The responding developer agreed with all but one of the reported vulnerabilities, promising to fix them. We asked for clarification as to why the last issue was not a vulnerability, but did not receive a reply. At the time of camera-ready preparation, we have not seen an updated version of the application in Google Play.

5.7 Threats to validity

The PCI DSS standard is a human-readable document and does not provide precise requirements. Furthermore, the standard applies to a wide variety of payment technology, and it is not specific to mobile applications collecting credit card information from users. Sections 2 and 4 describe our interpretation of PCI DSS into a precise static analysis task.

False Negatives: Due to the time needed for static program analysis, Cardpliance uses a lightweight filter based on credit card related keywords. Excluding applications during the filtering phase may result in false negatives. While we believe our keyword list is sufficiently comprehensive, it only contains keywords for the English language. Since a keyword search is also used by Test T3 to identify payment UIs, an incomplete keyword list may also result in false negatives for Test T3. Additional false negatives may occur when applications request user input through WebViews or use graphical icons to indicate the entry of a credit card number. Cardpliance is also reliant on UiRef [8] to identify taint sources. UiRef does not handle dynamically generated user interfaces.

Static program analysis tools such as Amandroid [19] are neither sound or complete. While any static analysis can be evaded with sufficient effort, we believe that most legitimate applications have little incentive to violate PCI compliance. We conservatively constructed our rules to mitigate false negatives and created test applications to thoroughly validate the logic for each test. Of note, Cardpliance detected when our test applications sent data over HTTP and sent an unprotected PAN through Android's SMS API or implicit intent, neither of which were observed in real applications.

Our SSL vulnerability study was limited to poor certificate validation, which is a common issue for Android applications. While we did not identify any `http://` URLs, this may have resulted from limitations in static analysis (e.g., string values not in the code). Our heuristics in Test T4 also did not consider

the cryptographic keys or cipher suites when determining if data is safely obfuscated before writing to persistent storage.

In Test T6, we assume explicit intents are used for ICC within an application. This assumption may introduce false negatives if applications use explicit intents to invoke components in external applications. However, doing so would require detailed knowledge of the external application's APIs, which may change in subsequent versions. Therefore, we expect it will only occur in rare circumstances.

False Positives: We used manual validation to eliminate false positives in our reported findings. False positives were observed in several situations. First, UiRef caused two false positives for Test T1 when determining UI input semantics (i.e., email address and card expiry). Second, a significant cause of false positives (particularly in tests T1 and T4) was tainting the context object in the `findViewById(context,id)` source. This context variable is a singleton for the entire Activity. When this common variable is tainted, the taint propagates to unrelated code where the context object is used, causing false positives. Third, several false positives in Test T5 resulted from the context-insensitive identification of vulnerable SSL libraries that were more generic rather than being specific to payment credentials. Fourth, false positives resulted from Test T3's lightweight heuristic for masking, because identifying user input from the network is difficult to perform statically. Finally, Test T6 assumes that implicit intents are only used for ICC between applications. Therefore, Test T6 may produce false positives if an application invokes its own components using implicit intents. However, we did not encounter such false positives in our study.

6 Recommendations for Developers

PCI DSS v3.2.1 contains 139 pages of requirements, many of which are not relevant to mobile applications. This section seeks to provide a consolidated list of "best practice" recommendations for developers building Android applications that ask the user to enter a credit card number.

1. *Delegate responsibility of payment processing to established third-party payment providers.* Where possible, we recommend developers consider using established third-party payment processors like Stripe, Square, or PayPal. By not requesting and processing payment information, developers can delegate much of the responsibility of PCI DSS compliance to the payment processor.
2. *Do not write the CVC to persistent storage or log files.* PCI DSS explicitly states that Sensitive Account Data (see Table 1) should never be written to storage. This includes the CAV2, CVC2, CVV2, and CID values.
3. *Avoid writing the credit card number to persistent storage or log files.* While PCI DSS does permit writing the credit card number to storage for a short period (if encrypted), it is safer to not write it all. If the user needs to save their card number, developers should consider storing it on a secure server along with the user's account.
4. *Encrypt credit card numbers with secure randomly generated keys before storing locally.* If the credit card number must be saved locally, it should be encrypted with a key managed by Android's Keystore. Keys hard-coded in applications are easily discovered. Developers should use randomly-generated keys (e.g., `SecureRandom` class without a hardcoded seed) and follow PCI DSS recommendations for key length and using established cryptographic libraries like `javax.crypto`.
5. *Always send payment information over a secure connection when transmitting over the network.* Applications should use HTTPS instead of HTTP when sending payment information over the network.
6. *Never modify the `SSLSocketFactory` or `TrustManager` within the application code.* If there is a need to pin the SSL connection to a specific CA, use the `networkSecurityConfig` option[5] in the application's manifest file. If a test server is needed during development, create a custom certificate for the development server and add the custom certificate to test devices. Developers should also vet that included third-party libraries do not include vulnerable implementations that override the default SSL socket factory and hostname verifiers.
7. *Always mask the credit card number before displaying it.* Only the first six and the last four digits may be displayed on subsequent screens.
8. *Only use explicitly-addressed `Intent` messages when sharing payment information across Android components.* Using implicit `Intents` addressed with action strings may result in unintentional access by other apps.

