

BLECrypto: Automatic Detection of Cryptographic API Misuses in BLE Communication

Haiqin Zhu^a, Weijing You^{†a}, Wenqing Li^b

^aCollege of Computer and Cyber Security, Fujian Normal University, Fuzhou, China

^bVenustech Group Inc., Beijing, China

Abstract

Bluetooth Low Energy (BLE) is increasingly used in domains such as smart homes, wearable devices, and healthcare systems to support short-range wireless communication. To protect sensitive data transmitted over BLE, BLE application-layer encryption is often adopted as a complement to the built-in link-layer security. The effectiveness of such encryption, however, heavily relies on correct cryptographic implementation. Previous studies suggest that cryptographic misuses are common in practice, yet remain underexplored in BLE scenarios due to analysis challenges. Existing cryptographic misuse analysis tools either lack the ability to assess BLE-specific data flows or fail to associate cryptographic parameters at the instance level.

In this paper, we present BLECrypto, a static analysis framework to detect cryptographic misuses in the application-layer encryption of BLE devices. Instead of analyzing BLE devices directly, which are often proprietary or hard to access, BLECrypto analyzes their Android companion apps as proxies. BLECrypto combines control-flow and semantics-guided data-flow analysis to trace propagation of BLE data and assess the correctness of associated encryption configurations. It employs a two-stage pipeline and a confidence-level mechanism to balance detection coverage and precision. We evaluate BLECrypto on 67,509 real-world Android apps, where only 7,283 (10.8%) apps have applied application-layer encryption for BLE communication. BLECrypto reported 5,332 (73.2%) with at least one cryptographic misuse. These findings suggest that cryptographic misuses are widespread in BLE ecosystems and demonstrate the practical value of BLECrypto for large-scale security assessment.

Keywords: Bluetooth Low Energy; application-layer encryption; cryptographic misuse; static analysis; software security

1. Introduction

Bluetooth Low Energy (BLE) has become a core communication standard for short-range interactions in domains such as smart homes, wearable devices, and healthcare systems [1]. These BLE-enabled systems frequently transmit sensitive information, including device control commands, user identifiers, or personal health metrics. Although BLE provides built-in link-layer security mechanisms like pairing and bonding, studies shown that these mechanisms are often insufficient or improperly configured in practice [2, 3]. Therefore, developers are advised

[†]Corresponding author: Weijing You (Email: youweijing@fjnu.edu.cn; ORCID: 0000-0001-7060-5820).

to implement additional BLE application-layer encryption to compensate for the limitations of built-in security of BLE [4].

It is often overlooked that, whether the BLE application-layer encryption is effective depends not only on the choice of algorithm but also on the implementation. Previous works show that developers may misuse cryptographic APIs by, for example, hardcoding encryption keys, selecting outdated algorithms, and omitting essential configuration parameters [5]. Although several static analysis tools have been proposed to detect cryptographic misuses [6, 7], most are designed for general-purpose applications rather than considering the communication patterns or data handling logic in BLE contexts. On the other hand, despite increasingly extensive studies [8, 9] towards IoT security in recent years, more efforts focus on vulnerabilities during workflow, e.g. logic flaws in protocol design, and less on quantifying implementation of underlying modules, e.g. the encryption for secure communication, and hence much less work on cryptographic misuse in BLE context. To the best of our knowledge, BLECryptotracer [4] and CryptoREX [10] are two typical related works. However, BLECryptotracer could answer if the BLE communication flow is encrypted, but has no idea about the quality of applied encryption. CryptoREX focuses on cryptographic misuse detection on firmware layer rather than on the application layer of IoT devices. Therefore, a precise cryptographic misuse detector for BLE communication is still missing in literature.

Detecting cryptographic misuse in BLE application-layer encryption faces two major technical challenges: 1) analyzing BLE device firmware directly is impractical in large-scale scenarios, as the code is often proprietary, heavily obfuscated, and highly vendor-specific; 2) even when analyzing from the Android side, accurately evaluating the security of BLE encryption requires associating multiple cryptographic parameters, including algorithm, key, mode, and IV (Initialization Vector), with the same encryption instance. However, most existing tools analyze cryptographic parameters in isolation and cannot track their coordinated use in BLE-related data flows.

