Analysis of Payment Service Provider SDKs in Android

Samin Yaseer Mahmud
North Carolina State University
Raleigh, North Carolina, USA
smahmud@ncsu.edu

K. Virgil English
North Carolina State University
Raleigh, North Carolina, USA
kvenglis@ncsu.edu

Seaver Thorn
North Carolina State University
Raleigh, North Carolina, USA
swthorn@ncsu.edu

William Enck
North Carolina State University
Raleigh, North Carolina, USA
whenck@ncsu.edu

Adam Oest
PayPal
Scottsdale, Arizona, USA
aoest@paypal.com

Muhammad Saad
PayPal
Scottsdale, Arizona, USA
muhsaad@paypal.com

ABSTRACT

Payment Service Providers (PSPs) provide software development toolkits (SDKs) for integrating complex payment processing code into applications. Security weaknesses in payment SDKs can impact thousands of applications. In this work, we propose AARDroid for statically assessing payment SDKs against OWASP's MASVS industry standard for mobile application security. In creating AARDroid, we adapted application-level requirements and program analysis tools for SDK-specific analysis, tailoring dataflow analysis for SDKs using domain-specific ontologies to infer the security semantics of application programming interfaces (APIs). We apply AARDroid to 50 payment SDKs and discover security weaknesses including saving unencrypted credit card information to files, use of insecure cryptographic primitives, insecure input methods for credit card information, and insecure use of WebViews. These results demonstrate the value of applying security analysis at the SDK granularity to prevent the widespread deployment of insecure code.

1 INTRODUCTION

Mobile devices are a critical component of the modern digital payments ecosystem. Security weaknesses in payment applications can allow malicious applications on the device and on-path attackers in the network to steal payment credentials and hijack transactions. Although banks and payment services have long been subject to government compliance regulations and mature industry security standards such as the Payment Card Industry Data Security Standard (PCI-DSS) [47], comparatively little scrutiny is given to third-party applications that integrate with payment services.

Mobile application security standards, e.g., OWASP's Mobile Application Security Verification Standard (MASVS) [43], are difficult to apply to SDKs in isolation. First, the standards assume the full context of the application is known and can be controlled. SDKs are designed to be included into applications and some requirements do not apply or only partially apply. Second, existing program analysis tools rely on the known runtime semantics of applications (e.g., Android's component lifecycle). In contrast, the semantics of entry points to code in SDKs is often unknown, and high-quality documentation is frequently not publicly available. Finally, while prior work has considered the security of payment protocols [63, 67] and applications taking payment information [12, 37], no prior work has performed a comprehensive security analysis of code in the payment SDKs themselves.

In this paper, we propose AARDroid[1] for identifying security weaknesses in Android SDKs, specifically those used for payments. The key contribution of this work is adapting the MASVS [43] application-level requirements and program analysis tools for SDK-specific analysis, tailoring dataflow analysis for SDKs using text analytics and a domain-specific ontology to infer the security semantics of SDK APIs. We used AARDroid to study how a set of 50 payment SDKs adhere to the MASVS requirements, finding that 37 SDKs did not meet at least one Level 1 requirement (Level 2 is recommended for applications requiring PCI-DSS compliance).

Reviewing the AARDroid analysis results in detail, we uncovered several concerning trends. First, we found three SDKs save unencrypted sensitive financial information such as credit card number and CVC to either persistent storage or device logs. A fourth SDK stores a CVC in encrypted form. PCI-DSS standards prohibit such storage. Second, we found 11 SDKs rely on outdated cryptographic primitives for sensitive functionality. Third, we found 10 SDKs do not follow industry standards for taking credit card data from users. Finally, we found 26 SDKs use WebViews, of which 20 allow JavaScript, 8 allow the JavaScript bridge, 23 allow local file access, and 21 do not clear the WebView cache. These uses of WebViews introduce an unnecessary attack surface. Our findings underscore the importance of increasing the minimum security standards to which payment SDKs are held, as well as the need for automatable methodologies for validating the compliance of such applications.

We make the following contributions in this paper.

- *We propose the AARDroid static program analysis framework for testing OWASP MASVS requirements on Android application SDKs.* Our program analysis checks span four categories of MASVS requirements and are capable of automatically analyzing the SDKs without relying on full applications.
- *We propose techniques for tailoring dataflow analysis for SDKs.* Our framework packages SDKs into applications and derives the semantics of security-relevant APIs using a domain-specific ontology of security-sensitive terms.
- *We use AARDroid to study 50 payment SDKs for Android applications.* We find that nearly three-quarters of the SDKs fail to meet at least one MASVS-L1 requirement. We also uncover concrete security weaknesses in many of the payment SDKs.

Conference acronym '22, December 05–09, 2022, Austin, TX
2022. ACM ISBN 978-1-4503-XXXX-X/22/12…$15.00
https://doi.org/XXXXXXX.XXXXXXX

[1] SDKs are packaged as Android Archive (`.aar`) files.

We note that security standards such as MASVS are designed to describe security best-practices for software development. Failure to meet requirements does not always mean that an application (or in this case, an SDK) is vulnerable. However, failures to meet best-practices can result in non-compliance with industry standards (e.g., PCI DSS) and indicate deeper flaws, as demonstrated in our empirical study. AARDroid provides an automated method of helping developers avoid these flaws.

The remainder of the paper proceeds as follows. Section 2 provides background and related work. Section 3 overviews our approach. Section 4 describes how we capture MASVS checks using program analysis. Section 5 details how we apply dataflow analysis to SDKs. Section 6 reports the results. Section 7 discusses limitations. Section 8 concludes.

2 BACKGROUND AND RELATED WORK

Security of payment systems has been studied exhaustively over the past decade beginning with payment card fraud at the ATM [1, 2, 10, 15] and later identifying novel solutions [55, 56]. Early studies identified flaws in the logic of Cashier-as-a-Service (CaaS) providers [63], which inspired further work that identified flaws in the business logic of e-commerce web applications [48, 61] and third-party payment services [65]. Automated identification of protocol vulnerabilities [13, 35, 46] also remains an active area of research.

The evolution of mobile payment systems has inspired the security investigation of financial applications [12, 37, 53, 58, 66] and e-wallets [29]. Yang et al. [67] and Shi et al. [57] studied several online payment service providers (cashiers), finding integration flaws and exploits in the SDKs. Chen et al. [12] identify sensitive input from user interfaces and apply static program analysis techniques to discover application vulnerabilities in banking apps. Prior work has also studied compliance with industry standards governing e-commerce websites [51] and mobile apps [37]; however, these studies have not investigated payment SDKs specifically.

Security analysis of non-payment applications is well-studied. Several lightweight static analysis scanners such as QARK [50], Androbugs [5], and MobSF[38] test for common security weaknesses in Android applications. However, their lack of dataflow analysis renders them either incapable of testing certain requirements or subject to high false alarm rates. Our work builds upon the knowledge gathered over the past decade by researchers studying Android application security. This work includes identification of sensitive data leakage [7, 17, 18, 32], flawed implementations of SSL/TLS [19, 20, 26, 41, 42, 49, 60], misuse of cryptographic primitives [16, 31, 39, 52], and identification of vulnerable WebViews [14, 36, 40, 59].

While many of these efforts touch upon aspects of OWASP's MASVS [43] requirements (and may have indeed inspired some of them), no prior work has sought to codify the MASVS requirements into program analysis tests. MASVS is considered the canonical definition of security-best practices for mobile application security. However, the requirements themselves are written in natural language, and while the accompanying Mobile Security Testing Guide (MSTG) [44] provides some technical detail, applying MASVS to an application is a largely manual process.

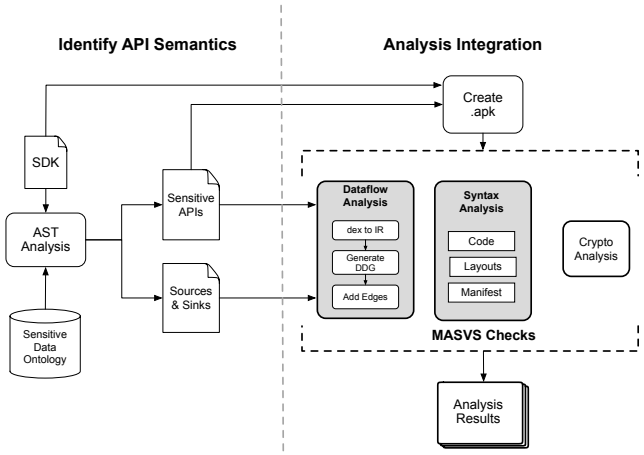

Figure 1: Overview of AARDroid analysis. The first phase identifies the security relevancy of APIs and arguments in the SDK, which are used by the dataflow analysis in the second phase. The second phase provides a suite of tests derived from OWASP's MASVS-L2 requirements for SDKs.

Threat Model: Failure to adhere to MASVS does not always imply serious security vulnerability. For concreteness, our threat model considers mobile applications accessing payment systems. Failure to comply with PCI DSS represents a significant risk for payment SDK providers and their customers. PCI DSS explicitly prohibits saving specific types of credit card data on the device, sharing it insecurely, or displaying it improperly. Historically, unnecessary storage of sensitive payment information has exacerbated security incidents (e.g., the Target security breach [11]). As such, we assume the attacker can access all device storage by either gaining physical access or installing malware. While different storage areas can be accessed more easily than others (e.g., SDcard storage is significantly more exposed than the app's private storage), PCI DSS does not differentiate where highly-sensitive information is stored. If credit card data is stored on the device, attackers targeting point-of-sale devices running payment applications can steal this data for many customers. For end-user smartphones, the attacker can distribute malware that collects credit card data from many devices. Finally, we assume both the SDK and application developers are benign.

3 OVERVIEW

The goal of our work is to enable automated analysis of security weaknesses in payment SDKs for Android applications. While both code security standards and program analysis tools exist for mobile applications, applying them to SDKs is nontrivial, presenting the following key challenges.

- *Security standards and tools are written with applications in mind.* It is not always clear whether the end app developer or the SDK developer is responsible for enforcing security requirements. Existing program analysis tools also assume full application packages with well-defined entry points.

- *Standards are written in natural language.* They typically reference technical artifacts at a high-level and do not provide procedures to programmatically validate an application against the prescribed guidelines.
- *The semantics of SDK application programming interfaces (APIs) are not easily inferred.* The knowledge of code entry points and security sensitivity of variables is necessary to verify program analysis requirements. Even if SDK documentation is available, such knowledge would need to be extracted from natural language.