7 Related Work

Securing payment cards has been an important question leading to seminal papers in computer security [7, 13], yet continues to remain relevant [4, 10, 13, 35, 36]. For example, magnetic stripe cards are easily cloned [4, 7], and only recently have mechanisms to detect this attack been developed [35, 36]. Instead, much of the research has examined EMV chip-based cards, finding and mitigating vulnerabilities related to unauthenticated terminals [13] and pre-play attacks [10].

Payments, however, have moved to mobile devices, making mobile app security an important question for payments. Recent analyses [11, 33] of branchless banking applications found flaws related to misuse of cryptography, flawed authentication, and SSL/TLS misconfiguration. SSL/TLS security is especially important for mobile payments, who primarily rely on HTTP-based APIs. Mobile platforms do this

[5] https://developer.android.com/training/articles/security-config

correctly by default, yet developers frequently break certificate validation, creating the possibility for man in the middle attacks [17, 18, 21, 31, 38]. Studies of mobile payment platforms [42] and documentation [12] in China have also demonstrated vulnerabilities in the payment protocols. Further studies on cryptography in Android apps have shown that incorrect use is rampant [14, 27].

Our work also builds on prior work studying information flows in Android apps. Much of this work has built tools to demonstrate undesired leakage of sensitive data [9, 15, 16, 23]. We rely on the extensive body of literature developing static analysis techniques for Android apps [9, 19, 20, 22, 29, 30].

The academic work closest to ours includes UIRef [8], which previously identified credit card collection in Android apps, but provided no further analysis. A second study investigated the PCI DSS compliance of e-commerce websites as well as the effectiveness of PCI scanners for the web [32]. However, our work is the first to investigate the question of payment card handling in the context of mobile apps.

8 Conclusion

Mobile Payment applications improve the standard of trade and commerce. Their ease and flexibility has attracted a wide range of customers and also potential adversaries. Therefore, vetting the security of these applications is paramount to reduce fraud and abuse. We designed and used Cardpliance to study 358 popular Android applications on Google Play that request credit card numbers. While our study demonstrates that most of the 358 applications (98.32%) properly handle payment data according to Cardpliance, some applications still improperly store credit card numbers and card verification codes. The findings from our study demonstrate a positive landscape of PCI DSS compliance in popular Android applications on Google Play.

Acknowledgments

We thank our shepherd, Mary Ellen Zurko and all the anonymous reviewers for their insightful comments. This work is supported in part by NSA Science of Security award H98230-17-D-0080 and NSF SaTC grant CNS-1513690. Any findings and opinions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

Availability

The source code for Cardpliance is publicly available at https://github.com/wspr-ncsu/cardpliance.

References

[1] Android Component and Services. https://developer.android.com/guide/components/services.

[2] Cryptography | Android Developers. https://developer.android.com/guide/topics/security/cryptography.

[3] How Google Pay works. https://support.google.com/pay/merchants/answer/6345242?hl=en.

[4] Insert skimmer and pin stealer. https://krebsonsecurity.com/2019/03/insert-skimmer-camera-cover-pin-stealer/.

[5] PCI Compliance Fees, Fines, and Penalties: What Happens After a Breach? https://www.lbmc.com/blog/pci-compliance-fees-fines-and-penalties/.

[6] The Payment Card Industry Data Security Standard. https://www.pcisecuritystandards.org.

[7] Ross Anderson. Why Cryptosystems Fail. In *Proceedings of the 1st ACM conference on Computer and Communications Security (CCS))*, 1993.

[8] Benjamin Andow, Akhil Acharya, Dengfeng Li, William Enck, Kapil Singh, , and Tao Xie. UiRef: Analysis of Sensitive User Inputs in Android Applications. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2017.

[9] Steven Arzt, Siegfried Rasthofer, Christian Fritz, and Eric Bodden. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[10] Mike Bond, Omar Choudary, Steven J. Murdoch, Sergei Skorobogatov, and Ross Anderson. Chip and Skim: cloning EMV cards with the pre-play attack. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[11] Sam Castle, Fahad Pervaiz, Galen Weld, and Richard Anderson. Let's talk money: Evaluating the security challenges of mobile money in the developing world. In *Conference: the 7th Annual Symposium*, 2016.