In this work, we tackle the aforementioned limitations and introduce BLECrypto, a static analysis framework designed to detect cryptographic misuse in BLE application-layer communication. BLECrypto employs a two-stage analysis pipeline: the first stage identifies whether BLE data is encrypted before transmission, and the second stage evaluates the correctness of the associated encryption configuration.

The framework integrates control-flow and semantics-guided data-flow analysis, enabling per-instance validation of critical cryptographic parameters along BLE communication paths. By analyzing Android companion apps as proxies for their BLE device counterparts, BLECrypto supports large-scale, firmware-free evaluation of real-world deployments and adopts a confidence-driven engine to balance detection coverage and accuracy across heterogeneous apps.

We evaluate BLECrypto on a dataset consists of 67,509 real-world Android applications that invoke BLE communication APIs, where only 7,283 (10.8%) apps are identified that apply application-layer encryption. Further, BLECrypto reported 5,332 (73.2%) apps with at least one cryptographic misuse.

To assess the practical impact of such misuses, we conduct a simulated attack based on a proof-of-concept BLE setup. The experiment shows that insecure encryption practices detected by BLECrypto can lead to full plaintext recovery, confirming their real-world exploitability.

Our contributions are as follows:

- We present BLECrypto, a novel static analysis framework that, to the best of our knowledge, is among the first to target cryptographic misuses in BLE application-layer commu-

nication.

- We implement a two-stage, semantics-guided analysis pipeline that traces BLE data flows and performs parameter-level validation of each Cipher instance.
- We conduct a large-scale empirical study on over 67,000 real-world Android apps, suggesting that cryptographic misuse in BLE application-layer may be more common than expected.

2. Background

2.1. BLE Communication Model and Security Motivation

BLE communication follows a client-server model, where the initiating device (typically a smartphone) acts as the client and the peripheral as the server. Data is exchanged through the Generic Attribute Profile (GATT), where *characteristics* serve as the fundamental unit of communication.

On Android, BLE characteristic access is provided through the `BluetoothGattCharacteristic` class. Developers use `setValue()` and `getValue()` to assign or retrieve local data, and invoke `writeCharacteristic()` and `readCharacteristic()` to send or receive data over BLE. As shown in Figure 1, we trace the values handled by `setValue()` and `getValue()` as entry points to assess whether they are properly encrypted before transmission.

Although BLE includes link-layer protections such as pairing, bonding, and shared-key encryption, studies have shown these are often misconfigured or bypassed [2, 3, 11]. For instance, weak reconnection logic may be exploited, or malicious co-resident apps may eavesdrop or tamper with BLE traffic despite link encryption [4, 12]. Therefore, developers are often encouraged to apply BLE application-layer encryption as an additional safeguard. However, the security of cryptographic systems largely depends on correct implementation.

Our study targets detecting cryptographic misuses at the BLE application layer. Since analyzing device firmware directly is often infeasible due to proprietary, obfuscated, or inaccessible code, we analyze Android companion apps as proxies. Given that BLE protocols exhibit a high degree of symmetry in message structure and semantics across the central and peripheral roles, we adopt a heuristic assumption: the encryption behavior implemented in Android companion apps often closely mirrors that of their BLE device counterparts. Prior studies [4, 13, 14] have leveraged similar assumptions to infer device behavior through app analysis at scale, including the reflection of BLE security configurations (e.g., access permissions and authentication modes) in companion apps.

Leveraging Android apps as proxies enables static analysis of BLE encryption logic without firmware access, providing a practical solution for large-scale assessment of real-world cryptographic security in BLE ecosystems.

In this work, we assume a passive adversary within BLE signal range who can intercept traffic between mobile apps and BLE peripherals. While the attacker cannot compromise firmware or the Android runtime, the attacker is assumed to have bypassed BLE link-layer protections and thus focuses on exploiting weaknesses in application-layer encryption, such as constant keys or insecure modes, to infer message semantics or plaintext through traffic analysis. The defined threat model motivates our focus on identifying cryptographic misuses that could undermine BLE communication confidentiality despite the presence of link-layer protections.