We propose a framework called AARDroid (Figure 1) to address these challenges. AARDroid captures textual requirements in MASVS as static analysis tests. Section 4 describes how we determined which requirements are relevant to SDKs, as well as the one-time effort of how we translated the natural language requirements into program analysis checks. We capture four categories of MASVS requirements consisting of 28 checks: (1) data storage and privacy (**DS1**-**DS12**), (2) cryptography (**CRYPTO1**-**CRYPTO4**), (3) network communication (**TLS1**-**TLS4**), and (4) platform interaction (**PLAT1**-**PLAT8**). Our core analysis is built upon Argus-SAF [22] (formerly Amandroid) to perform the dataflow analysis required for five of the data storage and privacy checks. We also define a set of Argus-SAF modules for verifying MASVS requirements that can be expressed as syntax checks. Finally, we use CryptoGuard [52] to evaluate most of the the cryptography and TLS requirements.

A key technical contribution of our work is the process of adapting Argus-SAF to perform dataflow analysis on SDKs without accompanying applications. Manually creating test applications for each SDK is time consuming and error prone, particularly when available SDK documentation is incorrect or not available. Section 5.1 describes how AARDroid automatically creates wrapper applications for each SDK and extends the program analysis to create dummy edges from the application lifecycle methods in the wrapper project to security-relevant APIs within the SDK.

Determining which SDK APIs and arguments should be linked to the dataflow analysis is nontrivial. Section 5.2 describes how AARDroid starts with all public methods using parameters that are or contain strings (determined recursively). It further refines the set using the names of methods, parameters, and class fields, inferring the semantics with a domain-specific ontology adapted from PolicyLint's [4] ontology of data used by mobile app privacy policies. While this refinement does not work for obfuscated SDKs, over 80% of the payment SDKs we studied were not obfuscated.

We used AARDroid to automatically evaluate a set of 50 payment SDKs against our 28 MASVS program analysis finding that 37 SDKs fail to meet at least on MASVS Level 1 requirement. Section 6 discusses our manual investigation of the security weaknesses raised by AARDroid. We found that our approach to adapting dataflow analysis to SDKs resulted in few false positives. We also identified a range of significant concerns, including saving unencrypted credit card information to files, use of insecure cryptographic primitives, insecure input methods for credit card information, and insecure use of WebViews. These results demonstrate the value of applying security analysis at the SDK granularity.

4 INTERPRETING MASVS

OWASP MASVS is considered the canonical definition of best-practices for mobile application security. It defines two primary levels for compliance. MASVS-L2 (Level 2) is defined as appropriate for applications requiring PCI-DSS compliance [47]. As with other security standards, MASVS is written in natural language, typically referencing technical artifacts at a high-level. While the accompanying Mobile Security Testing Guide (MSTG) [44] provides some technical detail, applying MASVS to an application is a largely manual process. A key contribution is translating these natural language requirements into automated static analysis checks.

MASVS organizes security requirements into multiple categories. Overall, we implemented 28 MASVS checks for four categories: Data Storage and Privacy (12), Cryptography (4), Network Communication (4), and Platform Interaction (8). The remaining categories (Architecture, Authentication, and Code Quality) or checks were either not feasible to identify using static program analysis, not in the scope for payment SDKs, or too generic. The remainder of this section describes our approach to encoding the requirements.

Data Storage and Privacy (DS): We use a series of checks to understand how an SDK stores sensitive data. The system credential storage facility should be used when appropriate (**DS1**). External storage should completely be avoided (**DS2**) for reading or writing sensitive data, and it should never be sent to device logs (**DS3**). Instead of storing sensitive information on the device, developers should retrieve it from a remote server and avoid local persistence (**DS11**). If persistence is needed, encryption should be used (**DS12**). Furthermore, sensitive data should not be passed to any third party APIs (**DS4**) or inter-process communication (IPC) (**DS6**). The above checks largely use dataflow tracking and we discuss the identification of sensitive data (taint sources) in Section 5. We also include a number of configuration and syntactic checks. SDKs should disable Android's auto-backup feature (**DS8**), as it can persist data longer than necessary. Many SDKs also include XML user interface layout files to accept credit card input from the user. Our checks ensure that sensitive data cannot be leaked through the UI (**DS7**) and UI input cache (**DS5**). MASVS also suggests defense-in-depth measures that are not mandatory but provide better security if implemented. We include checks for disabling screenshots on sensitive UIs (**DS9**), checking if device has set up a pass-code (**DS10**), and protecting against screen overlay attacks (**PLAT7**).[2] These checks automatically determine if the SDK is querying specific Android API (i.e `isDeviceSecure()`); however, they require manual inspection to understand the context.

MASVS includes three requirements in this category that we did not encode, as listed in Table 12 (appendix). MASVS 2.10 specifies that applications should not persist sensitive data in memory longer then necessary. It is not specified what is considered long, and we only analyze usage of local storage but not system memory. MASVS 2.12 requires applications to educate user about what data is processed. MASVS 2.15 requires applications to wipe local storage after excessive failed authentication attempts; however, this action requires application context.

[2]We mention **PLAT7** in the data storage and privacy category due to its similarity to other checks. MASVS categorizes it as a platform requirement.

Cryptography (CRYPTO): Misuse of cryptographic primitives is a well known category of software vulnerabilities. Several prior works [16, 31, 39, 52] have shown that Android application developers continue to make these mistakes. AARDroid captures the MASVS cryptography requirements as follows. AARdroid ensures hard-coded or predictable cryptographic keys are not part of the SDK code (**CRYPTO1**) and proper cryptography configurations are used (**CRYPTO2**). Insecure configurations include insufficient key length, insecure modes (e.g., ECB Mode), predictable or static IV in symmetric ciphers, and insufficient number of iterations in PBE. MASVS also prohibits usage of deprecated cryptographic algorithms (**CRYPTO3**), including DES, IDEA, BlowFish, RC4, RC2, MD5, MD4, MD2, and SHA1. Finally, AARDroid checks if random number generators are correctly chosen and configured (**CRYPTO4**). The full list of checks is provided in Table 9 (appendix). Since CryptoGuard [52] already captures all four of these requirements, AARDroid simply adopts it into the analysis pipeline rather than reimplementing its logic. The remaining MASVS requirements in this category include self-implemented cryptography (MASVS 3.2) and using the same key for multiple purposes (MASVS 3.5). As these requirements require semantic knowledge of the application or SDK functionality, we did not include them.

Network Communication (TLS): HTTP and HTTPS are the *de facto* means of network communication for Android applications. Despite Android providing safe defaults, prior research [19, 20, 26, 41, 42, 49, 60] continues to identify Android applications that misconfigure TLS. AARDroid captures the MASVS network communication requirements (Table 10 in appendix) as follows. First, it ensures that only TLS network connections are used by the SDK (**TLS1**). Any URLs specifying the `http` scheme are flagged. AARDroid also ensures proper TLS configuration (**TLS2**) and that endpoint certificates are properly validated (**TLS3**). It detects if specific methods in the `SSLSocketFactory` class and `TrustManager` interface are improperly implemented. Finally, AARDroid checks for certificate pinning (**TLS4**) as a defense-in-depth requirement. It captures both the legacy approach using `TrustManagerFactory` and the recent approach using NSC file methods for TLS certificate pinning. With the exception of **TLS4**, all the checks use CryptoGuard [52], as we are already running CryptoGuard for the cryptography tests. We did not evaluate MASVS 5.5, as it relates to account enrollment and account recovery, which is usually not a feature of payment SDKs. We also did not include a check for the versions of the SDK's dependencies (MASVS 5.6) as that would require profiling a large dataset of dependencies.

Platform Interaction (PLAT): The MASVS platform category specifies a collection of requirements to ensure Android's feature-rich runtime environment is used securely. First, the SDK should not ask for unnecessary permissions (**PLAT1**) as the permissions in the SDK manifest file propagate to the application. Next, the SDK should not export sensitive functionality through IPC (**PLAT3**) or custom URL schemes (**PLAT2**). SDKs embed WebViews for accessing web content (e.g., for payment transactions). WebViews are a well-known attack surface [14, 36, 40, 59]. Any use of a WebView should be combined with security-conservative configuration, including disabling JavaScript (**PLAT4**), specifying a minimum set of protocol handlers (**PLAT5**), not exposing a JavaScript bridge (**PLAT6**), and clearing web cache before closing (**PLAT8**). AARdroid performs these checks using syntax analysis. These checks are listed in Table 11 (appendix). We do not capture input validation (MASVS 6.2). If user input is sent to a network service, it should be checked server-side. Local injection attacks can only occur if the SDK uses an SQLite database, which none of our studied payment SDKs do.

5 ENABLING SDK DATAFLOW ANALYSIS

Dataflow analysis of Android applications is a well-studied problem. Most recent work builds upon either FlowDroid [7] or Argus-SAF [22] (formerly known as Amandroid), because they capture many aspects of the Android application life-cycle, components, and other framework-specific interfaces. Unfortunately, both FlowDroid and Argus-SAF assume they are analyzing a `.apk` file. This assumption is deeper than the packaging and filename: renaming a `.aar` file as `.apk` is not sufficient. Both tools use the semantics of Android component objects (e.g., activities) when constructing dataflow graphs. Furthermore, the sources and sinks for their dataflow analysis uses well-known Android framework APIs. Neither assumptions holds true for the studied payment SDKs. This section describes how we overcame these analysis challenges.

5.1 SDK Analysis

The goal of this subsection is to adapt Argus-SAF to perform a dataflow analysis of the `.aar` Android Archive file of an SDK. The trivial approach is to manually create an application project for each SDK, including the `.aar` file and using the available documentation to write code that exercises the SDK functionality. However, this approach as several shortcomings. First, it does not scale. Section 6 studies 50 SDKs, and manually creating this many applications would be very time consuming. Second, we found the documentation for some SDKs to be very poor, if it is available at all. We did not want to limit our analysis to only those SDKs with clear documentation. Third, even when documentation is available, it may be incorrect or incomplete. In such cases, our analysis may miss APIs used by real applications.

