Secret State Leakage Attacks and Their Impacts on EMV Contactless Payment Apps

Jesse Chen*, Rubin Yuchan Yang*, Ahmad Musa*,
Syed Rafiul Hussain†, Omar Chowdhury‡, Sazzadur Rahaman*
*University of Arizona, †Pennsylvania State University, ‡Stony Brook University

***Abstract*—This paper analyzes the security of EMV contactless mobile payment (ECM) apps, virtualization of physical EMV chip cards, in a less explored but relevant threat model. In this threat model, *a local adversary* such as the legitimate ECM app user (possibly, with root privileges) launches attacks to expose the *EMV protocol's internal secret states* from the app. Such secret leakage combined with the attacker's capability to modify the ECM app behavior can be exploitable for potentially self-serving purposes (*e.g.,* double-spending). To formally study such *secret state leakage attacks (*SecStLeak*) and their impacts*, we pose the *minimal satisfying cut-set identification* problem for the EMV contactless protocol design where the goal is identifying the minimal number of the protocol's secret state fields whose leakage can entail different attacks. We solve this problem by proposing a meta-level protocol analysis approach.**

Our analysis identified 4 minimal sets of secret state fields that ECM app developers must protect to prevent such attacks. We analyzed 136 Android ECM apps and identified all secret fields for 2 of the 4 minimal sets across 24 apps. In addition, one can leak a third minimal set in 3 of the 24 apps. The potential impact is significant, with these 24 apps having 82M downloads, 6 coming from developing countries, and 3 operating in a country without support for Google Wallet. To establish our findings' real-world applicability, we demonstrate a core guarantee-violating end-to-end attack on a real ECM app in an isolated test environment. This successful exploitation motivated us to study how developers in the real world protect the secret state fields belonging to the 4 minimal sets to thwart SecStLeak attacks. We observe that a majority of these apps, contrary to EMV standard's recommendation, rely on circumventable, sub-optimal software-only defenses.

1. Introduction

The landscape of the payment card industry ecosystem is rapidly shifting from chip cards to EMV contactless mobile payment (ECM) applications that support contactless transactions. ECM applications virtualize contactless EMV chip cards through card emulation technology in smartphones. The switch from physical cards to ECM apps has gained popularity, estimated to generate global transaction values over $9.4T [1], due to the improved user experience and convenience [2]. *Analyzing the security of ECM apps—specifically, in the light of moving from tamper-resistant chip cards to apps—is the main focus of this paper.*

The EMV protocol, by design, relies on cryptographic mechanisms and materials to provide security and privacy guarantees of transactions, and to safeguard honest cardholders and merchants from malicious communication peers. Note that the EMV protocol, like many protocols, explicitly assumes the secrecy of the cryptographic materials and other state information for ensuring its guarantees. We refer to the *secret* cryptographic materials (*e.g.,* keys, cardholder data) and other state information used in the ECM app side for providing security and privacy during the EMV protocol execution as its *secret state*. Leakage of the EMV protocol secret state can thus allow an attacker to violate the assured security and privacy guarantees of the EMV protocol.

A majority of the existing work focusing on the security and privacy of EMV protocol design and implementation primarily concerns itself with an unprivileged network adversary and the payment terminal [3], [4], [5], [6], [7], [8], [9], [10], [11]. Since ECM apps can potentially execute in adversary-controlled environments (*i.e.,* the smartphone), the study of their security and privacy additionally warrants considering a richer threat model. In this threat model, a local privileged adversary (*e.g.,* the legitimate user of the ECM app)—with the capabilities of: reverse engineering the ECM apps, observing and intercepting their network traffic, and modifying and inspecting their executions, memory, and stored data [12] —aim to extract secret EMV protocol state information from the ECM apps, and exploit them for their self-gaining purposes (*e.g.,* double-spending). We call such attacks *secret state leakage* (SecStLeak) *attacks*. In this paper, we specifically study the impact and feasibility of such SecStLeak attacks on ECM apps.

The secret EMV protocol state of an ECM app can be *conceptually* viewed as a record or structure. The fields of this secret state record correspond to different secret cryptographic materials and other state information critical to ensuring the overall protocol security and privacy. We want to emphasize that knowing some of the secret fields' values may allow one to derive others based on the protocol design. We consider both *derivable* (*e.g.,* ephemeral session keys) and *non-derivable* fields (*e.g.,* master key) of the secret state. The reasoning behind focusing on both types of fields is that their values may not receive the same level of strong protection due to a perceived difference in their importance. It is thus conceivable that, following the *principle of least effort* [13], the attacker goes after less protected derivable field values for launching EMV-guarantee-violating attacks.

It is *theoretically* possible to *correctly* use hardware security mechanisms such as Trusted Execution Environments (TEE) [14] to protect ECM apps against SecStLeak attacks. Such hardware features, however, are not always present in smartphones. This creates the following dilemma for the developers of ECM apps: *(i)* only support TEE-enabled smartphones at the cost of losing a large consumer base with non-TEE-enabled smartphones; *(ii)* support non-TEE-enabled smartphones at the expense of opening their apps to secret state leakage attacks. Our **central hypothesis** is that developers (along with relevant decision-makers) are most likely to choose option *(ii)*: *preferring economic incentives over security or lacking sufficient awareness to mitigate* SecStLeak *attacks*—especially the ones operating in markets where non-TEE-enabled devices constitute the majority of active smartphones.

In the absence of TEE-like mechanisms, app developers have to resort to the following software-only protection mechanisms that increase the bar for the attackers to launch SecStLeak attacks but cannot provide theoretically strong security guarantees: root checking; network integrity; anti-repackaging; anti-hooking; anti-debugging; and code obfuscation. Unfortunately, bypassing these mechanisms to launch SecStLeak attacks boil down to a cat-and-mouse game [15] and the attacker's success depends on their motivation, and incentive. Due to the inherent monetary incentives associated with ECM apps, attackers have a strong motivation to launch SecStLeak attacks.

In this paper, to study the SecStLeak attacks and their impacts on ECM apps' security and privacy in the absence of *proper* TEE-like mechanisms through a formal lens, we pose the *minimal satisfying cut-set identification* problem. The problem requires identifying the minimal number of the EMV protocol's secret state fields, the leakage of whose values can entail different attacks. We solve this problem by proposing a meta-level protocol analysis approach that uses a symbolic protocol verifier as the underlying reasoning engine. One of the desired features of this approach is that it allows one to systematically extend an existing symbolic EMV protocol design model [8], [9] — used for analyzing it against network attackers — by introducing controlled protocol secret state field leakages, and then checking for relevant desired self-serving property (*e.g.,* no double-spending) violations with a protocol verifier. Such an incremental approach essentially allows us to reuse symbolic EMV protocol design for a Dolev-Yao-style network attacker, a local attacker, and also their combinations; amortizing the upfront cost of model construction. Our analysis identified **4 different minimal sets** of secret state fields. If the values of *all fields in a minimal set* are leaked by attackers, then it is possible for them to launch self-serving attacks. To protect against such attacks, it is, thus, critical for developers to secure at least one field from each of the 4 minimal sets.

Conceptually, the protection from SecStLeak attacks on ECM apps can be viewed as a *Digital Rights Management* (DRM) problem [16]. In contrast to the classic DRM bypass attacks [17] where attackers have to invest substantial manual reverse engineering efforts to understand the behavior of the app before launching the attacks, the behavior of ECM apps is known and mandated by the EMV standard. In a similar vein, the Bellare–Rogaway adversary model [18] serves as a useful conceptual reference. It enables the study of secret-key leakage in cryptographic protocols under the computational model, whereas our analysis is symbolic. Conceptually, our minimal cut-set analysis captures a broader view of secret-state leakage by accounting for both derivable and non-derivable fields, whereas the Bellare–Rogaway model considers only non-derivable fields (i.e., long-term keys). As derivable and non-derivable fields may have different protection levels, it is worth investigating both.

To validate our central hypothesis and evaluate how developers protect fields in the 4 minimal sets we identified, we collected 136 Android ECM apps. We have also developed an automated approach to check whether ECM apps employ defenses suggested by the EMV standard to protect against SecStLeak attacks. Our evaluation revealed that only 12 of them use TEEs, and only 1 (Google Wallet) uses TEE securely. 82 can run on rooted devices, and 76 have interceptable network connections through which secret data elements are transmitted to the apps. Our analysis showed that 2 out of 4 minimum sets of secret state fields can be leaked in 24 apps and a third minimal set can be leaked in 3/24 apps. To show that these findings are applicable to real-world attacks, we demonstrate an end-to-end attack on a real ECM app in an isolated and controlled test environment and discuss how the exploit we developed has immediate consequences on at least 24 apps that leak 2 minimum sets.

Disclosure. We have responsibly disclosed our findings to the developers of 16 apps for which we located min-cut sets and observed minimal protection enforcements. We received responses from 4 and met with 1.

Technical Contributions. In summary, the paper has the following contributions. Artifacts can be found in [19].

- **Security Evaluation (§4):** We conduct a comprehensive measurement study on 136 ECM apps' security protections against SecStLeak. We found that while 12 apps may potentially use TEE to store secret data, only one app (GoogleWallet) imports keys securely. Our study also revealed that 82 payment apps ran on rooted devices, which validates our central hypothesisd that existing apps do not have adequate protection against SecStLeak.
- **Formal Foundation (§5):** We create a novel meta-level protocol analysis framework to study EMV contactless payment protocol under SecStLeak. We discover 4 min cut-sets that, when leaked, break the desired security properties, allowing fraudulent transactions.
- **Real-World Implications (§6):** We analyzed 136 ECM apps to locate min cut-sets. We identified two of them in 24 apps, and a third minimal set in 3 of those 24. We also demonstrate an end-to-end exploitation on a real ECM app in an isolated, controlled test environment, and show that the exploit has immediate security implications for at least the 24 apps leaking the 2 minimal sets.

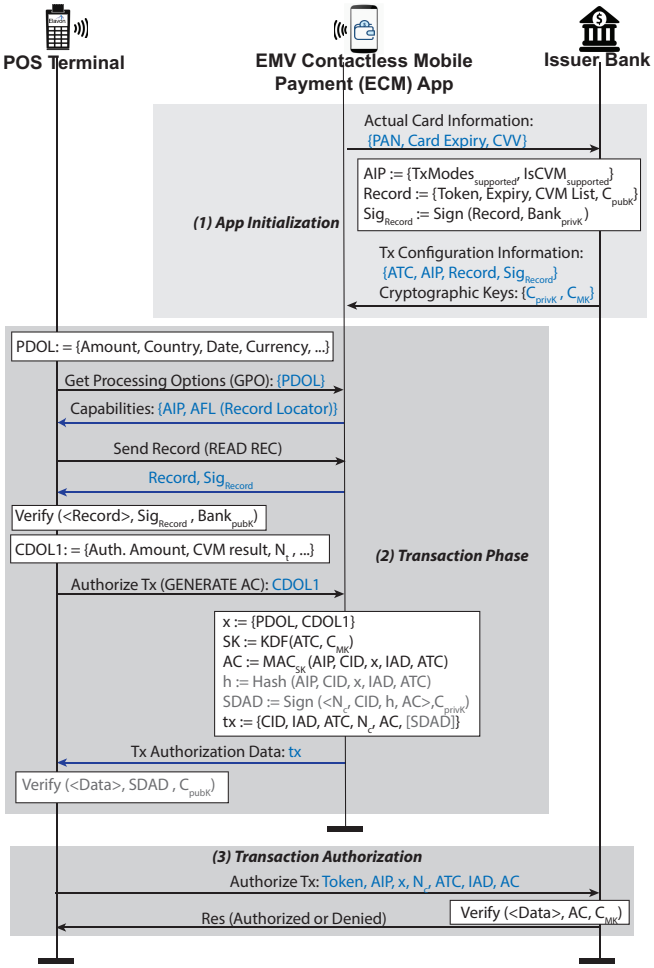

Figure 1: A simplified view of different stages of initializing and conducting EMV contactless protocol with EMV contactless mobile (ECM) payment apps. Note that generating and validating SDAD is optional in online mode [20]

2. Background: Mobile Payment Overview

For understanding the impact of secret state leakage (SecStLeak) attacks on real-world ECM apps, it is crucial to understand how ECM apps are provisioned with secret cardholder data elements and cryptographic key materials used in EMV contactless transactions. EMV contactless transactions support two operating modes: *online*, where the terminal seeks authorization from the issuer bank before making a decision, and *offline*, where the terminal accepts based on card-generated authentication proofs without checking it with the issuer bank. We now provide an overview of EMV contactless mobile payment (ECM) app initialization, transaction, and transaction authorization flows (Figure 1) for conducting online and offline mode transactions, and discuss the key differences and conceptual similarities of the ECM app with traditional chip cards.