[12] Yi Chen, Luyi Xing, Yue Qin, Xiaojing Liao, XiaoFeng Wang, Kai Chen, and Wei Zou. Devils in the Guidance: Predicting Logic Vulnerabilities in Payment Syndication Services through Automated Documentation Analysis. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[13] Saar Drimer and Steven J. Murdoch. Keep your enemies close: Distance bounding against smartcard relay attacks. In *Proceedings of the USENIX Security Symposium*, 2007.

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

[15] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol 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 (OSDI)*, October 2010.

[16] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the USENIX Security Symposium*, August 2011.

[17] Sascha Fahl, Marian Harbach, and Thomas Muders. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *Proceedings of the ACM conference on Computer and communications security(CCS)*, 2012.

[18] Sascha Fahl, Marian Harbach, and Henning Perl. Rethinking SSL development in an appified world. In *Proceedings of ACM SIGSAC conference on Computer & communications security*, 2013.

[19] Xinming Ou Fengguo Wei, Sankardas Roy and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2014.

[20] Christian Fritz and Steven Arzt. Highly precise taint analysis for android applications. In *Technical report, EC SPRIDE*, 2013.

[21] 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 ACM conference on Computer and communications security(CCS*, 2012.

[22] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*, February 2015.

[23] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 280–291, Piscataway, NJ, USA, 2015. IEEE Press.

[24] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. I Know What Leaked in Your Pocket: Uncovering Privacy Leaks on Android Apps with Static Taint Analysis. In *CoRR*, 2014.

[25] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 229–240, 2012.

[26] M Marlinspike. New tricks for defeating ssl in practice. In *Black Hat Europe*, 2009.

[27] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. Source attribution of cryptographic api misuse in android applications. In *ACM Asia Conference on Computer and Communications Security*, 2018.

[28] Yuhong Nan, Zhemin Yang, Xiaofeng Wang, Yuan Zhang, Donglai Zhu, and Min Yang. Finding Cludes for Your Secrets: Semantics-Driven, Learning-Based Privacy Discovery in Mobile Apps. In *Proceedings of the ISOC Network and Distributed Systems Security (NDSS) Symposium*, 2018.

[29] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting Android Applications to Java Bytecode. In *Procedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, November 2012.

[30] Damien Octeau, Patrick McDaniel, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[31] Lucky Onwuzurike and Emiliano De Cristofaro. Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks)*, 2015.

[32] Sazzadur Rahaman, Gang Wang, and Danfeng Yao. Security Certification in Payment Card Industry: Testbeds,

Measurements, and Recommendations. In *26th ACM Conference on Computer and Communications Security (CCS)*, 2019.

[33] Bradley Reaves, Nolen Scaife, Adam Bates, Patrick Traynor, , and Kevin R.B. Butler. Mo(bile) money, mo(bile) problems: Analysis of branchless banking applications in the developing world. In *Proceedings of the 24th USENIX Security Symposium*, 2015.

[34] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *Proceeddings of the ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.

[35] Nolen Scaife, Christian Peeters, and Patrick Traynor. Fear the Reaper: Characterization and Fast Detection of Card Skimmers. In *Proceedings of the USENIX Security Symposium*, 2018.

[36] Nolen Scaife, Christian Peeters, Camilo Velez, Hanqing Zhao, Patrick Traynor, and David Arnold. The cards aren't alright: Detecting counterfeit gift cards using encoding jitter. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2018.

[37] PNF Software. JEB, An android decompiler. https://www.pnfsoftware.com, 2019.

[38] David Sounthiraraj, Justin Sahs, Zhiqiang Lin, Latifur Khan, and Garrett Greenwood. SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*, February 2014.

[39] Nick Statt. Fortnite for Android will ditch Google Play Store for Epic's website. https://www.theverge.com/2018/8/3/17645982/epic-games-fortnite-android-version-bypass-google-play-store, August 2018.

[40] Nick Statt. Tinder is now bypassing the Play Store on Android to avoid Google's 30 percent cut. https://www.theverge.com/2019/7/19/20701256/tinder-google-play-store-android-bypass-30-percent-cut-avoid-self-install, July 2019.

[41] Justin Del Vecchio, Feng Shen, Kenny M. Yee, Boyu Wang, Steven Y. Ko, and Lukasz Ziarek. String Analysis of Android Applications. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[42] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, and Dawu Gu. Show Me the Money! Finding Flawed Implementations of Third-party In-app Payment in Android Apps. In *Proceedings 2017 Network and Distributed System Security Symposium*, San Diego, CA, 2017.