AARDroid automatically constructs `.apk` files by combining each `.aar` SDK file with a simple template Android application project. While the dataflow results for the `.apk` will include several default Android libraries (e.g., `android` and `androidx`), these can be easily filtered out using package names. The more difficult challenge is ensuring the code for the SDK under test is included in the dataflow analysis. Argus-SAF performs dataflow analysis by first constructing interprocedural control flow graphs (ICFG) using points-to analysis and then using def-use analysis to derive an interprocedural data dependency graph (IDDG). Dataflow analysis is performed by traversing the IDDG. The ICFG analysis begins from predefined entry points in an application (e.g., the `onCreate()` of an activity component). If the code in the SDK is not called by the application template, it will not be included.

We modified Argus-SAF to insert dummy call statements to SDK APIs into the intermediate representation (IR) before constructing the ICFG. The structure of the generated IR code for the dummy API calls depends on different API access modifiers. For example, the object of the corresponding class is included as the first parameter in the IR code for non-static functions but not static functions.

Our implementation is configurable using an input file generated by the analysis described in Section 5.2. Next, Argus-SAF defines timeouts for each component (rather than the application as a whole). To ensure the API invocations do not impact one another, we encapsulate each API call of interest in its own component instance (e.g., activity component). We did not encapsulate multiple API calls in a single component, because we found that doing so can result in analysis time-out, skipping the subsequent calls.

Another challenge for instrumenting dummy calls was initializing the API parameters with only the context of the function prototype. Dummy calls without parameter initialization can prevent Argus-SAF from including the call in the IDDG used for dataflow analysis. For complex objects, this initialization needs to be performed recursively. Given the poor and limited documentation for many studied SDKs, automating initialization was infeasible. Fortunately, Argus-SAF does not require parameter initialization for dataflow analysis f the call is a taint source. Therefore, we annotate each API call as a taint source to avoid this limitation.

SDK APIs also sometimes have dependencies upon one another. For example, method $f_a(\cdot)$ may take the string of a credit card number and return an object o. The SDK may be designed such that o should be passed to another method $f_b(\cdot)$, which performs a vulnerable execution involving the credit card number. We avoid the need to consider such dependencies by recursively determining the semantics of taint sources, as discussed in Section 5.2. That is, our approach allows us to consider each API call in isolation.

5.2 Identifying Relevant APIs

Dataflow analysis requires knowledge of taint sources of interest. Simply creating dummy calls to all APIs in the SDK is likely to result in long analysis times and very noisy results. We refine the set of APIs in two ways. First, since sensitive payment information is primarily string data, we recursively determine which APIs take strings as arguments. Second, we further refine and semantically annotate the APIs and their parameters by considering their names.

5.2.1 Identifying APIs Consuming Strings. Payment information such as credit card numbers and CVC codes are commonly stored as strings within applications and SDKs. Strings may be passed directly as a `String` object or indirectly as a `String` object that is nested within another object. We refine the set of relevant APIs by identifying which ones directly or indirectly consume a string.

We identify string consuming APIs by using ASM [8] to generate and traverse the abstract syntax tree (AST) of the SDK binary. Our algorithm determines the set of relevant methods R by iteratively inspecting each `public` method in the AST, as apps can only call `public` methods in the SDK. We traverse the AST using a breadth first search. For each method m, we inspect all parameters p individually. If a parameter p is a `String`, m is added to R along with the relevant parameter index. If a parameter is an object, we expand the object, inspecting all member variables of the class and the super class. If a member variable is a `String`, m is added to R with the relevant parameter index and member variable. If a member variable is an object, we repeat the process, recursively resolving objects to find `String` variables. Thus a method m is added to R with information on which of its parameters $\{p\}$ are `String` or an object containing `String`. We need to keep information about the API parameters for precise taint tracking.

We exhaustively search all parameters for `String` variables. First, we need to identify all possible taint sources. Second, we need to extract the *name* of the variable for the semantic resolution completed by the second phase of our refinement.

5.2.2 Determining Security Semantics. Refining the set of APIs to only the set that consumes strings still produces very noisy dataflow analysis results, as confirmed by tests with a handful of the SDKs studied in Section 6. Therefore, AARDroid uses the names of relevant methods and their `String` variables to further refine the set of taint sources. Assigning semantics to names serves two purposes. First, it eliminates sources that are not relevant to our security analysis. Second, it differentiates the sensitivity of data. For example, storing a credit card number to a file is a higher security risk than storing the user's name (see Section 5.3).

One disadvantage of this approach is that it does not work well for obfuscated SDKs. Fortunately, 41 of the 50 SDKs studied in Section 6 were not obfuscated. We were unable to perform the dataflow checks for the nine obfuscated SDKs.

Name Extraction: We extracted the names of methods, parameters, and member fields during AST traversal (Section 5.2.1). Method names and field names are present in the descriptor of that element. However, extracting method parameter names required further parsing of the SDK binary. Method parameter names are present in the constant pool section of the binary *only if* the SDK is not built with specific build flags. Of the 50 SDKs studied in Section 6, only the nine obfuscated SDKs used these build flags.

Method vs. Parameter Names: Our filtering prefers parameter names over method names. We found that method names tend to be more ambiguous and long, requiring more textual analysis than parameter names. We also found that parameter names more frequently describe the data of interest. For example, consider the API prototype `public void processTransaction(String creditCardNumber)`. Resolving "processTransaction" requires additional context to determine the type of "transaction." In contrast, "creditCardNumber" is clearly payment information. In general, we found many method names to be too generic (e.g, `process()`, `isValid()`, `onCreate()`, `onSuccess()`, `init()`). Along with parameter names, we also used recursively derived names of `String` fields within objects.

Ontology Adaption: Data ontologies are commonly used to resolve the semantics of a word. We started with the PolicyLint [4] ontology, which was generated by combining the subsumptive relationships extracted from the privacy policies of over 10,000 Android appliations. This ontology provides a hierarchical structure of *is-a* relationships between two data objects (e.g., a "credit card number" is a "payment information"). The PolicyLint ontology has 459 internal nodes (e.g., "credit card security code") and 3,235 leaf nodes (e.g., "cvv code," "cvv," "card cvv").

We evaluated the suitability of the PolicyLint ontology using the parameter names of all public APIs within the non-obfuscated SDKs in our dataset. To assess suitability, we extracted the parameter names from all 41 SDKs and then manually removed names that were obviously irrelevant, e.g., parameter names consisting of arbitrary letters or words (e.g., "arg," "var," "value," "param")

or variable names that do not describe sensitive information (e.g., "level," "attribute," "activity"). Next, we calculated what percentage of the refined list exists in the data ontology. While most of the refined parameter names (74%) were captured, we systematically extended the PolicyLint ontology by adding 492 leaf nodes. Our final ontology consisted of 3,727 leaf nodes.

5.3 Capturing MASVS Dataflows

Each MASVS dataflow check requires identifying tainted paths from API sources to specific sets of sinks. AARDroid uses two types of dataflow analysis. **DS3**, **DS4**, **DS6**, and **DS11** all use a traditional form of taint tracking to determine if sensitive data passed to relevent API makes its way to specific sinks (e.g., `Log` classes). However, other checks require finding invocation of an intermediate method along the path to the taint sink. Specifically, **DS12** requires identifying a data encryption method (`Cipher.doFinal()`) along the path data storage. We capture this path-constraint on the dataflow using technique similar to Cardpliance [37].

Many of the MASVS requirements only apply for highly sensitive data types. We use our adapted PolicyLint ontology to classify sensitivity. To define sensitivity, we needed to determine at what granularity to consider is-a relationships. For example, "credit card number" and "card expiry date" are both children of "payment information." However, "card expiry date" is not as sensitive as "credit card number" or "cvc." Therefore defining "payment information" as highly sensitive fails to make this distinction. In contrast, there are too many leaf nodes to manually classify. In the end, we choose the parents of the leaf nodes to be a reasonable classification level. Our ontology included 170 such nodes, which we manually classified into high, medium, and low.

We defined highly sensitive information to be information such as credit card numbers, CVC, bank account numbers, SSN, passport, biometrics, TIN, and user credentials. We defined medium sensitivity to be information such as phone number, account information, IMEI, contact information, and date of birth. We defined low sensitivity to be the remaining values, including information such as URL, device information, country, city, and gender. Table 13 in the appendix lists our categorization.

6 EVALUATION AND RESULTS

Given that payment SDKs are integrated into thousands of apps, security weaknesses could impact millions of users. This section first describes how we identified our dataset. We then evaluate AARDroid's accuracy using manual validation. Finally we present our key security findings and supporting evidence.

6.1 Experimental Setup

Dataset: There is no centralized distribution platform for mobile app SDKs. However, Google Pay [27] and Apple Pay [6] allow many payment service providers to incorporate their service in addition to credit card payments. The documentation for Google Pay and Apple Pay gave us a combined list of 145 PSP names. However, many of these PSPs are only supported for web and not mobile. Many of the 145 PSPs simply used Google Pay's tokenization facility rather providing their own SDK for credit card processing. Many PSPs did not have SDKs publicly available on their website.

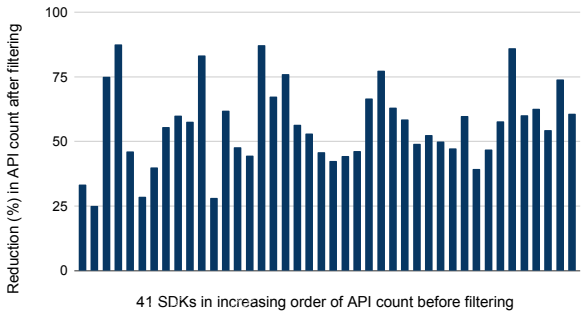

Figure 2: Reduction (percentage) of API count after applying filtering algorithm that incorporate the data ontology for the 41 SDKs without obfuscation.

In total, we gathered 50 Android payment SDKs from either GitHub or an artifact repository (e.g., JCenter, Maven) referenced from the PSP website. These 50 SDKs were collected during September and October 2020. Of these 50 SDKs, we could not extract the API semantics for nine SDKs, which limits the dataflow analysis. Additionally, 24 SDKs did not define a user interface and only processed payment data passed on from the application.

Analysis Runtime: Our analysis was run on a Apple MacBook Pro (2017) with a 3.1 GHz Dual-Core Intel Core i5 processor and 16 GB 2133 MHz LPDDR3 RAM running macOS Big Sur (Version 11.1). The analysis was run on each SDKs sequentially without parallelization. In total, the analysis of 28 MASVS checks on the 50 SDKs took 6 hours and 22 minutes. The average runtime per SDK was under eight minutes.