(1) App Initialization: After downloading an ECM app, a user first needs to *initialize* it. Initialization here refers to the *over-the-air* provisioning and personalization of the *virtual card* with secret card data, cryptographic key materials, and configuration parameters required for EMV contactless transactions. Conceptually, this process is *similar* to the personalization phase of a physical EMV chip card, during which these data elements are embedded into a tamper-resistant chip. The key *difference* is that, in ECM apps, actual cardholder data are replaced by tokenized credentials [21], device-specific surrogates that cannot be used outside their intended context (*e.g.,* for online payments). In this phase, two primary cryptographic keys are provisioned to the ECM app by the issuing bank: the Card Private Key (C_{privK}) and the Card Master Key (C_{MK}). The C_{privK} is the private component of the card's asymmetric key pair, while the C_{MK} is a symmetric key. Both these keys are used by the card for performing transactions. After initialization, all personalized data elements and key materials are stored within the app.

(2) Transaction Phase: Tapping the card on the terminal initiates the transaction phase (*i.e.,* EMV contactless protocol). The card is emulated through an ECM app on a mobile device. Notably, an ECM-based transaction is similar to a chip-card-based contactless transaction [22], with the following key difference. Unlike in chip-card-based transactions, the Cardholder Verification Method (CVM) is performed in the app rather than in the terminal. This special CVM is known as OD-CVM. While Figure 1 illustrates the key steps of the ECM transaction, from a security perspective, the most critical operations occur after the *GENERATE AC* command, which is worth discussing here. As the result of *GENERATE AC*, card produces the following two *authentication proofs*:

- Application Cryptogram (AC): A message authentication code (MAC) of transaction data, generated with Session Key (SK), such that:

$$x := \{\text{PDOL}, \text{CDOL1}\}$$
$$SK := KDF(\text{ATC}, C_{MK})$$
$$AC := MAC_{SK}(\text{AIP}, \text{CID}, x, \text{IAD}, \text{ATC})$$

  ATC (Application Transaction Counter) is a monotonic counter maintained by the ECM app to ensure transaction freshness and prevent replay. AIP (Application Interchange Profile) and CID (Cryptogram Information Data) describe the card's operational capabilities and the type of cryptogram being generated, respectively. IAD (Issuer Application Data) contains proprietary, issuer-specific data.
- Signed Dynamic Application Data (SDAD): A digital signature generated for proving the authenticity of *the card* and *the transaction data*. It is produced using the card's private key (C_{privK}) corresponding to the issuer-certified public key (C_{pubK}), such that:

$$h := Hash(\text{AIP}, \text{CID}, x, \text{IAD}, \text{ATC})$$
$$\text{SDAD} := Sign_{C_{privK}}(N_c, \text{CID}, h, \text{AC})$$

  Here, N_c is a card-generated random nonce ensuring freshness, PDOL and CDOL1 are transaction-related data provided by the terminal, which also includes a terminal-generated random nonce, N_t (Figure 1).

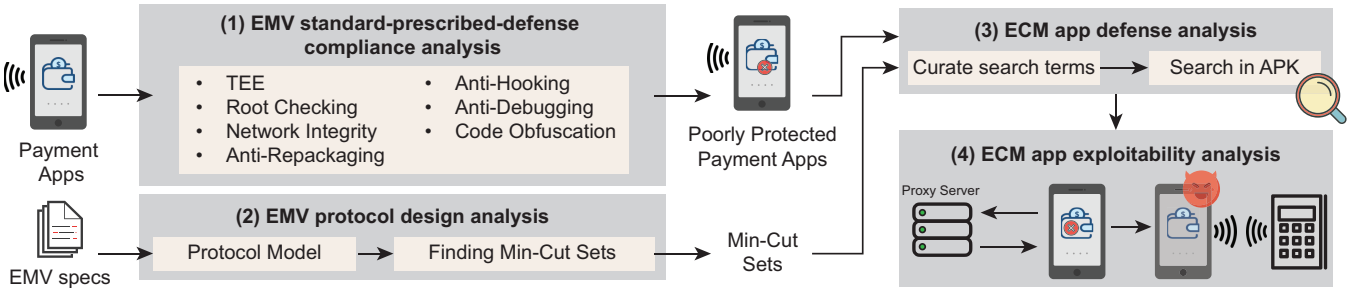

Figure 2: An overview of the approach for checking the impact of SecStLeak attacks on ECM apps.

(3) Transaction Authorization. In this phase, the issuer verifies the transaction by recomputing AC with the SK (reconstructed with C_{MK} and ATC). After that, the issuer sends the results (accepted or denied) to the terminal. *Note that, in offline mode, the terminal approves the transaction before receiving confirmation from the issuer, whereas in online mode, it approves only after the issuer has authorized it. Hence, in online mode, generating and verifying SDAD (performed by the terminal) is optional, as transaction authenticity is verified by the issuer through the AC.*

3. Approach Overview

We now discuss our overall approach for analyzing the impact of the secret state leakage (SecStLeak) attacks on real-world ECM apps. We conclude this section with a description of the applicable threat models for launching SecStLeak attacks along with precise definitions of what self-serving purposes mean in those threat models.

3.1. High-Level Approach

We have the following three overarching objectives

$\mathcal{O}_1$**:** Validate the central hypothesis that ECM app developers either (i) prioritize economic incentives (e.g., a larger customer base) over mitigating their apps' susceptibility to SecStLeak, or (ii) remain largely unaware of secret-state leakage threats, resulting in inadequate defenses.

$\mathcal{O}_2$**:** If the above hypothesis holds, identify the most critical secret-state data elements required to preserve protocol guarantees.

$\mathcal{O}_3$**:** Evaluate the effectiveness of existing defensive mechanisms in protecting these sensitive data elements.

Figure 2 shows our high-level approach for addressing the above objectives. It can be decomposed into the following three stages: ❶ Central hypothesis validation; ❷ EMV protocol design analysis; ❸ ECM app exploitability analysis.

❶ Central hypothesis validation ($\mathcal{O}_1$). The goal of this stage is to answer the following two research questions: (a) *Do most real-world ECM apps rely on TEE-like hardware security mechanism to protect from* SecStLeak *attacks*? (b) *In the absence of TEE-like mechanisms, do most real-world ECM apps employ any software-based defense mechanisms*? To answer these research questions, we designed and developed a tool called EMVResilienceChecker that automatically checks whether a given ECM app has adopted all the EMV-prescribed defenses. By design, our analysis is a conservative over-approximation of true EMV-standard compliance, as EMVResilienceChecker marks an app as non-compliant only when clear evidence indicates the absence of a prescribed defense—ensuring no false claims of non-compliance. The result of this analysis essentially validates our central hypothesis, indicating that real-world ECM apps do not sufficiently implement the protection mechanisms and are susceptible to SecStLeak attacks.

❷ EMV protocol design analysis ($\mathcal{O}_2$). Assuming our central hypothesis holds for the majority of the ECM apps, this step answers the following research question: *What are the most critical data elements that need to be protected to preserve protocol security under* SecStLeak*?* Towards that, we first analyze the EMV protocol design as described in the standard to identify ECM app's EMV protocol secret state fields whose leakage can enable an attacker to violate some desired EMV protocol security properties. We developed a meta-level protocol analysis, which uses a symbolic protocol verifier as the underlying reasoning engine. Our analysis can be viewed as a generalization of the symbolic cryptographic protocol verification against the Bellare-Rogaway threat model [18]. The Bellare-Rogaway threat model considers an adversary having all capabilities in a Dolev-Yao threat model [23] and, additionally, having capabilities of obtaining some long-term private keys by compromising the victim or through some other abstract leakage. Our analysis goes beyond long-term secrets and also considers session-specific, ephemeral secrets. The result of this analysis is a sequence of minimum subset of ECM app's secret protocol state whose leakage guarantees violation of some desired EMV security property.

❸ ECM app exploitability analysis ($\mathcal{O}_3$). The analysis performed in this stage is more focused compared to the ones in step ❶. The analysis in step ❶ essentially checks to see whether a real-world Android ECM app deploys EMV-standard defenses (*i.e.,* a Yes/No designation). In contrast, the analysis in this stage checks to see whether these apps explicitly protect the secret state fields identified as critical in analysis stage ❷. Additionally, to show that SecStLeak attacks is not purely theoretical and can impact real-world ECM apps, we demonstrate an end-to-end SecStLeak attack on a real-world ECM app in a controlled and isolated

environment, in which we show that the critical secret state fields are not only leakable but they can be exploited for self-serving purposes.

3.2. Applicable Threat Models

We now present two threat models that are relevant and applicable when analyzing an ECM app's susceptibility to SecStLeak attacks. In addition to explaining what self-serving purposes mean in each of these threat models, we also identify the desired EMV protocol property whose violation will entail the attacker achieving its self-serving purposes in the given threat model.

Threat Model I: Free-rider adversary (ADV_F): In this model, the device owner and ECM app account holder is the adversary. Precisely, in ADV_F, we consider an adversary with the following capabilities: (a) ADV_F is a legitimate account holder on the ECM app, and (b) ADV_F has physical access to the rooted device where the ECM app is running.

Self-serving purpose: In ADV_F, the self-serving purpose that the adversary aims to achieve is **not** needing to pay for a service. One way to achieve this would be to fool a legitimate terminal into accepting a transaction that will be rejected further down the payment transaction verification step, possibly by the bank.

Desired property violation entailing self-serving exploitation: Achieving the self-serving purpose in this threat model requires violating the following *mutual transaction authentication property* desirable from the EMV protocol to maintain merchant protection (see Definition 3.1).

Definition 3.1 (Simultaneous Transaction Authentication Property – $\Psi_{\texttt{simTxAuth}}$)**.** *An EMV transaction t_x is considered to be valid by a terminal only if it is deemed to be valid by the bank that holds the transaction-originating account.* We can formally state this property as the following first-order logic formula where we do not impose any ordering restrictions between the time points i and j as depending on the underlying operating mode (*offline* or *online*) the two events (*terminal accepting* or *the bank rejecting the transaction*) can legitimately happen in any order:

$$\forall t_x : \texttt{Transactions},\ i : \texttt{Timepoints}.$$
$$\mathsf{TerminalAcceptsTxAtTime}(t_x, i) \Rightarrow$$
$$\neg(\exists j : \texttt{Timepoints}.\ \mathsf{BankRejectsTxAtTime}(t_x, j))$$

Threat Model II: Piggyback-rider adversary (ADV_PB): In this threat model, an app installed on the victim's smartphone – different from the ECM app under analysis – is the adversary. In ADV_PB, an adversary has the following capabilities: (a) ADV_PB has an app installed in the victim's smartphone; (b) this attacker-controlled app can exploit a privilege escalation vulnerability in the victim's phone to obtain root privileges. Although at a first glance this seems like a strong adversary model, prior exploitations of such vulnerabilities indicate the feasibility of realizing this adversary. For instance, Pegasus uses zero-interaction, remote exploitation to spy on specific targets [24]. It is therefore essential for developers to recognize such vulnerabilities and implement preventive measures.

Self-serving purpose: In ADV_PB, the self-serving purpose that the adversary aims to achieve is paying for a service with someone else's account. One way to achieve this would be for the adversary-controlled app to gain access to some secret state field values that are sufficient to carry out a transaction.

Desired property violation entailing self-serving exploitation: Achieving the self-serving purpose in this threat model requires violating the following *cloning resistance property* desirable from the ECM app to maintain cardholder protection (see Definition 3.2).

Definition 3.2 (Cloning Resistance Property – $\Psi_{\texttt{cloneResist}}$)**.** *An EMV transaction t_x is successful only if when it originates from an ECM app in which a legitimate account is added (i.e., it cannot be carried out by an entity by stealing secret state information of the underlying EMV protocol state in an ECM app).* One can formalize the property as the following first-order logic formula, which essentially captures that in a model where the legitimate users do not perform any transactions and an adversarial wallet can perform transactions with stolen secret state fields, no transactions should be accepted by the terminal.

$$\neg(\exists t_x : \texttt{Transactions},\ i : \texttt{Timepoints}.$$
$$\mathsf{TerminalAcceptsTxAtTime}(t_x, i))$$

4. Central Hypothesis Validation

In this section, we study to what extent real-world ECM apps are susceptible to secret-state leakage attacks. Specifically, we study whether real-world ECM apps comply with EMV-prescribed hardware- or software-based defenses [25] to mitigate local adversary attacks, such as SecStLeak attacks described in §3.2. To conduct this study, we design a tool called EMVResilienceChecker [19] to automatically determine a given app's compliance. It is designed to automatically check the presence of 7 security properties specified in the EMV guidelines [25]. EMV specifically recommends using hardware-based Trusted Execution Environments (TEEs) whenever feasible for their tamper-resistance property. Thus, EMVResilienceChecker checks how existing ECM apps employ TEE-like solutions (i_1). The remaining 6 recommendations are software-based defenses intended to mitigate application tampering and reverse engineering: Root Checking (i_2), Network Integrity (i_3), Anti-repackaging (i_4), Anti-hooking (i_5), Anti-debugging (i_6), and Code Obfuscation (i_7). Next, we present our design principles for EMVResilienceChecker (implementation details in Appendix §A).