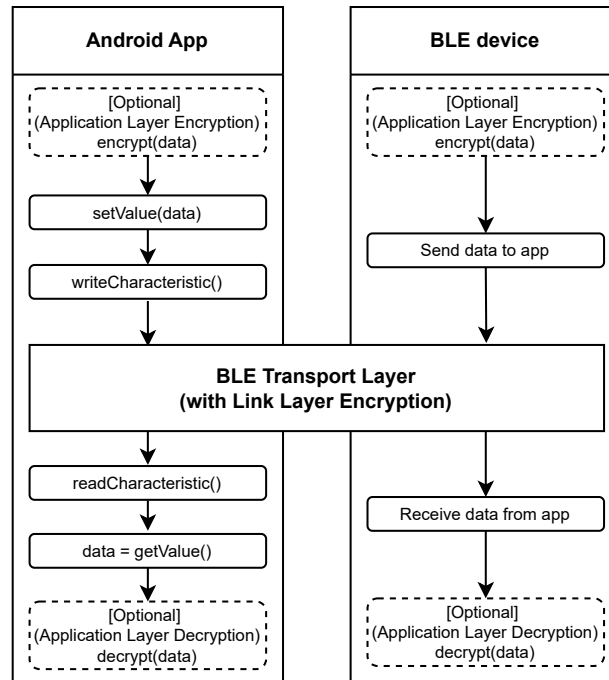


Figure 1. Overview of BLE communication with application-layer encryption

2.2. Cryptographic Misuse in Android Platform

The correct use of cryptographic techniques is fundamental to ensuring the confidentiality, integrity and non-repudiation of information. However, studies [5, 7, 15] have shown that cryptographic misuse is widespread in real-world software development. Such misuses, ranging from insecure algorithms to improper key management, not only weaken encryption but may also introduce new vulnerabilities.

```

1  /* iv and key should be random */
2  public byte[] encrypt(byte[] iv, byte[] key, byte[] data)
3  {
4      IvParameterSpec iv_spec = new IvParameterSpec(iv);
5      SecretKeySpec key_spec = new SecretKeySpec(key, "AES");
6      Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
7      //Cipher insecure=Cipher.getInstance("AES"); //may default to ECB!
8      cipher.init(Cipher.ENCRYPT_MODE, key_spec, iv_spec);
9      return cipher.doFinal(data);
10 }

```

Listing 1. Simplified symmetric key encryption in Java

This study focuses on analyzing cryptographic usage in Android applications as a means of inferring the security practices of BLE devices. While Android apps may be developed in Java, Kotlin, or other languages, their logic is ultimately compiled into Dalvik bytecode and can be disassembled into a language-agnostic Smali representation. At the Smali intermediate representation level, whether encryption is invoked directly through system APIs or indirectly via external libraries, the underlying implementation typically resolves to calls to the unified

cryptographic framework of the platform, which is known as the Java Cryptography Architecture (JCA). The stable invocation patterns of JCA in Smali enable reliable static tracing, allowing us to unify the detection scope.

Among the various cryptographic APIs exposed by JCA, the `Cipher` class is a central focus of our analysis due to its configurability and susceptibility to misuse. As a core cryptographic component supporting both symmetric and asymmetric encryption, `Cipher` instances are typically created using the `getInstance()` factory method, which accepts a `transformation` string to specify the algorithm, mode of operation, and padding scheme.

Listing 1 shows a typical usage example where the cipher instance is explicitly configured with the AES algorithm, CBC mode, and PKCS5 padding. Notably, only the algorithm component is mandatory in the transformation string. When the mode is omitted, the cipher defaults to ECB (details in Section 4, R2). Developers are therefore expected to specify complete transformation strings to avoid such insecure defaults.