6.2 Sensitive API Identification

API Reduction: We compared the number of identified sensitive API sources before and after using ontology-based filtering (Section 5.2). Due to the linear relationship between analyzed APIs and runtime, this reduction also impacts the performance. Figure 2 shows the distribution of reduction. The number of initial APIs ranged widely from 6 (sberbank) to 588 (dotpay).

Accuracy: We selected five (≈12%) of the unobfuscated SDKs to measure the accuracy of our sensitive API identification heuristic. For each SDK, we determined if the APIs filtered out by the heuristic are non-sensitive. As shown in the false negative (FN) column of Table 1, 14 of the total 97 API of the inspected dataset were incorrectly eliminated. One source of false negative was passing sensitive data using generic names (e.g., input). We also found SDK-specific clauses being used as parameter names (e.g., squareMerchantId). In the set of studied SDK, we found there were no zero-argument APIs performing sensitive functionality. Finally, of the 37 candidate API that were marked as sensitive, only 5 were not sensitive (false positives). The false positives resulted from the use of sensitive words (e.g., code, id) to pass insensitive values to functions.

6.3 Detection Accuracy

We evaluated the accuracy our MASVS checks by manually inspecting the raised alarms. Given that a key contribution of this paper is

Table 1: Accuracy of sensitive API identification

SDK	Total API	FN	API Count After Filtering	API Reduction	TP	FP
redsys	15	0	9	40%	7	2
simplify	18	3	8	55%	8	0
tranzzo	26	2	9	43%	7	2
square	30	6	5	84%	4	1
payjp	47	3	6	88%	0	0

Table 2: Tool positive and true positive of different Code and XML checks that require identification of specific primitives.

Check	DS1	DS2	DS5	DS7	PLAT2	PLAT3
Reported	12	9	3	16	2	3
True Positive	12	7	1	10	2	3

Table 3: Tool Positive and True Positive for checks in Crypto (CR) and TLS category performed by CryptoGuard.

Check	CR1	CR2	CR3	CR4	TLS1	TLS2	TLS3
Reported	5	2	13	6	1	6	6
True Positive	1	2	11	2	0	0	0

the dataflow analysis and that we rely on CryptoGuard for cryptography checks, a key focus of our evaluation is the accuracy of the dataflow analysis. For the dataflow analysis, we measure accuracy at the *SDK level* for each sensitivity level (high, medium, and low). That is, we did not manually inspect every reported data flow path, as this was sometimes in the thousands. A summary of our analysis results after manual inspection is shown in Table 7 (appendix).

Manual Validation Process: We manually reviewed the SDK code to determine whether each raised alarm was a true or false positive. Integrating the SDK into a test application was infeasible for a variety of reasons, including a lack of documentation and the ability to perform real transactions. The manual validation was performed by three of the authors of the paper, each having sufficient academic and industry experience in Java and Android programming. Each validation was performed by one author, cases deemed ambiguous were resolved by a group discussion. We used a combination of the JD-GUI [28] and FernFlower [23] decompilers to obtain the Java code of SDK binaries.

The validation process depended on the type of program analysis check. For code syntax checks where presence of a primitive with particular properties is problematic (e.g., `setJavaScriptEnable(true)`), we confirmed if that primitive is actually present, and if it could be invoked from a public API. AARDroid also has a few checks that parse specific properties of UI widgets (e.g., `EditText`) in layout files. However, UI widget properties can also be set on the encapsulating View object in Java code, which was out of scope of our analysis. For validating these checks, we inspected the XML layout code and rendered it in the Android emulator (when possible). We also inspected the Java code referencing objects for any additional properties. When validating CryptoGuard's identification of cryptographic API misuse and TLS vulnerabilities, we examined the security context of the offending code to verify its reachability.

Our validation process for MASVS checks requiring dataflow analysis was slightly different. Unlike code syntax analysis where the tool reports in which file the vulnerable API is present, dataflow analysis reports all the data flows associated with a specific API sink. In some cases we found thousands of such instances for which manual validation was infeasible. If there was at least one true positive flow for an SDK for a given sensitively level, we reported AARDroid as having a true positive for that sensitivity level.

For each inspected dataflow, we first considered the semantics linked to the API source to determine if the data sensitivity (i.e., high, medium, low) reported by AARDroid was accurate. This step evaluates ambiguity due to the data ontology (e.g., due to generic terms such as key, data, account). We then traced through the source code to determine if the path reported by AARDroid exists. If any of the identification was incorrect, we marked the flow as erroneous and moved on to validating the next. We note that validating **DS4** was slightly different than the other dataflow checks, as it required also validating if the sink is third party.

Results: Table 4 presents our results for the dataflow checks, grouping alarms by sensitivity and reporting a true positive if there eixsts any dataflow for that sensitivity. In total, AARDroid reported 51 dataflow alarms across 17 payment SDKs. We identified five false positives, which resulted from misclassification of ambiguous parameter names (e.g., "account" was interpreted as financial account information) and incorrect identification of third party URLs. For this SDK-level measurement, AARDroid's dataflow-based checks had greater than 90% precision. Tables 2, 3, 5, and 6 report AARDroid's accuracy for the non-dataflow checks.

6.4 Security Weaknesses

MASVS defines two primary levels of requirements: L1 and L2. L1 is the bare minimum for mobile application security. MASVS specifically states that L2 is appropriate for payment applications seeking to meet PCI-DSS requirements. While we expected that SDKs would not meet defense-in-depth requirements, we expected payment SDKs to meet bare minimum requirements. However, we found that of the 50 payment SDKs, 37 failed to meet at least one MASVS-L1 requirement.

Our study identified the following key findings.

- 3 SDKs save unencrypted sensitive financial information such as credit number and CVC to either persistent storage or device logs. A fourth SDK stores a CVC in encrypted form. PCI-DSS standards prohibit such storage.
- 11 SDKs rely on outdated cryptographic primitives for sensitive functionality.
- 10 SDKs do not follow industry standards for taking credit card data from users.
- 26 SDKs use WebViews, of which 20 allow JavaScript, 8 allow the JavaScript bridge, 23 allow local file access, and 21 do not clear the WebView cache. These uses of WebViews introduce an unnecessary attack surface.

This section discusses the MASVS failures in detail. Table 7 (appendix) provides an overall picture of the MASVS failures. Figure 3 (appendix) shows the number of SDKs failing each test.

6.4.1 Data Storage and Privacy Violations. We categorize the MASVS Data Storage and Privacy requirements into three types of failures:

Table 4: Results of Data Flow checks for Data Storage and Privacy requirements. A (✓) on a cell means that MASVS requirement has been violated or True Positive. (✗) means it was reported by the tool but is False Positive.

PSP	DS-3			DS-4			DS-6			DS-11			DS-12		
	H	M	L	H	M	L	H	M	L	H	M	L	H	M	L
bluesnap	✓	✓	✓		✗										
cardconnect	✓	✓													
computop	✗		✓												
datatrans		✓	✓												
dotpay	✓	✓	✓							✓	✓	✓	✓	✓	✓
eghl			✓												
eway		✓													
ingenico		✓	✓		✗						✗			✗	
paragon	✓		✓												
paymentwall		✓													
payu											✓			✓	
platon										✓	✓		✓	✓	
razer			✓												
stripe												✓			✓
tappay		✓	✓								✓	✓		✓	✓
wepay	✓	✓	✓								✓			✓	
xpay		✓	✓												

```
1    public void addCreditStoreSecurityCode(String credit_card_security_code)
       {
2        try {
3            String secure_code = this.RSAEncrypt(credit_card_security_code);
4            L.e("secure_code " + secure_code);
5            this.sharedPreferences.edit().putString("
       credit_card_security_code", secure_code).commit();
6            this.setOneClickCVVDataAvailable(true);
7        } catch (NoSuchAlgorithmException var3) {
8            var3.printStackTrace();
9
10   }
11 }
```

Listing 1: Code snippet of persisting encrypted CVC in Dotpay

```
1
2    ...
3    KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
4    keyPairGenerator.initialize(1024);
5    KeyPair keyPair = keyPairGenerator.genKeyPair();
6    PublicKey publicKey = keyPair.getPublic();
7    PrivateKey privateKey = keyPair.getPrivate();
8    this.sharedPreferences.edit().putString("publicKey", Base64.
       encodeToString(publicKey.getEncoded(), 2)).commit();
9    this.sharedPreferences.edit().putString("privateKey", Base64.
       encodeToString(privateKey.getEncoded(), 2)).commit();
10   ...
```

Listing 2: Code snippet of insecurely instantiating and storing RSA key for CVC encryption

sensitive data persistence, improper UI design, and lack of defense-in-depth measures.

Sensitive Data Persistence: Table 4 overviews the results of the five dataflow checks. Overall, our tool reported 17 distinct SDKs persisting data with different sensitivity levels.

Payment SDKs inherently process sensitive data, therefore the necessity of any such data persistence should be questioned. PCI-DSS strictly prohibits persistence of sensitive authentication data: "*PCI-DSS 3.2: Do not store sensitive authentication data after authorization (even if it is encrypted).*" For **DS11** (data persistence) and **DS12** (encrypted data persistence), we found six SDKs persisting data locally, two of which were highly sensitive data. For example, the DotPay SDK writes the credit card number and merchant ID to a Shared Preference file without encryption. The SDK also writes the CVC, but uses RSA encryption for doing so, as shown in Listing 1. Not only is storing an encrypted CVC a violation of PCI-DSS, it uses an 1024-bit RSA key, which is deprecated. To make matters worse, Listing 2 shows how the SDK stores both the public and private key in Shared Preferences, allowing an attacker to retrieve the private key to decrypt the CVC. Overall, the DotPay SDK violated four MASVS requirements: **DS1**, **DS11**, **DS12**, and **CRYPTO2**.

Logging of sensitive data (**DS3**) was a more frequent practice than local storage persistence. We discovered a range of logged sensitive data, including credit card numbers, CVC, IMEI, public keys, payment tokens (e.g., for Google Pay, Samsung Pay, Merchant's PayPal), MAC addresses, and URLs. In total 14 SDKs log critical data in ways that suggest that developers are unaware this is an insecure practice. For example, we found that users' credit card data (e.g., credit card Number, CVC, and expiry date) was recorded in device logs when (1) updating text fields [CardConnect], (2) updating databases [Dotpay], (3) initiating server requests [Paragon] and (4) potentially handling exceptions [WePay]. We did not identify any true positives for **DS4** (third-party leak) and **DS6** (IPC leak).