Next, we briefly discuss the design principles of our tool, followed by evaluation results.

4.1. EMVResilienceChecker Design Principles

In general, designing an automated tool to verify any non-trivial property of a program is undecidable [26]. Thus,

most approaches to check software compliance offer partial guarantees or best-effort approximations, where the primary focus is on balancing the trade-offs among false positives (FP), false negatives (FN), and scalability of the analysis [27], [28]. While developing EMVResilienceChecker, we also followed the same principle. Thus, our primary and secondary requirements are:

1) Fairness-Req: avoid unfairly labeling a compliant app as non-compliant
2) LowOH-Req: minimize the computational overhead without compromising the analysis quality

While we utilized both static and dynamic analysis methods to implement different checks, the choice was dictated by the implementation feasibility, which we discuss next.

Static analysis-based checks. We implement 3 checks using static analysis: TEE (i_1), anti-debugging (i_6), and code obfuscation (i_7). These checks are all designed to detect *compliance*. This decision was guided by the feasibility of implementing such checks *reliably* with minimal risk of breaking Fairness-Req. For example, to determine if an ECM app uses a trusted execution environment (TEE), studying the usage of TEE-relevant APIs [29] is sufficient, as this is the only known way to use TEE in Android apps (i_1). This approach yields low false negative rate (FNR), *i.e.,* missing detection [27], [30] with reasonable computational overhead, satisfying both Fairness-Req and LowOH-Req. Note that, this approach may result in false positives (FP), *i.e.,* an app might not store protocol-specific data in TEE, however, being conservative is acceptable, since it prioritizes avoiding unfair on-compliance attribution over *precision*.

Dynamic analysis-based checks. For the other 4 recommendations, we employed dynamic analysis-based approaches: root checking (i_2), network integrity checking (i_3), anti-repackaging (i_4), and anti-hooking (i_5). To implement a check with a dynamic analysis-based approach, the compliance or non-compliance symptoms must be externally visible. The choice of these checks was guided by this requirement. Specifically, 3 checks were designed to verify *non-compliance* ($i_3, i_4,$ and i_5) and one was designed to verify *compliance* (i_2).

Example: Root Checking Detection. Our tool detects the absence of root checking by running the app on a rooted device to verify if the app detects the device as rooted. If the app cannot detect that the device is rooted, we conclude that the app is non-compliant with the EMV guideline. Fortunately, root checking is often performed at launch [31], before the login screen, making runtime implementation feasible while reasonably satisfying Fairness-Req and LowOH-Req. However, determining whether an app successfully detected that it is running on a rooted device is challenging. This is because the app can respond in many ways. First, we need to check whether the app has crashed, which is done by detecting whether the app remains in the foreground 5s after launching the app. Should the app run without crashing, it can respond to root detection by displaying a message in any form supported by Android, dialogue box, Toast message, new Activity window, etc. For all of these variations, EMVResilienceChecker needs to decide whether the response is relevant to the presence of root-checking. Since modern large language models (LLMs) have been proven to be reliable for such contextual decision-making and text classification [32], we utilized the largest LLM that can run our machine (meta-llama/Llama-3.2-3B-Instruct [33]) for this purpose. Specifically, we use an LLM-based classifier to determine if any of the texts on screen are displaying a message indicating the detection of a rooted or insecure device. We opted for local LLMs to ensure reproducibility. Note that, overall, this approach provides a conservative approximation of non-compliant apps, as we only attempt the common Zygisk-based bypassing technique [34]. We argue that this design choice is reasonable because, even though sophisticated attackers may eventually succeed, defending against common cases demonstrates a baseline security.

4.2. ECM App Corpus Collection

For this study, we curated a real-world ECM app corpus by filtering the Androzoo dataset [35], with the following keywords: “bank”, “wallet”, “pay”, and “nfc”. Then, we also searched on Google Play with two additional keywords: “payment” and “tap&pay”. After the initial search, we remove apps that do not declare its host-based card emulation service class (done by extending `HostApduService`) [36]. We also discard apps that are not compatible with our Google Pixel 4 running Android 10. As a result, our corpus contains 136 unique ECM apps in total, representing a rich collection of bank and financial institutions worldwide, where their customers are likely dependent on these apps.

4.3. Protection Analysis Results

Here, we present the results for each compliance check with EMVResilienceChecker. We also report our manual validation results to assess its performance in avoiding the unfair classification of compliant apps as *non-compliant* (typically FPs). We argue that this focus is reasonable because (1) our tool is specifically designed with this principle (Fairness-Req), and (2) validating cases where a non-compliant app is classified as compliant (typically FNs) requires substantial effort to exhaustively check all possibilities. This validation choice is widely accepted in the existing literature [28], [37], [38]. Overall results are summarized in Table 1. Our findings, especially TEE (i_i) and root checking (i_2) results, validates our central hypothesis that companies either prioritize economic incentives over security or unaware of the secret state leakage adversary, leading to inadequate defense. Next, we present detailed findings for each rules.

TEE Usage (i_1): We found only 12/136 APKs used some form of TEE. Our manual validation confirms that all these instances were legitimate use of the APIs. Specifically, we found 11 using *isInsideSecureHardware()*, only 1 using `WrappedKeyEntry`, and none using *getSecurityLevel()*. For *isInsideSecureHardware()*, we found 8 from `com.visa`, 1 from `in.juspay`, and 2 from the main app’s package. The only app to use `WrappedKeyEntry`

ID	Rule	Analysis Type	Count	FPs
i_1	TEE Usage	Static	12	0
i_2	Root Checking	Dynamic (+ LLM)	54	4
i_3	Network Integrity	Dynamic	76	-
i_4	Anti-Repacking	Dynamic	64	-
i_5	Anti-Hooking	Dynamic	94	-
i_6	Anti-Debugging	Static	134	-
i_7	Code Obfuscation	Static		
	- Identifier Renaming	(Heuristic)	42	-
	- String Encryption	Existing tool (JEB)	121	-
	- Reflection	Existing tool (JEB)	112	-
	- Control-Flow	Existing tool (JEB)	108	-
	- Virtualization	Existing tool (JEB)	22	-
	Total # of apps	-	136	-

TABLE 1: Summary of our defense-check results across real-world ECM apps against secret-state leakage attacks—indicating inadequate protection. In the *Count* column, green colored numbers indicate number of apps that are protected (compliant), and red indicates the number of apps not protected (non-compliant). A dash indicates that FPs are not applicable to this check, as it is deterministic or not practical for manual verification.

is Google Wallet, meaning that no other app is importing keys (or secret data) from the server to the device securely.

Findings 1 (TEE usage): 12/136 apps employ some form of TEE usage to store keys, while only Google Wallet imports keys securely. In the other 124 apps, data elements can be intercepted before importing to TEE.

Root Detection (i_2): EMVResilienceChecker reported that 63/136 apps could detect that they were running on a rooted device without any root evasion, *i.e.,* 73 failed to detect our device as rooted. The use of large language model (LLM) to determine if the app was able to detect the rooted device may introduce imprecision here. Thus, to understand EMVResilienceChecker's performance, we manually validated the results from the LLM. We found that the LLM failed to detect 10 cases where the app successfully detected a rooted device (FNs). Of these 10 misclassifications, 7 had non-English language messages. Additionally, in 5 cases, LLM wrongfully reported that these apps detected a rooted device (FPs). For instance, it is considered *"Preventing fraud and enhancing the security of your account"* as an indication of detecting a rooted device. After manual validation, we found that 68/136 apps detected our rooted device, *i.e.,* 68/136 ECM apps failed to detect our device as rooted without any evasion. We then added a Zygisk-based evasion [34] and, after manual validation, we found that an additional 18/68 ran on our rooted device. In total, 82 apps can run on our rooted device , *i.e.,* we found that 54 apps detected our rooted device.

Finding 2 (Root Checking Usage): 68/136 apps run on our rooted device without evasion, and 18/68 with evasion. This totals to 82/136 that can run on rooted devices—indicating inadequate defense.

Network Integrity (i_3): Network integrity refers to the state in which communication between servers and ECM applications is both end-to-end encrypted and protected against unauthorized modification by man-in-the-middle (MITM) adversaries. EMVResilienceChecker uses two approaches to break network integrity: (1) tampering with the system certificate store by adding an MITM certificate; (2) modifying the application to bypass security checks when certificate pinning is in place (details in §A). In total, our tool was able to intercept the network traffic in 76/136 apps. Of these 76 app, it had to patch only 2 apps to disable certificate pinning. *For these apps, an attacker can likely steal secret data elements from the apps by simply intercepting the network traffic between the server and the apps.* EMVResilienceChecker failed to disable certificate pinning in the remaining 24 apps. The high failure rate is expected because repackaging is unreliable [39].

Findings 3 (Network Integrity): 74/136 apps blindly trust the system store, and it was possible to bypass certificate pinning for 2 other apps. Thus, for 76/136 apps, communications can be decrypted by a MITM.

Anti-Repackaging (i_4): App repackaging is a process where the APK is modified to alter its behavior and signed again before execution. EMVResilienceChecker found that 64/136 apps ran successfully after repackaging. Of the remaining apps, 18 failed repacking, 17 failed installing, and 37 crashed after launch. Crashing is a recommended action for apps that detect repackaging, though it could also be from repackaging problems.

Anti-Hooking (i_5): Hooking is the act of intercepting, monitoring, or modifying the behavior of code in an app during runtime. EMVResilienceChecker found that 94 apps did not prevent us from hooking. This makes them susceptible to functionality modifications (*e.g.,* bypassing root detection), and sensitive data leakage (*e.g.,* min-cut sets), if other protection mechanisms are not implemented sufficiently.

Anti-Debugging (i_6): Debugging capability lets an adversary run the app in debug mode to inspect different program points while running the app. We found that only 2 of the apps set the debuggable flag to true. However, it is worth noting that for at least 64 apps, debugging can be easily enabled by repackaging.

Findings 4 (Anti-Repackaging, Anti-Hooking and Anti-Debugging): 64/136 apps operate normally after being repackaged. 94/136 apps can run with hooking, while 2 permit debug mode.

Code Obfuscation (i_7): In total EMVResilienceChecker can detect 4 types of code obfuscation. We conducted a comprehensive search for Android obfuscation techniques, obfuscation tools, and obfuscation detectors to conduct this study. After an extensive evaluation of the obfuscation detectors, we found JEB to show reasonable performance and

chose it [1] to detect string encryption, reflection, control flow, and virtualization obfuscation in the 136 apps.

Since we did not find any reliable tool or methodology in existing literature to detect identifier renaming, we implemented a heuristic-based approach in EMVResilienceChecker (details in §A). This is because identifier renaming may play a critical role in obfuscating the names of the variables and methods for secret data elements. Our manual validation on 100 randomly selected identifiers shows EMVResilienceChecker missed 1 instance from 14 cases where identifiers were renamed (FN). It labeled the identifiers to be identifier renaming-obfuscated in 8 cases, where they were not (FP). This indicates that the performance of our identifier renaming detection method was reasonable while satisfying Fairness-Req. Our findings are detailed in Table 2, where we report usage rates in (1) the entire app and (2) the base package only (*i.e.,* developer code). One notable observation is that virtualization has a low usage rate of 28% likely because it slows down the app during operation, which could cause contactless transactions to exceed the 500ms time bound [41], [42].

What types of obfuscation are used in the 136 apps?

Obfuscation Type	# of apps used in			
	Entire app		Base package	
Identifier Renaming	42	(31%)	11	(24%)
String Encryption	121	(89%)	35	(78%)
Reflection	112	(82%)	6	(13%)
Control Flow	108	(79%)	25	(56%)
Virtualization	22	(16%)	1	(2%)
Total	136	(100%)	45	(100%)

TABLE 2: Parentheses show obfuscation usage percentages relative to the 45 *identifiable* base packages after decompilation. To ensure that we do not violate Fairness-Req, we consider the app to be compliant for a given obfuscation if it's used once or more or at least 50% renamed identifiers for identifier renaming.

Findings 5 (Obfuscation Usage): Identifier renaming substantially raises the bar for reverse engineering with minimal effort, but is used only in 42/136 (31%) apps.