3. Related Work

3.1. IoT Security and BLE Analysis

Static analysis of mobile companion apps has emerged as a powerful technique for assessing IoT security at scale, particularly when direct access to devices is limited. Zhou et al. [16] systematically examined security flaws in smart home ecosystems by modeling the interactions among apps, devices, and cloud services, revealing logic flaws across major platforms such as SmartThings and MIJIA. Schmidt et al. [13] proposed IoTFlow to infer IoT device behaviors and protocol usage at scale without requiring device access. Nan et al. [17] developed IoTProfiler to detect large-scale exposure of sensitive IoT data, such as health metrics and device identifiers.

Complementary to these general IoT efforts, prior work has specifically focused on BLE protocols, identifiers, and communication patterns. Zuo et al. [8] introduced BleScope to analyze static UUID usage in BLE apps and correlate them with real-world device behaviors for fingerprinting. Zhao et al. [9] proposed PeriScope to assess privacy and access control vulnerabilities in apps acting as BLE peripherals. Sivakumaran et al. [18] developed Ble-Guide to evaluate BLE-specific security configurations, such as firmware updates and adopted UUID usage, across more than 17,000 APKs. Du et al. [14] analyzed companion mobile apps to infer BLE security misconfigurations in IoT devices, demonstrating that app code often mirrors device-side BLE configurations and can be leveraged for large-scale ecosystem analysis.

Prior efforts in the IoT and BLE domains have primarily focused on inferring device behavior, ecosystem logic flaws, or authentication and privacy risks. In contrast, our work targets a complementary and often overlooked dimension: the correctness of application-layer cryptographic protection in BLE data channels. Specifically, we analyze whether BLE communication data is encrypted, whether the encryption is applied securely, and whether common misuse patterns exist in cryptographic API configurations.

3.2. Crypto Misuse Detection

General-Purpose Language and Platform Detection. A large body of research has focused on building static or dynamic analysis tools to identify cryptographic misuses across specific programming languages and platforms. CryptoLint by Egele et al. [7] employed static slicing to examine over 10,000 Android applications and found that 88% contained at least one cryptographic misuse. LICMA by Wickert et al. [19] leveraged abstract syntax trees and slicing to

support multi-language detection across Python and Java. CryptoGo by Li et al. [5] applied static taint analysis to Go projects. Additional tools include TAINTCRYPT [20] (for C/C++), CRY-LOGGER [21] (for Android), and CryptoShield [15] (for non-root Android apps), demonstrating the feasibility of cryptographic misuse detection across diverse language ecosystems.

Detection Effectiveness and Optimization. To reduce false positives and improve precision, several studies have introduced advanced analysis strategies. These include flow/context/field-sensitive slicing techniques as seen in CryptoGuard [6], mutation testing combined with deep learning models like Bi-LSTM [22], neural networks guided by static program slicing [23], and rule synthesis via unified cryptographic description languages as used in the Gopher framework by Zhang et al. [24].

Beyond Detection: Functional Enhancements. Some tools extend beyond simple misuse detection by offering automated repair or traceability support. For example, CDRep [25] is the first tool to provide automated fixes for cryptographic misuses. BinSight [26] localizes the root causes of crypto misuses, while CogniCrypt [27] integrates with Eclipse IDE to provide code suggestions and real-time detection as part of the development workflow.

Domain-Specific Detection. More recent efforts explore domain-specific or ecosystem-specific cryptographic analysis. NativeSpeaker [28] targets cryptographic misuse in native binary libraries. CryptoREX [10] combines static analysis with firmware unpacking to detect cryptographic misconfigurations at scale across IoT firmware, highlighting the prevalence of insecure cryptographic practices in embedded devices.

While existing tools have advanced cryptographic misuse detection, they fall short in BLE-specific scenarios. For instance, Android-focused tools like CryptoLint [7] and CryptoGuard [6] detect misuses (e.g., hardcoded keys, insecure modes) in isolation, without associating parameters within the same encryption operation, which limits their effectiveness for BLE contexts where encryption quality must be inferred along data paths tied to BLE APIs. Meanwhile, tools like CryptoREX [10] offer broader device-level coverage but lack visibility into application-layer logic and BLE communication flows. BLECrypto bridges the gap by statically analyzing BLE data flows in Android apps and performing per-instance validation of encryption configurations.