The checks not involving dataflow also uncovered several interesting findings. We found seven instances where data was accessed from external storage (**DS2**). Although the read data was not sensitive, reading from insecure storage puts the application at risk. Interestingly, we found that in many cases these reads were invoked by external libraries integrated within the SDKs. Next, we found that only three SDKs turned off Android's automatic backup of user data (**DS8**). Android automatically backs up user data, which could potentially leak if sensitive data is persisted or logged. As we discuss in Section 7, feedback from many SDK vendors indicate that they consider this requirement as the application's responsibility, and hence a gray area for classification as a security weakness. We also found that 12 SDKs use the software implemented Java KeyStore facilities for various purposes (e.g., storing certificates, storing RSA keys), however, *no SDKs* use the Android Keystore (**DS1**). Software implemented key stores persist credentials on app storage and could be vulnerable [24]. In contrast, the Android KeyStore provides better security by storing keys within specialized hardware (if available).

Improper UI Design: We found 26 SDKs provide a UI for accepting credit card numbers from users. Payment SDK developers should be aware of industry policies and regulations associated with designing a payment UI [43, 45, 47].

MASVS defines guidelines (**DS5**, **DS7**) for UI design, primarily to address shoulder surfing. **DS5** includes turning off keyboard auto-suggestion so that credit card numbers do not accidentally pop up. In Android, auto-suggestion is turned off by default for `number` fields; however, this default does not apply if the input method is set to `text`. We found the JudoPay SDK takes credit card number as text, and fails to turn off keyboard suggestions, which could result in leaking credit card numbers through the keyboard cache.

The MASVS requirements (**DS7**) also specify that if sensitive data is taken as input, it should not be exposed (e.g., to prevent shoulder surfing). While PCI-DSS is not clear whether or not CVC codes should be masked, MASVS descriptions imply it should [43, 45]. We found that even widely used payment SDKs such as Stripe fail to

mask the CVC code up on entry. PCI-DSS requires credit card fields to partially or fully mask the credit card number before displaying. It is unclear if this requirement applies for user inputs. Many of the SDKs we reviewed did mask the credit card number during entry. However, from usability perspective one might argue displaying the full number during entry is desirable. Therefore, our tables do not include not masking it during entry as a security weakness.

AARDroid identified 16 SDKs with a UI privacy issue for either the CVC, credit card, or both. We determined three of those were false positives for improper identification. Of the remaining SDKs, four had only the CVC field displaying the numbers after input, three had only the credit card field taking input without masking any part of it, and six had both. However, we conservatively mark not masking credit card numbers as false positives as mentioned above. The remaining 10 SDK displaying CVC cases are points of concern. These insecure UI designs increase the risk of exposing user's sensitive financial data.

Lack of Defense-in-Depth Measures: MASVS proposes a set of defense-in-depth measures for added security for sensitive applications. Disabling screenshot when the UI is taking sensitive input (**DS9**) can prevent leaking data to malicious services on rooted devices [12] or adb enabled device without root access [33]. Android also uses a screenshot-like feature when the application is backgrounded for use in the recent applications menu. We found only two SDKs (ACI and WireCard) have a UI that implements screenshot disabling, while the other 24 remain vulnerable to such sophisticated data leak attacks. None of the payment SDKs consider the device access security policy requirement (**DS10**) or take preventive measures against screen overlay attacks (**PLAT7**).

6.4.2 Cryptography violations. AARDroid reported that 5 SDKs use hard-coded keys (**CRYPTO1**), 2 use wrong configurations (**CRYPTO2**), 13 SDKs use deprecated algorithms (**CRYPTO3**), and 6 SDKs use insecure pseudo-random number generators (**CRYPTO4**). We manually inspected the corresponding code to understand the context in which these cryptographic primitives are used. The results of this manual inspection is reported in Table 3.

Of the six SDKs using insecure random number generators (**CRYPTO4**), only the Datatrans and CardConnect SDKs use them for sensitive operations (e.g., initializing payment objects). Additionally, only the CardConnect SDK violates **CRYPTO1** by using a hard-coded key in symmetric (CBC) encryption decryption schemes. The other instances were false positives, mostly due to finding no usage of the code. Both of the **CRYPTO2** checks (misconfiguration) were true positives, including using a 1024-bit RSA key (DotPay) and insufficient iteration count for PBE (WireCard).

We found 11 true positive instances of deprecated cryptography (**CRYPTO3**). These instances primarily involve the use of DES, Blowfish, SHA1, and MD5. From our verification, all cases are reachable through the code. Eight of the instances relate to hashing and integrity. The remaining three directly relate to encryption.

The use of MD5 or SHA1 to hash payment and merchant information is prevalent throughout the findings. Both are deprecated due to collisions. This use is especially concerning for financial transactions, which are both highly sensitive and unique. While exploitable collisions for SHA1 are relatively recent, MD5 has been shown to be completely compromised for quite some time [30, 64].

```
1
2     ...
3     String var2 = Secure.getString(this.c.getContentResolver(), "android_id")
          ;
4     byte[] var3 = var2.getBytes();
5     de.wirecard.paymentsdk.helpers.a.a(var3, (byte)-48);
6     byte[] var4 = var1 != null ? var1.getBytes("utf-8") : new byte[0];
7     SecretKeyFactory var5 = SecretKeyFactory.getInstance("PBEWithMD5AndDES");
8     SecretKey var6 = var5.generateSecret(new PBEKeySpec((new String(de.
          wirecard.paymentsdk.helpers.a.a(var3, (byte)35))).toCharArray()));
9     Cipher var7 = Cipher.getInstance("PBEWithMD5AndDES");
10    var7.init(1, var6, new PBEParameterSpec(Secure.getString(this.c.
          getContentResolver(), "android_id").getBytes("utf-8"), 20));
11    return new String(Base64.encode(var7.doFinal(var4), 2), "utf-8");
12    ...
```

Listing 3: Code snippet showing encryption function for SharedPreferences Editor in Wirecard

The Tpay, WePay, Platon, and Yandex SDKs all use MD5 hashes for sensitive information. Platon uses it for all encrypted returns. SHA1 is used for similar sensitive transactions.

Two SDKs use DES for sensitive encryption. In the Paypal SDK, the use of DES is alongside other more robust cryptographic methods. Presumably the DES support exists for legacy purposes. However, the code is still reachable. Finally, the Wirecard SDK uses both MD5 and DES. Specifically, it obfuscates shared preference values using `PBEWithMD5AndDES`. Just as concerning, `ANDROID_ID` is the source of randomness for key generation. Listing 3 displays this functionality. This finding raises multiple concerns. First, both MD5 and DES are insecure algorithms. Additionally, PBE is configured with 20 iterations, well below the suggested 1000.

6.4.3 Network communication violations. AARDroid reported that one SDK uses HTTP (**TLS1**), six SDKs have improper TLS configuration (**TLS2**), and six SDKs do not properly validate certificates (**TLS3**). However, as reported in Table 3, our manual inspection identified that the context in which the corresponding code was used could not directly be attributed to a vulnerability.

One SDK uses `http` (**TLS1**), but the connection is used for insensitive purposes. The same six SDKs violate both **TLS2** and **TLS3**, and for each SDK the same code triggers both alarms. Specifically, the SDKs use a custom `X509TrustManager` implementation that (1) does not throw a `CertificateException` in the `check(Client/Server)Trusted` method, and (2) fails to default to known certificate authority lists in `getAcceptedIssuers`. A custom `X509TrustManager` implemented as such will accept all certificates.

We determined all six pairs of **TLS2** and **TLS3** are false positives due to various mechanisms prevent exposure at default configuration. For example, to be vulnerable the CyberSource SDK would need an Android SDK level less than 10; however, the SDK manifest specifies a minimum SDK level of 14. On the other hand, the Paragon SDK hard-codes an internal boolean variable as true, which prevents the vulnerability. The other four SDKs are not vulnerable to similar types of configuration.

While we mark these alarms as a false positives, it is theoretically possible for an application developer to introduce a configuration that causes the TLS connections to be vulnerable. For example, the Paragon SDK code includes a public setter method that allows the application developer to disable the TLS checks. For the CyberSource SDK, a developer could force an SDK version less than 10, causing TLS connections to be vulnerable. Similar types of misconfiguration would cause the TLS connections of the other four

Table 5: No SDK using WebViews was configured properly.

Property	JS Enabled (PLAT4)	JS Bridge enabled (PLAT6)	Cache not Cleared (PLAT7)	Access Flags Enabled (PLAT5)			
				File	Content	Universal Access from File	File access from File
Reported	23	8	26	23	23	1	0
Validated	20	8	21	23	23	1	0

SDKs to be vulnerable. However, rather than assume developer negligence, we conservatively classify these alarms as false positives.

Finally, **TLS4** checks certificate pinning, which is an enhanced security feature to prevent on-path attacks. We found that only 12 SDKs adopt this defense-in-depth approach.

6.4.4 Platform Interaction violations. We categorize the MASVS platform interaction requirements into three types of failures: misconfigured WebViews, excess privileges, and insecure IPC schemes.

Misconfigured WebViews: Android WebViews present a well-known attack surface if they are not configured securely [14, 36, 40, 59]. None of the 26 SDKs using WebViews configure them according to MASVS standards (Table 5). In most cases the purpose of the WebView was trivial (e.g., checkout or confirmation of payments).

JavaScript execution within WebViews is turned off by default to mitigate a class of web attacks [9, 36]. However, we found 20 SDKs explicitly enable JavaScript (**PLAT4**). In the offending SDKs for which we could extract the URL, we found that all pages loaded without JavaScript but raised errors warning of insufficient data. We manually inspected the loaded JavaScript for these pages and concluded that the JavaScript renders UI elements and handles sensitive data such as phone and credit card numbers. Eight SDKs enable the JavaScript bridge, which allows the JavaScript executing in the WebView to execute Java methods within the SDK. Five of these SDKs do not restrict navigation by implementing `shouldOverrideURLLoading()`. The bridge was used to receive status on the payment confirmation (e.g., was the payment completed). However this breaks Android's WebView sandboxing [36] and is considered insecure (**PLAT6**). Providing a native Android user interface instead of using WebViews would mitigate this.