5. Formal Analysis for Examining the Impact of Secret State Leakage in ECM Apps

As shown in §4.3, a vast majority of the ECM apps are either unprotected or partially protected. This leads to the question of *"what are the most critical data elements that needs protection to preserve EMV protocol security under secret state leakage attacks."* Recall that the *secret state of the underlying EMV protocol within an ECM app* can be conceptually viewed as a record whose fields represent the various secret and cryptographic materials required to ensure the overall security guarantees of the EMV protocol design. Theoretically, it may appear obvious that any protocol relying on secret or cryptographic materials (*e.g.,* keys) is inherently vulnerable to assurance violations if those secrets are leaked. This legitimate observation seemingly calls into question the need for our analysis to find *critical* elements. Our analysis, however, is motivated by the insight that even if developers of ECM apps are often aware of this theoretical observation, they may primarily focus on protecting long-term cryptographic keys, while paying little to no attention to securing transient or seemingly irrelevant *derived* secret-state fields—thereby leaving their ECM apps exposed to various secret state leakage attacks (*i.e.,* SecStLeak attacks). Specifically, we aim to identify *minimal* subsets of the secret state fields whose leakage (*i.e.,* leakage of all elements of a minimal subset) enables an attacker to violate the simultaneous authentication or cloning resistance properties (see §3.2). These minimal subsets can also guide developers to apply security-enhancing mechanisms to at least one element in each minimal subset, thereby protecting their apps against such SecStLeak attacks.

1. JEB is an industrial-grade tool for reverse engineering apps [40].

5.1. Problem Statement

We now define the underlying analysis problem for identifying secret state fields whose leakage enables an attacker to launch the underlying-guarantee-violating attacks (*i.e.,* simultaneous transaction authentication and cloning resistance). We then show that one can reduce this problem to the minimal satisfying cut-set enumeration problem for a monotone predicate.

Definition 5.1 (Guarantee-Violating Minimal Secret State Field Set Identification Problem)**.** Given the secret state fields $\mathbb{F} = \{f_1, f_2, \ldots, f_m\}$ corresponding to the underlying EMV protocol of an ECM app, identify the *minimal leakage cut-set* $\mathbb{L}$ (*i.e.,* a set of sets) such that the following holds:

1) for each $\ell \in \mathbb{L}$, $\ell \in 2^{\mathbb{F}}$ (*i.e.,* ℓ is an element of the powerset of $\mathbb{F}$);
2) each element $\ell \in \mathbb{L}$ is also a ***cut-set*** such that if the values of all fields in ℓ of a EMV protocol session are leaked to the attacker, then the attacker can launch attacks to violate $\Psi_{\texttt{simTxAuth}}$ or $\Psi_{\texttt{cloneResist}}$ properties;
3) each element $\ell \in \mathbb{L}$ is a ***minimal*** *cut-set*, that is, if we remove one or more elements of ℓ, then the resulting set is not a cut-set according to condition 2 above.

In the above problem definition, it is clear that checking whether a set of fields $s \in 2^{\mathbb{F}}$ is a cut-set according to condition 2 can be abstractly viewed as checking whether s satisfies a monotone predicate $\mathcal{P} : 2^{\mathbb{F}} \rightarrow \{\texttt{True}, \texttt{False}\}$. In this abstraction, for a given $f \in 2^{\mathbb{F}}$, $\mathcal{P}(s)$ is true if and only if the values of all fields in s of an EMV protocol are leaked to the attacker, then the attacker can launch an attack to violate the $\Psi_{\texttt{simTxAuth}}$ or $\Psi_{\texttt{cloneResist}}$ properties. In our context, the resulting predicate $\mathcal{P}(\cdot)$ is monotone because if $\mathcal{P}(x)$ holds for any $x \in 2^{\mathbb{F}}$, then $\mathcal{P}(y)$ holds for all supersets y of x (*i.e.,* $\forall x, y \in 2^{\mathbb{F}}.x \subseteq y \wedge \mathcal{P}(x) \rightarrow \mathcal{P}(y)$). This can be understood intuitively by observing that if the attacker can violate the EMV protocol's guarantees by knowing the values of secret fields corresponding to the set x, then it can definitely violate the properties when given

access to values of more fields including those in x. This abstraction specifically enables us to reduce our problem to the following more general problem. We can then devise an algorithm for solving the general problem and then use this approach for solving the original problem.

Definition 5.2 (Minimal Satisfying Cut-Set Enumeration Problem)**.** Given a set of labels $L = \{l_1, l_2, \ldots, l_n\}$ and a monotone predicate $\mathcal{P} : 2^L \rightarrow \{\texttt{True}, \texttt{False}\}$, identify all minimal cut-sets $X \subseteq L$ such that $\mathcal{P}(X)$ holds.

5.2. High-Level Analysis Methodology

We now present the high-level approach for solving the minimal satisfying cut-set enumeration problem. In the following sub-section, we describe how we instantiate this high-level approach for solving the guarantee-violating minimal secret state field identification problem (Definition 5.1). If the semantics of the monotone predicate $\mathcal{P}(\cdot)$ used in the minimal cut-set enumeration problem can be symbolically captured as a monotone Disjunctive normal form (DNF)/Conjunctive normal form (CNF) circuit/formula, then one can reduce the minimal cut-set enumeration problem to the *monotone Boolean dualization problem* [43] and use the Fredman–Khachiyan algorithm to solve it in quasi-polynomial time (*i.e.,* $\mathcal{O}(n^{o(\log(n))})$ where $|L| = n$) [44]. Unfortunately, the semantics of $\mathcal{P}(\cdot)$ in our context (precisely, capturing condition 2 in Definition 5.1) cannot be symbolically represented as a DNF formula, making the Fredman–Khachiyan algorithm inapplicable in our context.

If the value of n is small (*i.e.,* $n \leq 20$) and evaluating $\mathcal{P}(\cdot)$ is inexpensive, one can easily enumerate all subsets of L in $\mathcal{O}(2^n)$ and then return only those subsets of L that are minimal and satisfy $\mathcal{P}(\cdot)$. In our context (*i.e.,* $n = 11$), such an approach would have sufficed. This approach, however, does not take advantage of the fact that $\mathcal{P}(\cdot)$ is a monotone predicate and evaluating $\mathcal{P}(\cdot)$ is expensive. We use an efficient iterative deepening search to incrementally construct the $\mathcal{P}(\cdot)$-satisfying minimal subsets of L where the depth of the search captures the maximum allowed cardinality of the candidate subsets of L. This approach constructs smaller cardinality cut-sets before constructing larger ones. It considers larger cardinality subsets only when none of its subsets are minimal, $\mathcal{P}(\cdot)$-satisfying cut-sets. This can be efficiently checked by storing already seen $\mathcal{P}(\cdot)$-satisfying subsets in a trie data structure while considering a fixed, total order of the elements in L (*i.e.,* any permutation of the elements of L will suffice as a fixed, total order). If the algorithm finds a candidate cut-set, it checks its minimality by trying to greedily remove elements from the candidate and checking the resulting sets' $\mathcal{P}(\cdot)$ satisfaction. Our full algorithm (*i.e.,* Algorithm 1) is presented in §B.2.

5.3. Minimal Secret State Set Identification by Instantiating L and $\mathcal{P}(s)$ in Algorithm 1

We now discuss how we solve the guarantee-violating minimal secret state field set identification problem (Definition 5.1) by instantiating L and $\mathcal{P}(\cdot)$ in the Approach discussed in §5.2 for solving the problem in Definition 5.2.

```
struct ECM_app_secre_state_record {
    app_interchange_profile : static Byte [2];
    app_transaction_counter : static Byte [2];
    card_master_key : static Byte [32];
    card_private_key : static Byte [248];
    card_public_key : static Byte [248];
    cardholder_verification_methods : static Byte [1];
    expiration_date : static Byte [2];
    issuer_application_data : static Byte [32];
    records_signature : static Byte [248];
    token : static Byte [19];
    session_key : derived Byte [32];
    /* calculated from card_master_key and
        application_transaction_counter */
};
```

Listing 1: The 11 secret state fields of the underlying EMV protocol considered to be part of L. Each field has a designation of "*static*" or "*derived*" (*i.e.,* derivable from some secret fields and fresh nonces). Each field also has a size designation (*i.e.,* the number inside square brackets).

Instantiating L for Algorithm 1 based on the EMV protocol design. Listing 1 shows the list of secret state fields we consider to be part of L (also, $\mathbb{F}$ in Definition 5.1). Since the card profile in an ECM app contains fields beyond cryptographic keys (*e.g.,* Token, records, and ATC), we also consider such non-cryptographic-key fields in L. We include fields in L if they correspond to a cryptographic key (*e.g.,* C_{privK}, C_{pubK}, SK, C_{MK}) used in EMV, used to generate such a key (*e.g.,* ATC), used in direct or indirect (signature or other) validation (*e.g.,* records, anything in SDAD and AC), or identifies the account in question (*e.g.,* Token). We do not include nonces (*e.g.,* N_c, N_t) in L, as they are fresh, unique, and session-specific random value used for preventing replay attacks and cannot necessarily be used in other sessions. Note that although we omit certain elements fields from the EMV protocol, which could theoretically exclude a minimal cut-set, we manually verified our selection to ensure that all relevant secret state fields, which can influence the satisfaction of the two security properties (*i.e.,* $\Psi_{\texttt{simTxAuth}}$ and $\Psi_{\texttt{cloneResist}}$), are included in L.

Instantiating $\mathcal{P}(s)$ in Algorithm 1 based on the EMV protocol design. Instantiating the monotone predicate $\mathcal{P}(s)$ for all $s \subseteq L$ in Algorithm 1 based on the EMV protocol design conceptually reduces to checking whether Φ_1 or Φ_2 holds, where Φ_1 and Φ_2 are defined as follows.

1) $\Phi_1 \triangleq \mathcal{M}[s] \not\models^? \Psi_{\texttt{simTxAuth}}$ ($\triangleq$ means "*defined*");
2) $\Phi_2 \triangleq \mathcal{M}[s] \not\models^? \Psi_{\texttt{cloneResist}}$.

Simply put, $\mathcal{P}(s) \triangleq \Phi_1 \bigvee \Phi_2$. Precisely, $\mathcal{M}[s] \not\models^? \Psi_{\texttt{simTxAuth}}$ (resp., $\mathcal{M}[s] \not\models^? \Psi_{\texttt{cloneResist}}$) holds if and only if the underlying EMV protocol model $\mathcal{M}[s]$, specialized for an ECM app (not for a physical card), violates the $\Psi_{\texttt{simTxAuth}}$ (resp., $\Psi_{\texttt{cloneResist}}$) property, where the adversary has access to the values of the secret state fields corresponding to the set s. It is clear that checking whether $\mathcal{M}[s] \not\models^? \Psi_{\texttt{simTxAuth}}$ (resp., $\mathcal{M}[s] \not\models^? \Psi_{\texttt{cloneResist}}$) holds boil down to solving a cryptographic protocol verification problem instance (*à la*, Tamarin [45] and ProVerif [46]).

In our instantiation of the monotone predicate $\mathcal{P}(s)$, we use the Tamarin protocol verifier [45] for checking Φ_1 and Φ_2. Additionally, instead of using $\Phi_1 \bigvee \Phi_2$, we separately analyze Φ_1 and Φ_2. This corresponds to finding the minimal, satisfying cut-sets for $\Psi_{\texttt{simTxAuth}}$ and $\Psi_{\texttt{cloneResist}}$ separately. As hinted before, we use Tamarin prover as the reasoning tool of our choice because of the presence of a Tamarin EMV protocol model [8] for EMV physical card transactions, which we adopt for the ECM app. This demonstrates our claim that one can essentially reuse the model built for an Dolev-Yao style network adversary [23] to reason about ADV_{F} and ADV_{PB} adversaries without incurring substantial manual effort. Our approach has the following stages:

① Adapt an existing physical EMV card protocol model $\mathcal{M}_{\texttt{PCard}}$ developed [8] to support ECM apps, obtaining the EMV protocol model $\mathcal{M}_{\texttt{ECMapp}}$. [**Offline bootstrapping**: *one-time manual effort* (50 human-hours)]

② Parameterize $\mathcal{M}_{\texttt{ECMapp}}$ with $s \subseteq L$ to construct the model $\mathcal{M}_{\texttt{ECMapp}}(s)$ as a meta-program $\mathsf{MetaP} : 2^L \rightarrow \mathcal{M}^{\mathsf{Leak(s)}}_{\texttt{ECMapp}}$, which takes as input any $s \subseteq L$ and synthesizes a concrete instance $\mathcal{M}^{\mathsf{Leak(s)}}_{\texttt{ECMapp}}$ of $\mathcal{M}_{\texttt{ECMapp}}(s)$ in which the adversary has access to values of secret fields correspond to the the set s. [**Offline bootstrapping**: *one-time manual effort* (32 human-hours)]

③ Given a concrete leak set s during the execution of Algorithm 1, invoke $\mathsf{MetaP}(s)$ to generate $\mathcal{M}^{\mathsf{Leak(s)}}_{\texttt{ECMapp}}$. [**Automated**]

④ Use Tamarin to check whether $\mathcal{M}^{\mathsf{Leak(s)}}_{\texttt{ECMapp}} \models^{?} \Phi$ (*i.e.,* $\Phi \in \{\Psi_{\texttt{simTxAuth}}, \Psi_{\texttt{cloneResist}}\}$). If $\mathcal{M}^{\mathsf{Leak(s)}}_{\texttt{ECMapp}} \models \Phi$, return FALSE. Otherwise, return TRUE. [**Automated**]

We now describe Stages ① and ② of our approach. Owing to space constraints, we assume readers are familiar with the basics of Tamarin (*e.g.,* rewrite rules, persistent/linear facts). A brief overview of protocol modeling in Tamarin is provided in §B.1. For a more comprehensive treatment, please refer to the Tamarin manual [45].