4. Cryptographic Rules for BLE Communication

To detect insecure cryptographic usage in BLE application-layer communication, we adopt four detection rules (R1–R4) that cover common classes of symmetric encryption misuse. These rules are inspired by the misuse patterns defined in CryptoGuard [6], but are specifically selected based on their applicability in the BLE context. The selection follows two criteria: (1) each rule targets misuse patterns that are statically analyzable in Android companion apps without requiring runtime information, and (2) each rule reflects a cryptographic practice that is realistically adopted in BLE device implementations, given the symmetric nature of BLE and cryptographic protocols.

Compared to the broader Android ecosystem, where various misuse types (e.g., password-based encryption, SSL/TLS misconfiguration, certificate validation) have been studied, our scope is deliberately restricted. These other patterns are excluded because they either (i) lack behavioral symmetry between BLE endpoints, (ii) rely on user input or device-specific context unavailable to static analysis, or (iii) pertain primarily to Internet-facing communication rather than BLE-based data exchange. By focusing on the BLE application-layer setting, we ensure both practical relevance and detection feasibility. BLECrypto is designed to be modular, and its rule set can

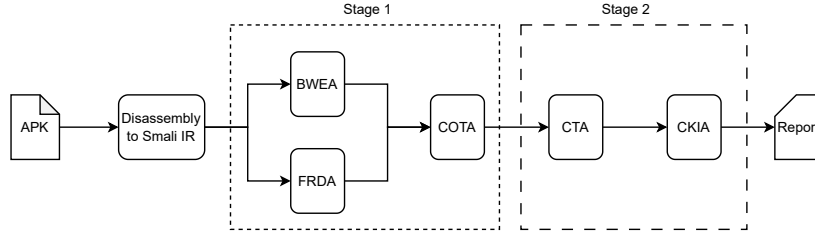


Figure 2. System overview. The analysis consists of two stages: (1) encryption identification using *BWEA* (*Backward-WriteEncryptionAnalyzer*), *FRDA* (*ForwardReadDecryptionAnalyzer*), and *COTA* (*CipherOperationTypeAnalyzer*); and (2) misuse detection using *CTA* (*CipherTransformationAnalyzer*) and *CKIA* (*CipherKey/IVAnalyzer*)

be extended to cover additional patterns (e.g., insecure or improper padding modes, asymmetric key size constraints) that are semantically expressible and amenable to static analysis.

The selected rules are as follows:

- *R1 (Insecure Algorithm): Do not use insecure encryption algorithms.* Legacy algorithms like DES or 3DES are still supported by some cryptographic libraries but are widely known to be insecure. In BLE contexts, such weak algorithms can be brute-forced by adversaries with minimal computation, enabling plaintext recovery from intercepted traffic, especially since BLE packets are short and predictable in structure, e.g. status flags.
- *R2 (Encryption Mode): Do not use ECB mode for encryption.* ECB mode leaks plaintext patterns when identical input blocks yield identical ciphertexts. In BLE scenarios, such deterministic leakage enables fingerprinting of commands, e.g. ON/OFF toggles, or repeated status reports. Because BLE data is often semantically repetitive (e.g., polling sensors), ECB usage creates measurable leakage that adversaries can exploit through traffic observation.
- *R3 (Constant Keys): Do not use constant keys.* If an application embeds a constant encryption key directly in its codebase, an attacker with access to the APK may extract it through static analysis. Assuming the same key is deployed across all devices, a one-time extraction would enable the attacker to decrypt messages from any device using that app, or impersonate the app during communication.
- *R4 (Constant IVs): Do not use constant IVs in CBC mode.* The initialization vector (IV) in CBC mode introduces randomness into each encryption session. Using a fixed IV, however, undermines the randomness and causes identical plaintext blocks to yield identical ciphertexts across sessions, exhibiting ECB-like leakage behavior. For example, repeated plaintext inputs (e.g., periodic BLE polling responses) can result in ciphertext patterns that attackers may exploit to identify repeated operations or infer message semantics, assuming passive interception and access to known plaintext samples.

5. BLECrypto Design

5.1. Overview

To systematically address cryptographic misuses in BLE application-layer encryption, we design and implement **BLECrypto**, a static analysis framework that detects cryptographic misuses

in BLE on Android companion apps.