By default, WebViews are granted file access privileges, allowing access to all files the application can access (including external storage), potentially exposing the SDK to known attacks [14]. We found only 3 of the 26 SDKs using WebViews restrict file access.

Finally, 21 of the SDKs with WebViews (16 of which have JavaScript enabled) do not clear the WebView cache when the WebView is closed (**PLAT8**). Not clearing the cache may leave sensitive information (e.g., WebView cookies) in memory longer than necessary.

Excess Privileges: Overprivileged applications are a recognized and prevalent concern for Android applications [21, 25, 54, 62]. Our tool examines SDK manifests to identify when permissions with the "dangerous" protection level are assigned. We manually determined if the permission is needed by locating Android API calls requiring target permissions. We found 16 dangerous permissions across the manifests of 9 SDKs. Four had no identifiable use within the decompiled SDK and are therefore considered to not be needed by the SDK. Of the 12 remaining, six appear to stem from libraries, although these are used. These results are shown in Table 6.

Insecure IPC or URL Schemes: We found three SDKs include exported Android components, primarily for checkout activities.

Table 6: 16 dangerous permissions requested by 9 SDKs

Permission	External Storage	Location	Camera	Phone State	Record Audio	Call Phone	Get Accounts
Reported	6	2	2	2	2	1	1
Used	5	2	2	2	1	0	0

Two of these define a custom URL scheme. There are two types of URL schemes used by exported components: app links and deep links. App links use `https` or `http` schemes and the domain is verified by the platform (e.g., `https://www.example.com`). Therefore when such URLs are triggered, the platform knows which application to launch. However, deep links use custom schemes and are not verified (e.g., `custom://www.example.com`). Therefore, deep links are at risk of hijacking, as one application can freely register another application's schemes [34] and the platform would let the user to choose between multiple options. We found both of the SDKs use deep links on sensitive components. Neither validate the input, which could make them vulnerable.

7 DISCUSSION

Limitations: AARDroid inherits the fundamental limitations of static program analysis. The tests include a combination of precise dataflow analysis and coarser-grained syntax-based heuristics. The MASVS requirements themselves are often heuristical in nature, implying"security weaknesses" rather than explicit vulnerabilities. Our encoding of the MASVS requirements into program analysis checks was best effort, and our implementation is incomplete in some ways (e.g., identifying networking libraries). Our approach of identifying sensitive APIs in SDKs relies on successful extraction of the API semantics, which are not always available. Our semantic inference for taint sources does not consider APIs that take no parameters. For example, an SDK could store sensitive data in a global or member variable that is used by a no-parameter API called directly by the application. We did not identify any such cases in the five SDKs studied in Section 6.2.

Our current implementation also relies on a domain-specific data ontology generated from studying privacy policies, which we further adapted for payment SDKs. While we experimentally found the ontology to be sufficient, we can not make claims about its completeness. Finally, implementation for resolving UI input semantics was limited to credit card data only. A generic tool could adopt sophisticated UI analysis tools [3].

Responsible Disclosure: For each of the 50 SDKs, we searched websites and SDK repositories for contact email addresses for developers. Contact information was unavailable for four SDKs. We emailed the remaining 46 SDKs, identifying the the SDK name, SDK version, source, timeline and the MASVS violations. For each MASVS violation, we reported where the violation occurred and why it was a violation with reference to the MASVS document. We received 27 acknowledgements of receipt, although 13 were auto-generated. Of the 14 non-auto-generated responses, 7 vendors claimed to pass the message to their engineering team internally for further investigation, 1 claimed to release patches, 1 claimed the violations less severe, and 1 claimed the SDK version was deprecated and informed affected merchants.

The remaining four vendors provided additional feedback on our findings. Although some findings were clear security violations

(e.g., persisting sensitive data), some requirements were disputed by the vendors about their applicability. One vendor claimed some of the findings (e.g., not masking CVC) were intentional business decisions. A few vendors disagreed that the SDK should implement specific protections indicating it is the application's responsibility, stating *"The backup flag is up to the host app to control. In general, we do not want to put additional burdens on integrators by setting flags that lead to non-standard, unexpected behavior on their side."* Finally, of the 14 responding vendors, 4 claimed the violations were identified because we analyzed an old version of their SDK. However, we re-ran the analysis on the most recent version of the SDKs and found similar issues. The vendors affirmed to resolve most of the security weaknesses in future releases.

8 CONCLUSION

Payment Service Providers have simplified mobile payment integration for application developers. However, security weaknesses present in payment SDKs can impact thousands of applications. In this work, proposed AARDroid for automated security assessment of payment SDKs with respect to OWASP's MASVS security standard for mobile applications. In addition to capturing MASVS requirements as program analysis checks, we demonstrated how to adapt such analysis to test an SDK without the presence of an accompanying application. We studied 50 payment SDKs, identifying widespread security weaknesses.

REFERENCES

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

[2] Ross Anderson and Steven Murdoch. 2014. EMV: Why Payment Systems Fail. In *Communications of the ACM*.

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

[4] Benjamin Andow, Samin Yaseer Mahmud, Wenyu Wang, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Tao Xie. 2019. PolicyLint: Investigating Internal Privacy Policy Contradictions on Google Play. In *Proceedings of the USENIX Security Symposium*.

[5] Androbugs Framework 2015. Androbugs Framework. https://github.com/AndroBugs/AndroBugs_Framework.

[6] Apple. [n.d.]. Apple Pay's Payment Service Provider List. https://developer.apple.com/apple-pay/payment-platforms/.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[8] ASM. [n.d.]. ASM: A Java bytecode manipulation and analysis framework. https://asm.ow2.io.

[9] Bhavani A B. 2013. Cross-site Scripting Attacks on Android WebView. In *International Journal of Computer Science and Network*.

[10] Mike Bond, Omar Choudary, Steven J. Murdoch, Sergei Skorobogatov, and Ross Anderson. 2014. Chip and Skim: Cloning EMV Cards with the Pre-play Attack. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[11] Brian Krebs. 2014. The Target Breach, By the Numbers. https://krebsonsecurity.com/2014/05/the-target-breach-by-the-numbers/.

[12] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. 2020. An Empirical Assessment of Security Risks of Global Android Banking Apps. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

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

[14] Erika Chin and David Wagner. 2013. Bifocals: Analyzing WebView Vulnerabilities in Android Applications. In *Proceedings of the International Workshop on Information Security Applications*.

[15] Tomi Dahlberga, Niina Mallata, and Jan Ondrusb. 2007. Past, present and future of mobile payments research: A literature review. In *Electronic Commerce Research and Applications*.

[16] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the ACM conference on Computer and Communications Security (CCS)*.

[17] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

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

[19] Sascha Fahl, Marian Harbach, and Thomas Muders. 2012. 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)*.

[20] Sascha Fahl, Marian Harbach, and Henning Perl. 2013. Rethinking SSL development in an Appified World. In *Proceedings of ACM conference on Computer and Communications Security (CCS)*.

[21] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*. 627–638.

[22] Xinming Ou Fengguo Wei, Sankardas Roy and Robby. 2014. 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)*.

[23] FernFlower 2021. Fernflower. https://github.com/JetBrains/intellij-community/blob/master/plugins/java-decompiler/engine/README.md.

[24] Riccardo Focardi, Francesco Palmarini, Marco Squarcina, Graham Steel, and Mauro Tempesta. 2018. Mind Your Keys? A Security Evaluation of Java Keystores. In *Proceedings of the ISOC Network and Distributed Systems Security Symposium (NDSS)*.

[25] Dimitris Geneiatakis, Igor Nai Fovino, Ioannis Kounelis, and Pasquale Stirparo. 2015. A Permission verification approach for android mobile applications. *Computers & Security* 49 (2015), 192–205.

[26] M Georgiev, S Iyengar, S Jana, R Anubhai, D Boneh, and V Shmatikov. 2012. 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)*.

[27] Google. [n.d.]. Google Pay's Payment Service Provider List. https://developers.google.com/pay/api.

[28] JD Project 2019. JD Project. https://java-decompiler.github.io/.

[29] Ratinder Kaur, Yan Li, Junaid Iqbal, Hugo Gonzalez, and Natalia Stakhanova. 2018. A Security Assessment of HCE-NFC Enabled E-Wallet Banking Android Apps. In *Proceedings of the IEEE Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 02. 492–497. https://doi.org/10.1109/COMPSAC.2018.10282

[30] Vlastimil Klima. 2006. Tunnels in Hash Functions: MD5 Collisions Within a Minute. *IACR Cryptol. ePrint Arch.* 2006 (2006), 105.

[31] S Kruger, J Spath, Karim Ali, Eric Bodden, and Mira Mezini. 2017. Crysl: Validating correct usage of cryptographic apis. *CoRR* (2017).

[32] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-component Privacy Leaks in Android Apps. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[33] Chia-Chi Lin, Hongyang Li, Xiao-yong Zhou, and XiaoFeng Wang. 2014. Screenmilker: How to Milk Your Android Screen for Secrets.. In *NDSS*.

[34] Fang Liu, Chun Wang, Andres Pico, Danfeng Yao, and Gang Wang. 2017. Measuring the Insecurity of Mobile Deep Links of Android. In *Proceedings of the USENIX Security Symposium*.

[35] Jiadong Lou, Xu Yuan, and Ning Zhang. 2021. Messy States of Wiring: Vulnerabilities in Emerging Personal Payment Systems. In *Proceedings of the USENIX Security Symposium*.

[36] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. Attacks on WebView in the Android System. In *Proceedings of the 27th Annual Computer Security Applications Conference*.

[37] Samin Yaseer Mahmud, Akhil Acharya, Benjamin Andow, William Enck, and Bradley Reaves. 2020. Cardpliance: PCI DSS Compliance of Android Applications. In *Proceedings of the USENIX Security Symposium*.

[38] MobSF 2015. Mobile Security Framework: MobSF. https://github.com/MobSF/Mobile-Security-Framework-MobSF.

[39] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. 2018. Source Attribution of Cryptographic API Misuse in Android Applications. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*.