Constructing ECM App Base Model $\mathcal{M}_{\texttt{ECMapp}}$ (Stage ①): We manually and faithfully implement all three phases (*i.e.,* initialization, transaction, and transaction authorization phases in Figure 1) of the underlying EMV protocol model relevant to an ECM app in Tamarin while consulting the existing model [8].

In general terms, our changes were mostly relevant to (1) adapting the model to the ECM environment and (2) implementing our adversary model.

(1) Adaption to the ECM Environment. While the existing EMV protocol model [8] for physical cards models the card PIN, our model does not include it. This is because ECM apps default to the OD-CVM (on-device cardholder verification method; *e.g.,* fingerprint, mobile PIN, face detection) and do not have a card PIN. In our model, we assume that OD-CVM is always verified successfully. Additionally, we do not distinguish between high- and low-value transactions, since this is relevant to other CVM methods. In terms of authentication, we only model CDA because it is the option used in ECM settings. Another difference is that we implement tokenization.

(2) Implementation of the Adversary Model. For both properties, we consider a model in which legitimate card holders only perform the card initialization, but do not perform any transactions. In contrast, we consider specific rules allowing adversarial wallets to receive secret state field values from legitimate wallets (discussed below), and use these leaked values to carry out transactions.

Parameterizing $\mathcal{M}_{\texttt{ECMapp}}$ with leak set s (Stage ②): We explain the leakage parameterization with an example in Figure 3. Rule 1 shows a simplified version of the EMV bootstrapping process where a fresh master key (cmk) of the card is generated, and an application transaction counter (atc_1) is initialized. Both of these are stored in the protocol state as persistent facts. A session key (sk) is then generated by applying a key derivation function $\mathsf{kdf}(\cdot,\cdot)$ on cmk and atc_1, which is then stored in the protocol state. Recall that $\mathsf{cmk}, \mathsf{atc}_1, \mathsf{sk} \in L$ signifying that they are part of the ECM app secret state we are interested in leaking (see Listing 1). Rules 5–7 essentially leak the value of cmk, atc, and sk, respectively, to the attacker as long as they are already computed, and are allowed by the leak set. The latter condition is guarded by a 0-ary persistent fact (a Boolean flag), such as !ATCLeakAllowed() when leaking the atc to the attacker in Rule 6. The rules 2–4, when present, unconditionally allow the leakage rules 5–7 to be enabled by generating the necessary 0-ary guard, persistent facts in their conclusions (*e.g.,* rule 3 generates the guard !ATCLeakAllowed(), which enables rule 6 for leaking atc to the attacker). Given a concrete leak set s, the meta-program $\mathsf{MetaP}(s)$ checks to see whether $\mathsf{atc} \in s$. If it is the case, it includes the rule 3 to allow its leakage.

5.4. Minimum Cut-set Results

The execution of our minimal secret state field set identification algorithm, to find minimal cut-sets for ECM-based EMV contactless protocol, successfully terminates for all instances of $\mathcal{M}^{\mathsf{Leak(s)}}_{\texttt{ECMapp}}$. The analysis identified four minimal sets of secret elements, as summarized in Table 3. In Tamarin, possession of a private key implies the ability to derive the corresponding public key. However, we present the minimal cut sets using C_{pubK} for clarity, although such occurrences are redundant when C_{privK} is present. Next, we discuss the implications of these findings.

Attack I: Attacking the Merchant (ADV_{F}). We identified a minimal cut set, denoted as SA_1 (see Table 3), which violates the simultaneous authentication property ($\Psi_{\texttt{simTxAuth}}$). This vulnerability enables an attacker to execute a fraudulent transaction by exploiting leaked credentials—obtaining goods or services without completing payment to the merchant. The exploitation works as follows.

1) The adversary leaks data elements in SA_1, which contains secret cardholder data and C_{privK}.
2) Using the leaked data, the adversary emulates the transaction up until the *GENERATE AC* command.
3) When responding to *GENERATE AC*, it (a) forces offline mode by setting CID to 0x40; (b) generates

① EMVGenAndDerive : [Fr(∼cmk)] → [!CardMasterKey($C, ∼cmk), !ATC($C, atc$_1$), !EmvSessionKey($C, kdf(∼cmk, atc$_1$))]

② EnableMasterKeyLeak : [ ] → [!MasterKeyLeakAllowed()]

③ EnableATCLeak : [ ] → [!ATCLeakAllowed()]

④ EnableSessionKeyLeak : [ ] → [!SessionKeyLeakAllowed()]

⑤ Leak_MasterKey : [!CardMasterKey(C, cmk), !MasterKeyLeakAllowed()] → [Out(cmk)]

⑥ Leak_ATC : [!ATC(C, atc), !ATCLeakAllowed()] → [Out(atc)]

⑦ Leak_SessionKey : [!EmvSessionKey(C, sk), !SessionKeyLeakAllowed()] → [Out(sk)]

Figure 3: An example showing the parameterization of the EMV protocol model of an ECM app with guarded leakage (**Rules** 2–7 in the blue-colored font are extensions to support leakage parameterization)

What are the resulting min-cut sets?

ID	Property	Mode	Min-Cut Set
SA_1	`simTxAuth`	Offline	{Token, exp-date, AIP, CVM, C_{pubK}, rec-sig, C_{privK}}
CR_1	`cloneResist`	Offline	{Token, exp-date, AIP, CVM, C_{pubK}, rec-sig, C_{privK}}
CR_2	`cloneResist`	Online	{Token, exp-date, AIP, CVM, C_{pubK}, rec-sig, SK}
CR_3	`cloneResist`	Online	{Token, exp-date, AIP, CVM, C_{pubK}, rec-sig, C_{MK}, ATC}

TABLE 3: Min-cut sets are identified by leaking various combinations of data elements and testing which ones break the security properties, starting from the single-element sets. The set of {Token, exp-date, AIP, CVM, C_{pubK}, rec-sig} are the elements in records.

a valid SDAD signed with C_{privK}, while providing an invalid AC into it.

4) Upon receiving the response from the card, the terminal validates the transaction with C_{pubK} and authorizes it without checking with the bank (**offline mode**).
5) At a later time, when the merchant sends invalid AC to the bank for validation, the bank declines it.

Conceptually, this attack can be compared to Basin *et al.*'s relay attack to fool the terminal [9]. The key difference is that, in our case, the attacker can directly use the extracted credentials to create a counterfeit virtual wallet and perform a fraudulent transaction, without needing to tamper with the transaction protocol in real-time through a man-in-the-middle (MiTM) attack.

Findings 6 (§5.4) Free-riding: By using leaked cardholder data and C_{privK}, it is possible to create a counterfeit virtual wallet that will always execute offline mode transactions, which will be accepted by the terminal but rejected by the bank at a later time.

Attack II: Attacking the cardholder (ADV_{PB}). We found three min-cut sets that broke the cloning resistance property ($\Psi_{\texttt{cloneResist}}$): CR_1, CR_2, and CR_3. The consequence is that an adversary can steal the credentials and keys from a legitimate wallet to create a cloned wallet, allowing them to execute transactions on the cardholder's behalf without their consent. Here, CR_1 is similar to SA_1 in that an adversary can create a counterfeit wallet that always to execute offline mode transactions and, thus, does not need to generate a valid AC. Note that, although CR_1 and SA_1 contains the same data elements, they represent two different traces that break different security properties. A consequence is that, like in SA_1, the transaction will eventually be rejected by the bank. This means that while the transaction can be traced back to the valid cardholder, their account will not be charged. Among all these cut-sets, the most interesting case is CR_2, as it signifies that, an attacker can simply use *derived credentials* (SK) to clone a card—which asserts the need for this work.

Findings 7 (§5.4) Cloning Cards: We found three different ways to clone cards—one in offline (CR_1) and two in online mode (CR_2, and CR_3). While CR_1 and CR_3 contains long-term cryptographic keys, CR_2 does not. This indicates the possibility of conduct fraudulent transactions with *derived* keys, which might not have the same level of protection as long-term keys.

6. Secret Leakage in Real-World ECM Apps

In this section, we investigate the possibility of exploiting real-world ECM apps by extracting the min-cut sets found in §5.4. First, we explore the possibility of locating the the min-cut sets in real-world ECM apps (§6.1). Then, we conduct a case study on an app to show that they are actually exploitable (§6.2). Finally, in §6.3, we present how this case study implicates other min-cut set leaking apps with no to low protections.

6.1. Locating Min-cut Set Data Elements.

In this section, we aim to answer: *In how many apps from our real-world ECM app corpus can we locate the min-cut set data elements?* Next, we present our methodology to answer this question and our findings.

Methodology. We reduce the problem of locating min-cut data elements to finding their usage in code—class names, fields, or method identifiers. Theoretically speaking, once identified, these locations can be instrumented to exfiltrate the corresponding values at runtime, specially, if the app lacks runtime anti-tampering protections as discussed in §4.3. To find the locations of these elements in the app,

we can simply string search in the apps' decompiled code using their names as the search query. However, an identifier representing the same data element could be named differently across the various apps' payment libraries. To overcome this challenge, we used an LLM (OpenAI's GPT-4o in August 2025) to generate variations of the data element name, resulting in between 1 (Token only had "token") and 72 names in total for each data element. We then manually validate the results to remove any FPs, *i.e.,* results that are most likely to be non-payment related.

Results. Table 4 summarizes our findings. We found 24/136 apps with exposed SA_1 and CR_1 elements. CR_2 appeared in only 3 apps, all of which are from the 24 apps. We could not find CR_3 in any app as it seems like C_{MK} is usually stored on the issuer server. These 24 apps have 82M downloads, indicating a broad user reach and significant real-world impact potential. Additionally, in 10/136 other apps, we found at least one min-cut set partially.

Our manual validation found FPs in only one ECM app. We found "token" and "sessionKey" in a chat module of a payment library *e.g., com.*****.android.chat*. However, this did not affect the number of min-cut sets found.

Min-Cut Set	Num Found
SA_1	24
CR_1	24
CR_2	3
CR_3	0

TABLE 4: Number of min-cut sets found in the 136 apps. All 24 SA_1 and CR_1 were identified in the same apps, and the 3 CR_2 were found in the same 24 apps. We found partial min-cut set matches in 10 other apps.

Finding 8: We identified 2/3 minimum cut sets in 24/136 ECM apps, and found all 3 minimum cut sets in 3/24 apps. We identified 2 min-cut sets in 24/136 ECM apps, and found a third min-cut set in 3/24 apps.

6.2. End-to-End Exploitation

For a given ECM app with identified min-cut set data elements (§6.1), demonstrating exploitability requires two steps: (i) exfiltrating data elements from the app, and (ii) using those elements to create a counterfeit wallet and execute fraudulent transactions. However, to conduct this experiment ethically, one needs to ensure that the experiment's impact is *reasonably* contained. For this purpose, previous studies [8] used their own cards (cardholders) and terminals (merchants). This is infeasible for our case, as the 24 apps with leaking min-cut sets originate from different geographic locations than those of the authors. Thus, we are required to initialize apps in an isolated environment to separate them from real-world databases and systems, both for functional and ethical reasons.

Experimental Setup. Here, we discuss how we initialize a given ECM app in an isolated environment and create a terminal application to demonstrate the exploitation.

Initializing ECM app. Initialization in the isolated testbed proceeds in three steps: (1) reverse-engineer the app to identify REST endpoints and payloads, (2) implement *stubs* that emulate the API requests/responses, and (3) initialize and exercise the app against these emulations, ensuring no connection to live services. Note that failing to stub even one API would render this effort unsuccessful. Thus, to keep this effort manageable and fail-safe, we picked 1 of the 3 ECM apps for which we were able to locate all three min-cut data elements (SA_1, CR_1, and CR_2).