BLECrypto is built upon Androguard [29] and BLECryptracer [4]. While BLECryptracer focuses solely on detecting whether BLE communication data is encrypted, it does not examine the correctness or security of the applied cryptographic configurations. BLECrypto, by contrast, is designed with a broader analysis goal and performs systematic enhancements in both architectural structure and semantic coverage. Specifically, BLECrypto introduces the following key enhancements:

- *Two-Stage Modular Analysis Pipeline*: BLECrypto separates encryption identification and misuse detection into two decoupled stages handled by dedicated semantic analyzers. The modular design improves analytical clarity and enables focused reasoning for each analysis task;
- *Control-Flow-Based Basic-Block-Level Analysis*: BLECrypto leverages control-flow graphs (CFGs) to perform instruction-level tracking within reachable basic blocks, improving path precision and increasing resilience to complex branching logic;
- *Semantics-guided Cryptographic Misuse Detection*: BLECrypto incorporates a dedicated analyzer set to examine algorithm choices, cipher modes, key and IV configurations based on Java Cryptography Architecture (JCA) semantics, enabling the detection of four BLE-specific misuse patterns (R1–R4);
- *Optimized Confidence-Driven Task Scheduling*: To improve result interpretability and reduce noise from uncertain static traces, BLECrypto adopts a simplified confidence scheme that focuses on high- and medium-confidence paths, aligning the framework more closely with the demands of cryptographic misuse detection in BLE applications.

The overall architecture of BLECrypto is illustrated in Figure 2. The system takes Android APKs as input, disassembles them into Smali-based intermediate representations, constructs program structures including control-flow graphs (CFGs) [30] and call graphs, and then performs a two-stage analysis guided by semantic analyzers and task confidence. The first stage identifies whether BLE communication data is encrypted or decrypted; the second stage detects potential misuses in the associated cryptographic configurations. Confidence levels influence the prioritization of analysis paths throughout both stages, balancing precision and coverage. Finally, BLECrypto outputs a structured report including the locations of problematic encryption paths and associated misuses.

5.2. Disassembly and Intermediate Representation

To enable static analysis over Android applications, BLECrypto first disassembles each input APK using Androguard into Smali code, which is a register-based, low-level representation of Dalvik bytecode that preserves instruction semantics and is resilient to obfuscation. Based on the Smali output, BLECrypto constructs intermediate program structures, including control-flow graphs (CFGs) that capture intra-procedural basic-block relationships, and call graphs that model inter-procedural method invocations.

These representations serve as the foundation for forward and backward data-flow analysis [31] in subsequent stages.

5.3. Two-Stage Analysis Pipeline

BLECrypto performs static analysis in two sequential stages: the first identifies whether BLE communication data is processed by cryptographic operations (e.g., encryption or decryption); the second traces the associated cryptographic configurations, such as algorithm, mode, key, and IV, from these operations to detect potential misuses. Each stage leverages a dedicated set of semantic analyzers, operating on Smali intermediate representations, guided by control-flow graphs and using both forward and backward data-flow analysis.

Stage 1: Identify cryptographic use in BLE Communication.. Stage 1 focuses on determining whether data exchanged through BLE APIs is encrypted or decrypted within the application. BLECrypto targets Android's BluetoothGattCharacteristic methods such as `setValue()` for data writing and `getValue()` (and its variants) for data reading.

To support cryptographic use identification, BLECrypto incorporates two dedicated semantic analyzers:

BackwardWriteEncryptionAnalyzer (BWEA) BWEA performs backward data-flow analysis starting from the argument of `setValue()`, tracing the data to determine whether the value comes from the return of a `Cipher.doFinal()` call, which would indicate encryption. Upon identifying a valid path, the corresponding `Cipher` instance is recorded for further analysis.

ForwardReadDecryptionAnalyzer (FRDA) FRDA performs forward data-flow analysis starting from the return value of BLE read operations (e.g., `getValue()`, `getIntValue()`, or `getStringValue()`), and tracks whether the received data is subsequently passed to `Cipher.doFinal()` as an indication of decryption.