[40] Patrick Mutchler and Adam Doupe†. 2017. A Large-Scale Study of Mobile Web App Security. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[41] Marten Oltrogge, Nicolas Huaman, Sabrina Amft, Yasemin Acar, Michael Backes, and Sascha Fahl. 2021. Why Eve and Mallory Still Love Android: Revisiting TLS (In) Security in Android Applications. In *Proceedings of the USENIX Security Symposium*.
[42] Lucky Onwuzurike and Emiliano De Cristofaro. 2015. Danger is My Middle Name: Experimenting with SSL Vulnerabilities in Android Apps. In *Proceedings of the ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*.
[43] OWASP. [n.d.]. OWASP's Mobile Application Security Verification Standard. https://github.com/OWASP/owasp-masvs.
[44] OWASP. [n.d.]. OWASP's Mobile Security Testing Guide. https://github.com/OWASP/owasp-mstg.
[45] OWASP. 2013. Handling E-Commerce Payments. https://wiki.owasp.org/index.php/Handling_E-Commerce_Payments#Displaying_portions_of_the_credit_card.
[46] Maurizio Panti, Luca Spalazzi, Simone Tacconi, and Salvatore Valenti. 2003. *Automatic Verification of Security in Payment Protocols for Electronic Commerce*. Kluwer Academic Publishers, USA, 276–282.
[47] PCI SSC. 2018. Payment Card Industry's Data Security Standard (PCI-DSS). https://www.pcisecuritystandards.org/documents/PCI_DSS-QRG-v3_2_1.pdf.
[48] Giancarlo Pellegrino and Davide Balzarotti. 2014. Toward Black-Box Detection of Logic Flaws in Web Applications. In *Proceedings of the ISOC Network and Distributed Systems Security Symposium (NDSS)*.
[49] Andrea Possemato and Yanick Fratantonio. 2020. Towards HTTPS Everywhere on Android: We Are Not There Yet. In *Proceedings of the USENIX Security Symposium*.
[50] QARK 2015. QARK: Quick Android Review Kit. https://github.com/linkedin/qark.
[51] Sazzadur Rahaman, Gang Wang, and Danfeng Yao. 2019. Security Certification in Payment Card Industry: Testbeds, Measurements, and Recommendations. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[52] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. 2019. CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-Sized Java Projects. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2455–2472.
[53] Bradley Reaves, Nolen Scaife, Adam Bates, Patrick Traynor, , and Kevin R.B. Butler. 2015. Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications in the Developing World. In *Proceedings of the USENIX Security Symposium*.
[54] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. 2012. Android Permissions: A Perspective Combining Risks and Benefits. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*. 13–22.
[55] Nolen Scaife, Christian Peeters, and Patrick Traynor. 2018. Fear the Reaper: Characterization and Fast Detection of Card Skimmers. In *Proceedings of the USENIX Security Symposium*.
[56] Nolen Scaife, Christian Peeters, Camilo Velez, Hanqing Zhao, Patrick Traynor, and David Arnold. 2018. The Cards Aren't Alright: Detecting Counterfeit Gift Cards Using Encoding Jitter. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.
[57] Shangcheng Shi, Xianbo Wang, and Wing Cheong Lau. 2021. Breaking and Fixing Third-Party Payment Service for Mobile Apps. In *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*. Springer, 3–26.
[58] Shangcheng Shi, Xianbo Wang, Kyle Zeng, Ronghai Yang, and Wing Cheong Lau. 2021. An Empirical Study on Mobile Payment Credential Leaks and Their Exploits. In *International Conference on Security and Privacy in Communication Systems*. Springer, 79–98.
[59] Wei Song. 2018. Understanding JavaScript Vulnerabilities in Large Real-World Android Applications. In *IEEE Transactions on Dependable and Secure Computing*.
[60] David Sounthiraraj, Justin Sahs, Zhiqiang Lin, Latifur Khan, and Garrett Greenwood. 2014. 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)*.
[61] Fangqi Sun, Liang Xu, and Zhendong Su. 2014. Detecting Logic Vulnerabilities in E-commerce Applications. In *Proceedings of the ISOC Network and Distributed Systems Security Symposium (NDSS)*.
[62] R Vinayakumar, KP Soman, and Prabaharan Poornachandran. 2017. Deep Android Malware Detection and Classification. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 1677–1683.
[63] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. 2011. How to Shop for Free Online–Security Analysis of Cashier-as-a-Service Based Web Stores. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.
[64] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. 2004. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. *IACR Cryptol. ePrint Arch.* 2004 (2004), 199.
[65] Luyi Xing, Yangyi Chen, XiaoFeng Wang, and Shuo Chen. 2017. InteGuard: Toward Automatic Protection of Third-Party Web Service Integrations. In *Proceedings of the ISOC Network and Distributed Systems Security Symposium (NDSS)*.
[66] Wenbo Yang, Juanru Li, Yuanyuan Zhang, and Dawu Gu. 2019. Security analysis of third-party in-app payment in mobile applications. *Journal of Information Security and Applications* 48 (2019), 102358. https://doi.org/10.1016/j.jisa.2019.102358
[67] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, and Dawu Gu. 2017. Show Me the Money! Finding Flawed Implementations of Third-party In-app Payment in Android Apps. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*.

A APPENDIX

Table 7 details the AARDroid results for all 28 MASVS tests on all 50 SDKs. Tables 8 to 11 list the AARDroid checks in detail, listing the MASVS IDs, check type, severity, and L1 or L2 classification. Table 12 lists the MASVS checks that were out of scope for this study. Table 13 lists the data-sensitivity classification of nodes in our topology. Figure 3 shows the counts of SDKs failing each checks.

Table 7: This table overviews the AARDroid analysis results of 28 MASVS requirement on 50 SDKs. A (✓) on a cell means that MASVS requirement has been violated or True Positive. (✗) means it was reported by the tool but is False Positive. (●, ◐, ○) means leakage of High, Morderate, Low sensitive data. (-) means that test was not applicable for the particular SDK. Empty cell means this MASVS requirement has not been violated. SDKs marked with (*) maintains all the MASVS-L1 checks that are applicable. Only Veritrans maintains all the MASVS-L2 checks that are applicable. Column colors indicate severity level: High Severity ■; Medium Severity □; Low Severity □

PSP-SDK	Version	**DS1**: No Android Keystore Usage	**DS2**: External Storage Usage	**DS3**: Logging	**DS4**: Third Party API Leak	**DS5**: Keyboard Cache Enabled	**DS6**: IPC Leak	**DS7**: UI Leak	**DS8**: Backup Allowed	**DS9**: Screenshot Enabled UI	**DS10**: Device Access Policy	**DS11**: Data Persistence	**DS12**: Data Encryption	**CRYPTO1** : Hardcoded Keys	**CRYPTO2** : Misconfiguration	**CRYPTO3** : Deprecated Crypto	**CRYPTO4** : Insecure PRNG	**TLS1**: HTTP	**TLS2**: Broken TLS	**TLS3**: Untrusted CA	**TLS4**: No Certificate Pinning	**PLAT1**: Dangerous Permission	**PLAT2**: Custom URL Scheme	**PLAT3**: Exported Component	**PLAT4**: JS Enabled	**PLAT5**: Unsafe protocol	**PLAT6**: JS Bridge Enabled	**PLAT8**: Cache Persisted	**PLAT7**: No Screen Overlay Test
2c2p	4.2.1	✓		-		-		-	✓	-	-	-	-													✓		✓	-
aci	2.2.0		✓	-				✗	✓		✓	-	-								✓	✗			✓	✓		✓	✓
authorize *	1.0.2					-		-	✓	-	-										✓				-	-	-	-	-
bambora	1.0.1					-		-	✓	-	-										✓				✓	✓	✓	✓	-
bluesnap	2.2.6			●	✗			✗	✓	✓	✓										✓				-	-	-	-	✓
braintree *	3.10.0					-		-	✓	-	-														-	-	-	-	-
cardconnect	3.0.71	✓	✓	●		✗		✗	✓	✓	✓			✓		✓	✓	✗			✓	✗			-	-	-	-	✓
checkout	3.0.1							✓	✓	✓	✓										✓				✗	✓		✓	✓
cloudpayment	1.0.7								✓	✓	✓										✓				✗	✓	✓	✓	✓
computop	1.5.2	✓	✓	◐					✓	✓	✓			✗		✓						✓	✓	✓	✓	✓		✓	✓
cybersource	1.0.0	✓				-		-	✓	-	-								✗	✗					-	-	-	-	-
datatrans	3.5.3	✓	✗	◐		-		-	✓	-	-			✗			✓		✗	✗					✓	✓		✓	-
dotpay	1.2.18	✓		●				✓	✓	✓	✓	●	●		✓	✗					✓	✓			✓	✓	✓	✓	✓
ebanx *	1.0.0					-		-	✓	-	-										✓				-	-	-	-	-
eghl	2.5.3			○				✗	✓	✓	✓						✗				✓				✓	✓	✓	✗	✓
eway	1.2.2			◐		-		-	✓	-	-										✓				-	-	-	-	-
fondy	1.11.1								✓	✓	✓										✓				✓			✗	✓
googlewallet *	18.0.0			-					✓			-	-								✓				-	-	-	-	✓
ingenico	5.2.0			◐	✗	-		-	✓	-	-	✗	✗								✓				-	-	-	-	-
ipay	1.0.5								✓	✓	✓										✓		✓	✓	✓	✓		✓	✓
judopay	5.7.1			-		✓		✓	✓	✓	✓	-	-								✓				✓	✓		✓	✓
mastercardpayment	1.1.4	✓							✓	✓	✓														✓	✓		✓	✓
mundipaqq *	1.1.4					-		-	✓	-	-										✓				-	-	-	-	-
paragon		✓		●		-		-	✓	-	-								✗	✗					-	-	-	-	-
payjp	1.2.1								✓	✓	✓														✓			✗	✓
paymentwall				◐				✓	✓	✓	✓					✓	✗		✗	✗	✓				✓	✓	✓	✓	✓
paymo	1.6							✓	✓	✓	✓										✓			✓	✗	✓		✓	✓
paypal	2.14.2	✓	✓	-		-		-	✓	-	-	-	-	✗		✓	✗									✓		✓	-
paysafe *						-		-		-	-										✓				-	-	-	-	-
payu	1.0.1					-		-	✓	-	-	◐	◐								✓				-	-	-	-	-
platon	1.0.0					-		-	✓	-	-	●	●			✓					✓				-	-	-	-	-
portmoney	1.3.4		✓						✓	✓	✓										✓	✗				✓		✓	✓
razer	3.27.1		✓	○					✓	✓	✓										✓	✗			✓	✓		✗	✓
rbkmoney	0.0.23			-				✗	✓	✓	✓	-	-		✗						✓				✓	✓		✗	✓
redsys	1.0.0		✗			-		-	✓	-	-					✓					✓	✓			-	-	-	-	-
sberbank *	1.0.2					-		-	✓	-	-										✓				-	-	-	-	-
shopify *	3.2.3					-		-	✓	-	-										✓				-	-	-	-	-
simplify	3.4.0	✓						✓	✓	✓	✓														✓	✓		✓	✓
square *	1.4.0					-		-	✓	-	-										✓				-	-	-	-	-
squarenonce *	1.4.0					-		-	✓	-	-										✓				-	-	-	-	-
stripe	14.5.0					✗		✓	✓	✓	✓	○	○								✓				✓	✓		✓	✓
tappay	2.3.1			◐				✓		✓	✓	◐	◐			✓									-	-	-	-	✓
tpay									✓	✓	✓					✓					✓				✓	✓	✓	✓	✓
tranzzo *	1.0.1					-		-	✓	-	-										✓				-	-	-	-	-
veritrans **	1.23.1			-		-		-		-	-	-	-												-	-	-	-	-
wepay	3.0.1			◐		-		-	✓	-	-	◐	◐			✓					✓	✓			-	-	-	-	-
wirecard	2.18.2	✓	✓	-				✓	✓		✓	-	-	✗	✓	✓			✗	✗		✗			✓		✓	✓	✓
worldpay *						-		-	✓	-	-										✓				-	-	-	-	-
xpay				◐				✓	✓	✓	✓								✗	✗	✓				✓	✓	✓	✓	✓
yandex	3.0.3	✓		-				✗	✓	✓	✓	-	-			✓	✗				✓				✓	✓		✓	✓