Each response body is either a JSON object or a base64-encoded binary object, whose format and content are determined by debugging the app's source code. In the end, *over a period of 3 months, we stubbed 18 REST API calls with 867 JSON key-value pairs exchanged between the server and the app*. For investigating and stubbing REST APIs, we used Fiddler Everywhere [47]. We used JEB [40], an Android reverse engineering tool, to understand the business logic of the selected ECM app.

Creating a terminal. To ensure that our terminal is also running in an isolated environment, we develop a terminal application in Java, by consulting EMV contactless specifications [48]. Our terminal exchanges APDU commands and responses through a physical contactless reader.

Exploitation I: Attacking the Merchant (ADV_{F}). The exploitation proceeds in two steps: (1) exfiltrate the min-cut set elements from the legitimate wallet, and (2) construct a counterfeit wallet that uses those leaked credentials to force an offline transaction producing a valid SDAD (generated with C_{privK}) but an invalid AC (as outlined in §5.4). Note that if an attacker can modify the original wallet app to produce the counterfeit, then the modified app will already contain the code to receive the min-cut elements from the server, allowing the first step to be bypassed. We leverage this insight to create a counterfeit wallet for this exploitation. Specifically, we modify the CID by disassembling and modifying the corresponding Smali code from "`const/16 v2, 0x80`" to "`const/16 v2, 0x40`" to produce the counterfeit wallet. An experimental evaluation to conduct a fraudulent transaction with the terminal indicates success.

Exploitation II: Attacking the cardholder (ADV_{PB}). This exploitation also has two steps: (1) remotely extract min-cut set elements from the victim's wallet and (2) use the extracted elements to create a *cloned* wallet. To demonstrate remote extraction, we run a dynamic instrumentation agent in the phone that hooks the app's execution and sends the leaked information to a computer. In Listing 2, we show how `toString()` method reveals SK and ATC. Then, we create a simple app populated with the leaked data elements and successfully conduct a transaction via our terminal.

```
public c d() { return this.c; } // ATC
public c e() { return this.a; } // UMDSessionKey
public String toString() {
   return "...UMDSessionKey=" + this.a.c() +...+ "␣
       ATC=" + this.c.c() +...; }
```

Listing 2: Code snippet from class `f` of package `com.mastercard.mcbp.core.mpplite`.

6.3. Implications for min-cut set leaking apps

In light of our case study (§6.2), it is crucial to examine how the 24 apps leaking min-cut set data elements (§6.1) protect themselves against SecStLeak. Table 4 summarizes the results of our analysis of their adopted protection mechanisms. We found that 15 apps did not use TEE, 12 apps run on root, 14 have interceptable network communication, 14 are repackable, 23 are hookable, and 1 permits running in debug mode. For those with TEE usage, we found that 5 apps run on root, 6 have interceptable network communication, and 6 are repackable. Although using TEEs protects the secret keys, most ECM apps do not import secret keys into TEEs securely via `WrappedKeyEntry` (§4.3) This enables adversaries to repackage the apps to turn off TEE capabilities and make payments.

These findings indicate that, by following our exploitation methodology (Exploitation I, in §6.2), any legitimate users of the *14 repackable* apps can immediately launch attacks to enjoy services or goods without paying. It also indicates that users of these apps using rooted devices are also in immediate danger of card cloning attacks (Exploitation II, in §6.2). Given that 8/24 of the apps are from the underprivileged part of the world, it is not unlikely that a significant user base of those apps would use rooted devices or devices with privilege escalation vulnerabilities [49]. Perhaps the most important implication of this finding is that any newly discovered privilege-escalation vulnerability could quickly enable large-scale extraction of wallet secrets for card cloning-based exploitations.

Use of Protection Mechanisms	#Apps
No TEE	15
No Root Checking	12
No Network Integrity Checking	14
No Anti-Repackaging	14
No Anti-Hooking	23
No Anti-Debug	1
TEE + No Root Checking	5
TEE + No Root Checking + No Anti-Hooking	5
TEE + No Network Integrity Checking	6
TEE + No Anti-Repackaging	6

TABLE 5: Mechanisms not implemented in the 24 apps that are leaking secret state data elements.

Finding 9: Our case study and the protection analysis of 24 apps that leak min-cut set data elements indicate that (1) legitimate users of these apps can immediately exploit terminals and (2) users with rooted devices or devices with privilege escalation vulnerability are under immediate risk of card cloning-based exploits.

7. Discussion

Limitations. Although our approach automatically generates the models starting from the base model, our base model is created manually. It contains simplified abstractions of the protocol in certain parts, which might not capture all the details that could exist in the real world. There may also exist other important security properties that we did not check in our model. In addition, our formalized cloning-resistance lemma is restrictive, which may miss discovering attacks in which observing fully or partially executed transactions from the legitimate wallet can help launch follow-up attacks. Another possible limitation is the gap between our symbolic model and real-world ECM app implementations. As a result of not faithfully following the protocol design or using weaker cryptographic constructs, ECM apps may introduce additional implementation-level vulnerabilities undetectable by our meta-level analysis. Lastly, we could not check the exploitability of each app because such a study would require installing and initializing each of the apps locally in an isolated environment, which entails non-trivial efforts for non-users like us.

Threats to Validity. Even though we made our best effort to ensure the model is correct (i.e., regularly discussed and validated by multiple authors, compared the implementation with the decompiled app's code), there is no way to guarantee that it is bug-free. Our demonstration using the real-world app involves custom-built infrastructure for ethical reasons, which may also not reflect real-world implementations. Lastly, our results in §4 may be different if one were to spend significant efforts to break each app. Nonetheless, it represents a conservative estimate.

8. Related Work

EMV Protocol. There are many works related to the EMV protocol, which include designing new EMV-compliant payment protocols [4], [50], revealing flaws [8], [10], [51], [52], proposing fixes [8], [51], and formal verification of the EMV protocol [3], [4], [5], [6], [7], [8]. Unlike our work, they all assume a Dolev-Yao style threat model, where the network adversary can eavesdrop, synthesize, and intercept messages [23]. For example, Basin *et al.* assumes a Dolev-Yao adversary that can listen, block, inject, and modify the transmitted data in the contactless channel in the chip card and terminal [8]. To emulate the lack of tamper-proof guarantees, we instead assume a threat model where the adversary is capable of leaking secret data elements similar to the Bellare-Rogaway adversary [18]. This threat model difference requires a novel formal modeling and analysis, which we discussed in §5.

Formal Verification in Other Domains. Applications of formal verification span numerous domains beyond EMV protocols, including cellular networks [53], [54], [55], micro-controller secure boot protocols [56], digital rights management protocols [57], mobile messaging app protocols [58], [59], and networking protocols [60], [61]. A work with an intuition similar to ours is Basin *et al.*'s investigation of the 5G authentication protocol [54]. While we share a similar goal of identifying how security properties depend on security assumptions, our approach is more automated and can be generalized to algorithmically enumerate the minimal leakage cut sets for any protocols.

Device and App Protection. Many works exist on Android rooting, including the detection and evasion of rooting [15], [31], detection of data leakage in rooted devices [62], detection of root exploits [63], and effectiveness of root detection [64]. There are also many works on repackaging attacks [65], [66], [67], [68], [69] and network integrity measurements [64], [70]. Code obfuscation is also widely studied by researchers as well, including obfuscation techniques for Java [71], [72], [73], [74], malware [75], [76], and IP protection [77]. Works specific to Android code obfuscation have focused on obfuscation techniques [78], [79], obfuscation detection [80], [81], [82], [83], [84], obfuscated malware-app detection [85], [86], [87], anti-obfuscated-malware tools [83], [88], [89], and deobfuscation [90], [91]. Unlike our results in §4, none of these studies present findings specific to payment apps.

Financial mobile and web apps. Increase in financial app and web usage has induced significant research work in the field [28], [64], [92], [93], [94], [95], [96]. Prior work has, for example, investigated the security of branchless apps [92], [96] and digital credit applications [94]. Beyond security analysis, studies have also examined compliance across different facets of the payment ecosystem, including financial apps with respect to PCI-DSS information storage policies [95] and websites in relation to PCI-DSS network security requirements [28].

9. Conclusion

In this work, we studied the extent to which real-world EMV contactless apps are susceptible to secret state leakage issues due to not adopting the prescribed defenses. We found that a significant majority are potentially vulnerable. Then, we built a meta-model analysis framework to investigate what secret data elements need to be protected to guarantee protocol security, finding 4 min-cut sets on 2 threat models. Lastly, we located the min-cut set elements in the code of these apps and successfully demonstrated exploitation potentials with an end-to-end attack demonstration of one app in an isolated, controlled environment. We conclude that, when developing payment apps, one needs to carefully consider the *local adversary* to preserve underlying guarantees of the protocol.

LLM Usage Considerations

We use LLMs in only two instances, both for automation. First, in §4, we used an LLM in EMVResilienceChecker to detect messages about root detection, essentially acting as a classifier. We validate its performance, like one would any other machine learning model. Second, we use it to generate variations of search terms to find the location of min-cut elements in decompiled code. Using an LLM here not only speeds up an otherwise manual-intensive task but also ensures a more comprehensive set of variations.

Ethics Considerations

Responsible Disclosure. We have so far disclosed our findings to vendors of apps for which we located one or more min-cut sets and did not implement TEE or root checking, as these apps are the ones most susceptible to SecStLeak attacks. In total, we contacted 16 vendors and received confirmation from 4. We had an in-depth meeting with the vendor whose app we used in our case study (§6.2). During this meeting, we learned that they relied on reverse engineering being difficult instead of implementing more robust software protection mechanisms. They also mentioned that their app was recently exploited in a way that involved device rooting and bypassing certificate pinning, highlighting the real-world relevance of EMV-recommended software protection mechanisms.

We study and evaluate the usage of software protection mechanisms in 136 real-world ECM apps. To prevent facilitating real-world exploitations, we keep these results anonymous. In addition, we use one of these apps to prove the exploitability of the proposed attacks. To ensure that we do not interact with real-world systems when conducting this demonstration, we develop a proxy server and terminal to operate it in an isolated environment. Furthermore, we responsibly disclose our findings to the app vendors.

Acknowledgments

This work was partially supported by University of Arizona's IT4IR TRIF and TRIF NSS programs, State University of New York's Empire Innovation Program, and NSF under grants 2145631 and 2215017. We thank Robert Lorch for carefully reviewing our Tamarin model and providing insightful feedback. We also thank Talha Abrar, Muhammad Bilal, Sonja Brown, and Priya Kaushik for their thoughtful comments on our writing.

References

[1] Statista, "Mobile wallet transactions by region 2025," https://www.statista.com/statistics/1227576/mobile-wallet-transactions-worldwide/.

[2] S. C. Alliance, "The mobile payments and nfc landscape: A us perspective," *Smart Card Alliance*, 2011.

[3] J. De Ruiter and E. Poll, "Formal analysis of the emv protocol suite," in *Theory of Security and Applications*. Springer, 2012.

[4] A.-I. Radu *et al.*, "Practical emv relay protection," in *IEEE Symposium on Security and Privacy*, 2022.

[5] T. Chothia *et al.*, "Modelling and analysis of a hierarchy of distance bounding attacks," in *USENIX Security Symposium*, 2018.

[6] S. Mauw *et al.*, "Post-collusion security and distance bounding," in *ACM Computer and Communications Security*, 2019.

[7] L.-A. Galloway and T. Yunusov, "First contact: New vulnerabilities in contactless payments," *Black Hat Europe*, 2019.

[8] D. Basin, R. Sasse, and J. Toro-Pozo, "The emv standard: Break, fix, verify," in *IEEE Symposium on Security and Privacy*, 2021.

[9] D. Basin *et al.*, "Card brand mixup attack: bypassing the pin in non-visa cards by using them for visa transactions," in *USENIX Security Symposium*, 2021.

[10] ——, "Inducing authentication failures to bypass credit card pins," in *USENIX Security Symposium*, 2023.

[11] M. Roland and J. Langer, "Cloning credit cards: A combined pre-play and downgrade attack on EMV contactless," in *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, 2013.

[12] "Android | frida • A world-class dynamic instrumentation framework," https://frida.re/docs/android/.

[13] Wikipedia contributors, "Principle of Least Effort — Wikipedia, The Free Encyclopedia," 2025, [Accessed: 2025-01-08]. [Online]. Available: https://en.wikipedia.org/wiki/Principle_of_least_effort

[14] "Keyinfo," https://developer.android.com/reference/android/security/keystore/KeyInfo.

[15] L. Nguyen-Vu, N.-T. Chau, S. Kang, and S. Jung, "Android rooting: An arms race between evasion and detection," *Security and Communication Networks*, no. 1, p. 4121765, 2017.

[16] Q. Liu and Others, "Digital rights management for content distribution," in *ACSW Frontiers*, 2003.

[17] R. Wang *et al.*, "Steal this movie: Automatically bypassing DRM protection in streaming media services," in *USENIX Security Symposium*, 2013.

[18] M. Bellare *et al.*, "Entity authentication and key distribution," in *Annual international cryptology conference*. Springer, 1993.

[19] SPRLab, "Emv contactless study," https://github.com/sprlab/EMV-Contactless-Study.

[20] EMV Co., "Emv issuer and application security guidelines," https://www.emvco.com/specifications/emvissuer-and-application-security-guidelines/, Tech. Rep., 2023.

[21] "EMV Payment Tokenisation Specification: Technical Framework," https://www.emvco.com/terms-of-use/?u=/wp-content/uploads/documents/EMVCo-Payment-Tokenisation-Specification-Technical-Framework-v2.0-1.pdf, 2017.

[22] J. van den Breekel *et al.*, "Emv in a nutshell," KPMG, IBM Research Zurich, Radboud University Nijmegen, Tech. Rep., 2016.

[23] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, 1983.

[24] J. Rudie, Z. Katz, S. Kuhbander, and S. Bhunia, "Technical analysis of the nso group's pegasus spyware," in *Computational Science and Computational Intelligence*, 2021.

[25] EMV Co., "Software-based mobile payment security requirements," https://www.emvco.com/resources/emv-mobile-payment-software-based-mobile-payment-security-requirements/, Tech. Rep., 2020.

[26] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical society*, no. 2, 1953.

[27] S. Rahaman *et al.*, "Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects," in *ACM Computer and Communications Security*, 2019.

[28] S. Rahaman, G. Wang, and D. Yao, "Security certification in payment card industry: Testbeds, measurements, and recommendations," in *ACM Computer and Communications Security*, 2019.

[29] "Android keystore system," https://developer.android.com/privacy-and-security/keystore.

[30] S. a. Arzt, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM Sigplan Notices*, 2014.

[31] S.-T. Sun *et al.*, "Android rooting: Methods, detection, and evasion," in *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2015.

[32] A. Kostina, M. D. Dikaiakos, D. Stefanidis, and G. Pallis, "Large language models for text classification: Case study and comprehensive review," *arXiv preprint arXiv:2501.08457*, 2025.

[33] "meta-llama/llama-3.2-3b-instruct," https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct.

[34] "Zygisk: What is it? how to download and use it?" https://teamandroid.com/zygisk-magisk/, accessed: 2025.

[35] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *International Conference on Mining Software Repositories*, 2016.

[36] "Host-based card emulation overview | Connectivity." [Online]. Available: https://developer.android.com/develop/connectivity/nfc/hce

[37] S. Y. Mahmud *et al.*, "Cardpliance:PCIDSS compliance of android applications," in *USENIX Security Symposium*, 2020.

[38] D. S. Guamán *et al.*, "Automated gdpr compliance assessment for cross-border personal data transfers in android applications," *Computers & Security*, p. 103262, 2023.

[39] N. Higi, https://github.com/shroudedcode/apk-mitm, 2023.

[40] "JEB Decompiler," https://www.pnfsoftware.com/jeb/android.

[41] T. Chothia *et al.*, "Relay cost bounding for contactless emv payments," in *Financial Cryptography and Data Security*, 2015.

[42] "Kernel 8 guidance," https://www.emvco.com/specifications/contactless-specification-for-payment-systems-kernel-8-guidance-document-2/.

[43] T. Eiter *et al.*, "New results on monotone dualization and generating hypergraph transversals," in *ACM Symposium on Theory of Computing*, 2002.

[44] M. Fredman *et al.*, "On the complexity of dualization of monotone disjunctive normal forms," *J. Algorithms*, p. 618–628, 1996.

[45] D. Basin *et al.*, "Tamarin prover manual," https://tamarin-prover.com/manual/master/tex/tamarin-manual.pdf, acessed: 2024.

[46] B. Blanchet *et al.*, "Proverif 2.00: automatic cryptographic protocol verifier, user manual and tutorial," 2018.

[47] "Fiddler everywhere | debugging proxy for mac, linux, windows," https://www.telerik.com/fiddler/fiddler-everywhere.

[48] "Book c-2 kernel 2 specification." [Online]. Available: https://www.emvco.com/specifications/book-c-2-kernel-specification/

[49] A. Diallo *et al.*, "Security evaluation of android apps in budget african mobile devices," *arXiv preprint arXiv:2509.18800*, 2025.

[50] V. Cortier *et al.*, "Designing and proving an emv-compliant payment protocol for mobile devices," in *IEEE European Symposium on Security and Privacy*, 2017.

[51] S. J. Murdoch *et al.*, "Chip and pin is broken," in *IEEE Symposium on Security and Privacy*, 2010.

[52] M. Emms *et al.*, "Harvesting high value foreign currency transactions from emv contactless credit cards without the pin," in *ACM Computer and Communications Security*, 2014.

[53] S. Hussain, O. Chowdhury, S. Mehnaz, and E. Bertino, "Lteinspector: A systematic approach for adversarial testing of 4g lte," in *Network and Distributed Systems Security Symposium*, 2018.

[54] D. Basin *et al.*, "A formal analysis of 5g authentication," in *ACM Computer and Communications Security*, 2018.

[55] R. Miller, I. Boureanu, S. Wesemeyer, and C. J. Newton, "The 5g key-establishment stack: In-depth formal verification and experimentation," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022.

[56] Z. Tao, A. Rastogi, N. Gupta, K. Vaswani, and A. V. Thakur, "DICE*: A formally verified implementation of DICE measured boot," in *USENIX Security Symposium*, 2021.

[57] S. Delaune, J. Lallemand, G. Patat, F. Roudot, and M. Sabt, "Formal security analysis of widevine through the W3C EME standard," in *USENIX Security Symposium*, 2024.

[58] F. Linker, R. Sasse, and D. Basin, "A formal analysis of Apple's iMessage PQ3 protocol," in *USENIX Security Symposium*, 2025.

[59] C. Cremers, C. Jacomme, and A. Naska, "Formal analysis of Session-Handling in secure messaging: Lifting security from sessions to conversations," in *USENIX Security Symposium*, 2023.

[60] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, "A comprehensive symbolic analysis of tls 1.3," in *ACM Computer and Communications Security*, 2017.

[61] C. Cremers *et al.*, "Formal analysis of SPDM: Security protocol and data model version 1.2," in *USENIX Security Symposium*, 2023.

[62] L. Casati and A. Visconti, "The dangers of rooting: data leakage detection in android applications," *Mobile Information Systems*, no. 1, p. 6020461, 2018.

[63] I. Gasparis, Z. Qian, C. Song, and S. V. Krishnamurthy, "Detecting android root exploits by learning from root providers," in *USENIX Security Symposium*, 2017.

[64] T. Kim, H. Ha, S. Choi, J. Jung, and B.-G. Chun, "Breaking ad-hoc runtime integrity protection mechanisms in android financial apps," in *ACM Asia Computer and Communications Security*, 2017.

[65] L. Luo *et al.*, "Repackage-proofing android apps," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016.

[66] K. Chen *et al.*, "Leveraging information asymmetry to transform android apps into self-defending code against repackaging attacks," *IEEE Transactions on Mobile Computing*, no. 8, 2017.

[67] A. Merlo *et al.*, "You shall not repackage! demystifying anti-repackaging on android," *Computers & Security*, 2021.

[68] S. Rastogi *et al.*, "Android applications repackaging detection techniques for smartphone devices," *Procedia Computer Science*, 2016.

[69] J.-H. Jung *et al.*, "Repackaging attack on android banking applications and its countermeasures," *Wireless Personal Communications*, 2013.

[70] S. Pourali *et al.*, "Racing for TLS Certificate Validation: A Hijacker's Guide to the Android TLS Galaxy," in *USENIX Security Symposium*, 2024.

[71] J.-T. Chan and W. Yang, "Advanced obfuscation techniques for java bytecode," *Journal of systems and software*, no. 1-2, 2004.

[72] Y. Sakabe *et al.*, "Java obfuscation approaches to construct tamper-resistant object-oriented programs," *IPSJ Digital Courier*, 2005.

[73] T.-W. Hou *et al.*, "Three control flow obfuscation methods for java software," *IEE Proceedings-Software*, no. 2, 2006.

[74] M. Ceccato *et al.*, "A large study on the effect of code obfuscation on the quality of java code," *Empirical Software Engineering*, 2015.

[75] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, 2010.

[76] J.-M. Borello and L. Mé, "Code obfuscation techniques for metamorphic viruses," *Journal in Computer Virology*, no. 3, 2008.

[77] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *IEEE Transactions on Software Engineering*, no. 8, 2002.

[78] V. Balachandran *et al.*, "Control flow obfuscation for android applications," *Computers & Security*, 2016.

[79] S. Aonzo *et al.*, "Obfuscapk: An open-source black-box obfuscation tool for Android apps," *SoftwareX*, 2020.

[80] D. Wermke *et al.*, "A large scale investigation of obfuscation use in google play," in *Computer Security Applications Conference*, 2018.

[81] S. Dong *et al.*, "Understanding android obfuscation techniques: A large-scale investigation in the wild," 2018.

[82] P. Wang *et al.*, "Software protection on the go: A large-scale empirical study on mobile app obfuscation," in *IEEE/ACM International Conference on Software Engineering*, 2018.

[83] L. Glanz *et al.*, "Codematch: obfuscation won't conceal your repackaged app," in *Foundations of Software Engineering*, 2017.

[84] M. Kühnel, M. Smieschek, and U. Meyer, "Fast identification of obfuscation and mobile advertising in mobile malware," in *2015 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 2015.

[85] Z. Li *et al.*, "Obfusifier: Obfuscation-resistant android malware detection system," in *Security and Privacy in Communication Networks*. Springer, 2019.

[86] M. D. Preda and F. Maggi, "Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology," *Journal of Computer Virology and Hacking Techniques*, 2017.

[87] M. Protsenko and T. Müller, "Pandora applies non-deterministic obfuscation randomly to android," in *International Conference on Malicious and Unwanted Software*, 2013.

[88] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth attacks: An extended insight into the obfuscation effects on android malware," *Computers & Security*, 2015.

[89] P. Faruki *et al.*, "Evaluation of android anti-malware techniques against dalvik bytecode obfuscation," in *International Conference on Trust, Security and Privacy in Computing and Communications*, 2014.

[90] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of android applications," in *ACM Computer and Communications Security*, 2016.

[91] S. K. Udupa *et al.*, "Deobfuscation: Reverse engineering obfuscated code," in *Working Conference on Reverse Engineering*, 2005.

[92] B. Reaves *et al.*, "Mo (bile) money, mo (bile) problems: Analysis of branchless banking applications," 2017.

[93] C. W. Munyendo, Y. Acar, and A. J. Aviv, ""desperate times call for desperate measures": User concerns with mobile loan apps in kenya," in *IEEE Symposium on Security and Privacy*, 2022.

[94] J. Bowers *et al.*, "Characterizing security and privacy practices in emerging digital credit applications," in *Security and Privacy in Wireless and Mobile Networks*, 2019.

[95] S. Y. Mahmud *et al.*, "Cardpliance: {PCI} {DSS} Compliance of Android Applications," in *USENIX Security Symposium*, 2020.

[96] S. Castle *et al.*, "Let's talk money: Evaluating the security challenges of mobile money in the developing world," in *Symposium on Computing for Development*, 2016.

[97] A. Pradeep *et al.*, "A comparative analysis of certificate pinning in android & ios," in *ACM Internet Measurement Conference*, 2022.

[98] A. Cortesi, M. Hils, and T. Kriechbaumer, "mitmproxy," https://github.com/mitmproxy/mitmproxy.

[99] "cert-fixer," https://github.com/pwnlogs/cert-fixer.

[100] S. Dai *et al.*, "Networkprofiler: Towards automatic fingerprinting of android apps," in *IEEE InfoCom*, 2013.

[101] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Tech. Rep., 1997.

Appendix A. EMVResilienceChecker Detection Methods

TEE Detection. EMVResilienceChecker uses Soot to look for invocations of the following TEE APIs in each app: `WrappedKeyEntry`, *isInsideSecureHardware()*, and *getSecurityLevel()* [29] [2]. An app is compliant if these APIs are invoked in developer-defined packages or within contactless payment-related libraries (manually curated, *e.g.,* Visa, Mastercard, etc.).

Root checking. Details in §4.1.

Network Integrity. We use an MITM-based dynamic analysis approach to check which app checks for network integrity [97], leveraging [98]. To setup the device, we copy the MITM certificate into the device's system store [99]. Apps often send remote API calls right at launch [100], so we collect them for 10s right after launch. We collect background processes after rebooting the app prior to app launch and remove them from our findings. If the app does not trust the mitmproxy certificate for a given API call, mitmproxy reports errors. If no such error is reported for a single connection attempt or the connection was successful, we assume that the app trusted the MITM certificate.