Table 8: MASVS checks for Data Storage and Privacy requirements.

ID	Requirement	MASVS ID	Check Type	Severity	L1	L2
DS-1	System credential storage facilities need to be used to store sensitive data.	2.1	Code Syntax	Mid	✓	✓
DS-2	No sensitive data should be stored outside of the app container or system credential storage facilities.	2.2	Code Syntax , XML	High	✓	✓
DS-3	No sensitive data is written to application logs.	2.3	Dataflow	High	✓	✓
DS-4	No sensitive data is shared with third parties unless it is a necessary part of the architecture.	2.4	Dataflow	High	✓	✓
DS-5	The keyboard cache is disabled on text inputs that process sensitive data.	2.5	XML	Mid	✓	✓
DS-6	No sensitive data is exposed via IPC mechanisms.	2.6	Dataflow	High	✓	✓
DS-7	No sensitive data, such as passwords or pins, is exposed through the user interface.	2.7	XML	Mid	✓	✓
DS-8	No sensitive data is included in backups generated by the mobile operating system.	2.8	XML	Low		✓
DS-9	The app removes sensitive data from views when moved to the background.	2.9	Code Syntax	Mid		✓
DS-10	The app enforces a minimum device-access-security policy, such as requiring the user to set a device passcode.	2.11	Code Syntax	Low		✓
DS-11	No sensitive data should be stored locally on the mobile device. Instead, data should be retrieved from a remote endpoint when needed and only be kept in memory.	2.13	Dataflow	High		✓
DS-12	If sensitive data is still required to be stored locally, it should be encrypted using a key derived from hardware backed storage which requires authentication.	2.14	Dataflow	High		✓

Table 9: MASVS checks for Cryptography requirements.

ID	Requirement	MASVS ID	Check Type	Severity	L1	L2
CRYPTO-1	The app does not rely on symmetric cryptography with hardcoded keys	3.1	CryptoGuard	High	✓	✓
CRYPTO-2	The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices.	3.3	CryptoGuard	High	✓	✓
CRYPTO-3	The app does not use cryptographic protocols or algorithms that are widely considered deprecated for security purposes.	3.4	CryptoGuard	High	✓	✓
CRYPTO-4	All random values are generated using a sufficiently secure random number generator.	3.6	CryptoGuard	Mid	✓	✓

Table 10: MASVS checks for Network Communication requirements.

ID	Requirement	MASVS ID	Check Type	Severity	L1	L2
TLS-1	Data is encrypted on the network using TLS. The secure channel is used consistently throughout the app.	5.1	CryptoGuard	High	✓	✓
TLS-2	The TLS settings are in line with current best practices, or as close as possible if the mobile operating system does not support the recommended standards.	5.2	CryptoGuard	High	✓	✓
TLS-3	The app verifies the X.509 certificate of the remote endpoint when the secure channel is established. Only certificates signed by a trusted CA are accepted.	5.3	CryptoGuard	High	✓	✓
TLS-4	The app either uses its own certificate store, or pins the endpoint certificate or public key, and subsequently does not establish connections with endpoints that offer a different certificate or key, even if signed by a trusted CA.	5.4	Code, XML	Mid		✓

Table 11: MASVS checks for Platform Interaction requirements.

ID	Requirement	MASVS ID	Check Type	Severity	L1	L2
PLAT-1	The app only requests the minimum set of permissions necessary.	6.1	XML	Mid	✓	✓
PLAT-2	The app does not export sensitive functionality via custom URL schemes unless properly protected.	6.3	XML	High	✓	✓
PLAT-3	The app does not export sensitive functionality through IPC facilities unless properly protected.	6.4	XML	High	✓	✓
PLAT-4	JavaScript is disabled in WebViews unless explicitly required.	6.5	Code Syntax	Mid	✓	✓
PLAT-5	WebViews are configured to allow only the minimum set of protocol handlers required	6.6	Code Syntax	Mid	✓	✓
PLAT-6	If native methods of the app are exposed to a WebView, verify that the WebView only renders JavaScript contained within the app package.	6.7	Code Syntax	Mid	✓	✓
PLAT-7	The app protects itself against screen overlay attacks.	6.8	Code Syntax	Low		✓
PLAT-8	A WebView's cache, storage, and loaded resources should be cleared before the WebView is destroyed.	6.10	Code Syntax	Mid		✓

Table 12: MASVS checks that were out of scope of the study.

MASVS ID	Requirement	L1	L2
2.10	The app does not hold sensitive data in memory longer than necessary, and memory is cleared explicitly after use.		✓
2.12	The app educates the user about the types of personally identifiable information processed, as well as security best practices the user should follow in using the app.	✓	✓
2.15	The app's local storage should be wiped after an excessive number of failed authentication attempts.		✓
3.2	The app uses proven implementations of cryptographic primitives.	✓	✓
3.5	The app doesn't re-use the same cryptographic key for multiple purposes.	✓	✓
5.5	The app doesn't rely on a single insecure communication channel (email or SMS) for critical operations, such as enrollments and account recovery.		✓
5.6	The app only depends on up-to-date connectivity and security libraries.		✓
6.2	All inputs from external sources and the user are validated and if necessary sanitized. This includes data received via the UI, IPC mechanisms such as intents, custom URLs, and network sources.	✓	✓
6.8	Object deserialization, if any, is implemented using safe serialization APIs.	✓	✓
6.11	Verify that the app prevents usage of custom third-party keyboards whenever sensitive data is entered (iOS only).		✓

Table 13: Nodes of the Data Ontology of sensitivity level High and Medium. Sensitivity level Low had 124 nodes which was too large to put in the table.

Sensitivity	Ontology Nodes
High	bankaccountnumber,bankroutingnumber,biometricinformation,credential,creditcardsecuritycode,driverlicensenumber, financialaccountinformation,government-issueidentificationinformation,password,paymentcardinformation, paymentcardnumber,securitycode,socialmediaaccountcredential,ssn,tin
Medium	accountbalanceinformation,accountinformation,androidid,contactinformation,credithistory,creditinformation,dateofbirth, deviceidentifier,emailaddress,expirationdate,financialtransactioninformation,identifier,imei,insurancepolicynumber,ipaddress, key,healthinformation,loginformation,macaddress,medicalinsuranceinformation,paymentcardexpirationdate,paymentinformation, phonenumber,postaladdress,sensitivepersonalinformation,serialnumber,transactioninformation,token,vehicleidentificationnumber, vehiclelicensenumber,zipcode

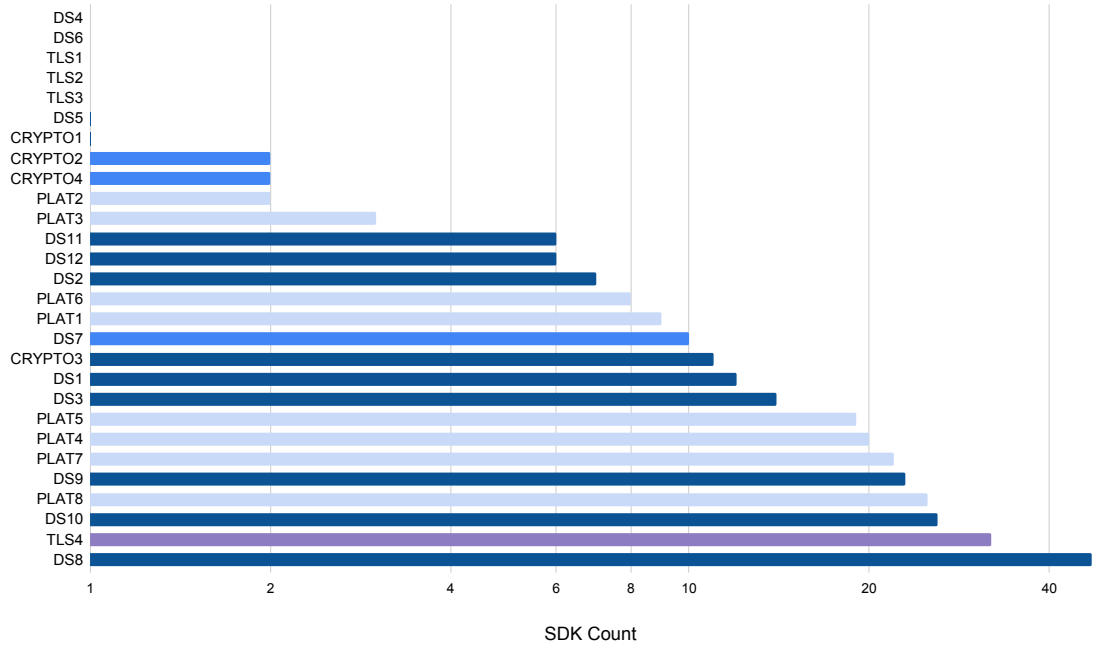

Figure 3: Count of SDK failing each check