We apply the following rules to determine the final result. First, if all connection attempts were trusted, then the app trusts the system store blindly and does not have network integrity checking. Second, if no connection attempts were trusted, then the app does not trust the system store blindly and uses certificate pinning. Third, if only a subset of the connection attempts were not trusted, then it is likely that certificate pinning was applied to only select APIs. Thus, we consider the app to have partial network integrity checking. Since we are detecting non-compliance, we only consider the first case as non-compliant, *i.e.,* all connection attempts are trusted. For the second and third cases, we attempt to bypass certificate pinning by instrumenting the app and then re-evaluate the resulting app.

Anti-Repackaging. EMVResilienceChecker decodes the app, inserts a text file as a form of modification, rebuilds the app, installs the repacked the app on our device, and runs it. An app is not compliant if we can run it successfully (launches and does not crash) after repackaging. Since failure can occur at any step during the repackaging process, we note it accordingly.

Anti-Hooking. EMVResilienceChecker launches the app and attempts to hook into its main activity, as it is easily identifiable in each app. We modify the main activity to send a log message, indicating a successful hook and non-compliance. Since the main activity launches instantaneously and all apps have been verified to launch successfully via the main activity, we consider the hook unsuccessful if such message is not received by the callback in 5s.

Anti-Debugging. EMVResilienceChecker decompiles the APK and parses the manifest to check if the `debuggable` flag is set to false or not set at all, in which case it defaults to false.

Code Obfuscation. EMVResilienceChecker uses JEB, an industrial-grade tool for reverse engineering apps to detect string encryption, reflection, control flow, and virtualization [40]. Upon detection, JEB places a comment on the method indicating as such, which is used as the signal.

Since JEB does not detect identifier renaming and there is no definitive way to detect it, we implement a heuristic-based detector. For each identifier in the decompiled code, we preprocess the identifier by splitting it into parts by camel-case, PascalCase, and by numbers and symbols (*e.g.,* `isCar` becomes [*is, car*] and `ab0c` becomes [*ab, 0, c*]). We then remove all non-English words. With the two parts, we determine if the identifier is renamed using two heuristics based on the fact that identifier renaming results in short and/or meaningless names [101]. First, we check if the average length of the English parts are less than 2.5 characters long. We chose the threshold of 2.5 because 95% of the nouns in the Brown corpus are longer than 3 characters and 95% of the verbs are longer than 2 (*e.g.,* `isCar`). Second, we check if the majority of the parts are not in the Brown corpus. If either of the two heuristics are true, then the identifier is likely renamed. Finally, we determine the percentage of identifiers renamed in the decompiled code. For boolean labeling, we consider the app to use IR if the result (0-1.0) is greater than 0.5.

Appendix B. Tamarin Model

B.1. Modeling Protocols in Tamarin

We now provide a terse introduction to modeling cryptographic protocols in Tamarin's input language.

Terms. To understand Tamarin's protocol modeling discipline, one has to first understand the notion of a ***first-order term***. A term in Tamarin is used to capture protocol data (*e.g.,* messages, keys, nonces). More technically, a term is a symbolic expression constructed from uninterpreted *constants* (*e.g.,* $\mathtt{\$A}$), *variables* (*e.g.,* $\sim\mathtt{ltk}$), and *function symbols* applied to other terms (*e.g.,* $\mathtt{hash}(\mathtt{m})$ denoting the hash of a message m). Tamarin has support for both built-in functions (*e.g.,* $\mathsf{hash}(\mathsf{m}), \mathsf{pk}(\mathsf{sk}), \mathsf{sign}(\mathsf{m},\mathsf{sk}), \mathsf{verify}(\mathsf{sig},\mathsf{m},\mathsf{pk})$) and user-defined functions, along with their equations (*e.g.,* $\mathsf{sdec}(\mathsf{senc}(\mathsf{m},\mathsf{k}),\mathsf{k}) = \mathsf{m}$ signifying decryption of a message's encryption results in the original message).

Facts. The next important concept in Tamarin is that of a ***fact***. Facts can be viewed as special datatype constructors applied to zero or more first-order terms. Facts come in the following categories: *linear facts* (*e.g.,* $\mathtt{StagedKey}(\mathtt{\$A}, \sim\mathtt{ltk})$ in Figure 4); *persistent facts* (*e.g.,* $!\mathtt{Ltk}(\mathtt{\$A}, \sim\mathtt{ltk})$); *action facts* (*e.g.,* $\mathtt{KeyGen}(\mathtt{\$A}, \mathsf{pk}(\sim\mathtt{ltk}))$ where $\mathsf{pk}(\sim\mathtt{ltk})$ is a symbolic term of applying the built-in function $\mathsf{pk}(\cdot)$ over the fresh variable $\sim\mathtt{ltk}$). Some linear facts have fixed interpretation in Tamarin, such as: $\mathsf{In}(m)$ (*i.e.,* message m is sent over the public-channel); $\mathsf{Out}(m)$

2. Note that *isInsideSecureHardware()* is deprecated and replaced by *getSecurityLevel()* in API level 31 [14].

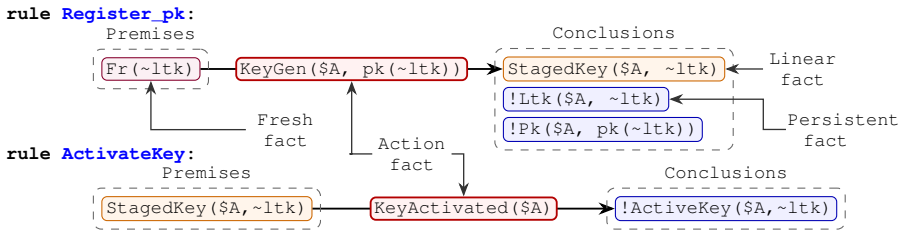

Figure 4: An example of modeling protocol steps as Tamarin-style multi-set rewriting rules over first-order terms.

(*i.e.,* message m is received over the public-channel and the adversary knows about m); $\mathsf{Fr}(\sim x)$ (*i.e.,* x is a freshly generated). Users can also introduce their own uninterpreted facts, such as the persistent fact $!\mathtt{Ltk}(\$\mathtt{A}, \sim \mathtt{ltk})$ in Figure 4.

Tamarin protocol model. A Tamarin protocol model has the following sections: *imports and definitions*; *multiset-rewriting rules*; *properties*. The imports and definitions essentially contain the built-in function symbols used in the model, and the user-defined function symbol signatures and their option equations. The multiset-rewriting rules (or, just ***rules***) capture the protocol execution and attacker interference. The properties are first-order logic formulas capturing the desired security and privacy properties to be verified on the execution trace of the protocol under analysis (described as rules) with respect to the adversary model.

Modeling protocol steps as rules. A rule in Tamarin has four sections: *name of the rule*; *premise*; *action fact list*; *conclusion*. The premise, which is a list of facts, describes the precondition under which this rule can be applied. The action fact list captures the list of uninterpreted action facts that will be added to the protocol's execution trace when this rule is executed. Without the loss of generality, one can only refer to action facts (and, some built-in facts such as built-in adversary-knowledge facts such as $\mathsf{KU}(x)$ and freshness facts such as $\mathsf{Fr}(\sim x)$, when describing properties. The conclusion, also a list of facts, captures what new facts are generated when this rule is executed. The semantics of a rule is that if the current state of the protocol (*i.e.,* a multiset of facts) contains facts that match the premise of a rule, then one can execute the rule, which entails recording the action facts in the trace, and extending the protocol state with the new facts mentioned in the rule conclusion. If a fact in the current state is designated as a linear fact, then one removes that fact from the state when that fact matches with the premise of a rule and the rule is executed. Persistent facts capture persistent state information of a protocol, and they are retained during execution.

Example. Consider the following toy protocol example:

1) $\langle \mathsf{pubKey}, \mathsf{secKey} \rangle \leftarrow \mathsf{AssymetricKeyPairGen}()$ and register the public key pubKey.
2) Activate the corresponding secret key secKey for use.
3) Use secKey for other operations only after activation.

The steps (1) and (2) of the above protocol modeled in Tamarin is shown in Figure 4. Each step becomes a rule in Tamarin (Step 1 becomes the rule Register_pk and Step 2 becomes the rule ActivateKey) where $\sim\mathtt{ltk}$ corresponds to secKey whereas $\mathtt{pk}(\sim\mathtt{ltk})$ corresponds to pubKey. To maintain that the rules are ordered in the same way as in the original protocol we use the linear fact $\mathtt{StagedKey}(\$\mathtt{A}, \sim\mathtt{ltk})$. The first rule generates the linear fact, which is used in the second rule's premise.

B.2. Minimal Satisfying Cut Set Identification

Algorithm 1 below shows our approach for identifying the minimal, satisfying cut-set described in §5.2.

Algorithm 1 Minimal Satisfying Cut-set Enumeration for a Monotone Predicate

Input: A fixed-order indexable, label set $L = \{l_1, l_2, \ldots, l_n\}$ and a monotone predicate $\mathcal{P}(\cdot)$ over sets of labels ($\forall x, y : x \subseteq y \wedge \mathcal{P}(x) \rightarrow \mathcal{P}(y)$)
Output: All minimal cut-sets $s \subseteq L$ such that $\mathcal{P}(s) = \text{true}$

Global: List⟨Set⟩ `Result`
Global: Trie⟨L⟩ `SeenCandidate`

Main EnumerateMinimalSatCutSets($L, \mathcal{P}(\cdot)$):
`Result` ← `Null`
`SeenCandidate` ← `Empty`
if $\neg\mathcal{P}(L)$ **then return** `Result`
for `DepthLimit` $= 1$ **to** $|L|$ **do**
  Dfs($\emptyset$, 1, `DepthLimit`, $\mathcal{P}(\cdot)$, L)
return `Result`

procedure Dfs($\mathcal{S}$, `nxtLblIdx`, `DepthLimit`, $\mathcal{P}(\cdot)$, L):
if $\mathcal{S} \in$ `SeenCandidate` or $|\mathcal{S}| >$ `DepthLimit` **then**
  return //Seen candidate or Hit the depth limit
if $\mathcal{P}(\mathcal{S})$ **then**
  `s` ← ShrinkToMinimal($\mathcal{S}, \mathcal{P}(\cdot)$) //Remove redundancy
  if No subset of `s` appears in `SeenCandidate` **then**
    `SeenCandidate`.insert(`s`)
    `Result`.append(`s`)
  return //Found minimal set
for $i =$ `nxtLblIdx` **to** $|L|$ **do**
  if $|\mathcal{S}| + 1 >$ `DepthLimit` **then break**
  Dfs($\mathcal{S} \cup \{l_i\}, i + 1,$ `DepthLimit`, $\mathcal{P}(\cdot), L$)

procedure ShrinkToMinimal($\mathcal{S}, \mathcal{P}(\cdot)$)
  `M` ← $\mathcal{S}$
  for all $x \in \mathcal{S}$ **do**
    `M`′ ← `M` $\setminus \{x\}$
    if $\mathcal{P}$(`M`′) **then**
      `M` ← `M`′
  return `M`

Appendix C.
Meta-Review

C.1. Summary of the Paper

This paper presents a security analysis of EMV contactless mobile payment applications, examining the consequences of an attacker gaining access to sensitive data stored on a smartphone. The relevance of this threat model is assessed through an analysis of 136 applications. A Tamarin-based formal study is then conducted to identify the minimal set of data that must be protected to prevent attacks. Finally, the authors validate their findings by demonstrating a controlled attack on a mobile payment application in an isolated environment.

C.2. Scientific Contributions

3. Create a New Tool to Enable Future Science.
5. Identifies an Impactful Vulnerability.

C.3. Reasons for Acceptance

1. The paper analyzes a well-known protocol under a new, relevant threat model that captures strong compromise scenarios, identifying previously unreported vulnerabilities.
2. It provides EMVResilienceChecker, a practical tool to verify whether an application enforces the security measures prescribed by EMV to prevent sensitive data leakage.

C.4. Noteworthy Concerns

1. The paper provides limited details about the Tamarin model. In particular, it is unclear what modifications were made to the reused model for their analysis.
2. The Tamarin models and the EMVResilienceChecker tool were not available to reviewers at submission time and therefore could not be evaluated.

Appendix D.
Response to the Meta-Review

We acknowledge that our original submission did not include details about the modifications we made to Basin *et al.*'s original Tamarin model. We addressed this issue in our camera-ready version. We also acknowledge that the Tamarin models and EMVResilienceChecker were not included in the original submission. However, it is available in this GitHub repository [19] after acceptance.