CipherOperationTypeAnalyzer (COTA) Both BWEA and FRDA rely on COTA to verify the operational mode of each detected `Cipher` instance. COTA traces backward from `doFinal()` to `init()` to determine whether the `opmode` parameter indicates encryption or decryption. Only validly configured instances proceed to the next stage.

BLECrypto analyzes each BLE communication method individually. When encryption is detected for a given method, analysis for that method terminates, while examination continues for other methods to ensure complete coverage of all communication paths.

Stage 2: Cryptographic Misuse Detection.. Stage 2 inspects the configuration of each `Cipher` instance identified in Stage 1 as participating in BLE data and checks for potential cryptographic misuses.

CipherTransformationAnalyzer (CTA) CTA checks *R1* and *R2* by inspecting the transformation string used in `getInstance()`, which specifies the encryption algorithm, mode, and padding (e.g., AES/ECB/PKCS5Padding). Building on the `init()` call identified by COTA, CTA performs backward tracing to locate the corresponding `getInstance()` invocation and extract the transformation.

CipherKeyIVAnalyzer (CKIA) With respect to *R3* and *R4*, CKIA examines whether the cryptographic key and initialization vector (IV) used in a `Cipher` configuration are derived from static constants. Starting from the arguments passed to `init()`, CKIA performs backward analysis to locate the associated `SecretKeySpec` (for keys) and `IvParameterSpec` (for IVs) calls, and traces their constructor parameters to determine whether any constant value is used. CKIA is selectively triggered based on the algorithm and mode extracted by CTA. Specifically, key analysis is applied only for symmetric encryption algorithms, and IV analysis is conducted only when the encryption mode is CBC.

The two-stage analysis enables BLECrypto to not only determine whether BLE data is encrypted, but also assess the security of the applied cryptographic configuration.

5.4. Confidence-Driven Scheduling Strategy

To balance detection precision and coverage, BLECrypto employs a confidence-driven task scheduling strategy. Each data-flow trace is labeled as either *high-confidence* or *medium-confidence*, depending on how explicitly the data dependency between BLE communication and cryptographic operations can be established during static analysis. The assignment of confidence levels follows these definitions:

- **High Confidence:** A trace is classified as high confidence when a direct data dependency is established with minimal ambiguity. This includes (i) intra-procedural propagation through registers within the same method, (ii) inter-procedural propagation via explicit method calls with statically resolvable caller-callee relationships, and (iii) field-based data flow through instance or static fields when both the write and read sites are directly identifiable.
- **Medium Confidence:** A trace is classified as medium confidence when the dependency involves indirect or potentially ambiguous flows. These include (i) interface or abstract method invocations, where multiple implementations may exist, (ii) method dispatch affected by inheritance and overriding, and (iii) argument-level associations within `invoke` instructions where data correlation is implied rather than explicit.

BLECrypto first analyzes high-confidence paths where direct dependencies are available. If no such path is found, data-flow tracking continues over medium-confidence paths, which allow additional inferences with relaxed constraints. Confidence labels are explicitly reported in the output to help users interpret the underlying reasoning behind each detection.

6. Implementation and Evaluation

6.1. Dataset

The dataset used in this study is derived from AndroZoo [32], a large-scale repository of Android applications widely used in academic research. Focusing on samples from Google Play, we first filtered for APKs with Bluetooth permissions based on AndroZoo metadata [33], resulting in approximately 3 million initial candidates, then deduplicated by package name to retain only the latest version of each app, and further selected those invoking BLE-related API calls (e.g., `setValue()`, `getValue()`). Finally, we obtained a final dataset of 67,509 apps for static analysis.

6.2. Experiment Setup

BLECrypto introduces three key configuration mechanisms to balance detection coverage, performance overhead, and false positive rates: path depth limitation, analysis timeout configuration, and whitelist mechanism.

Path Depth Limitation. The path depth d denotes the number of interprocedural calls traversed from the initial communication API during static analysis, starting at $d = 0$ and incremented by one at each function boundary. Once d reaches a predefined threshold, further tracking along that path terminates.

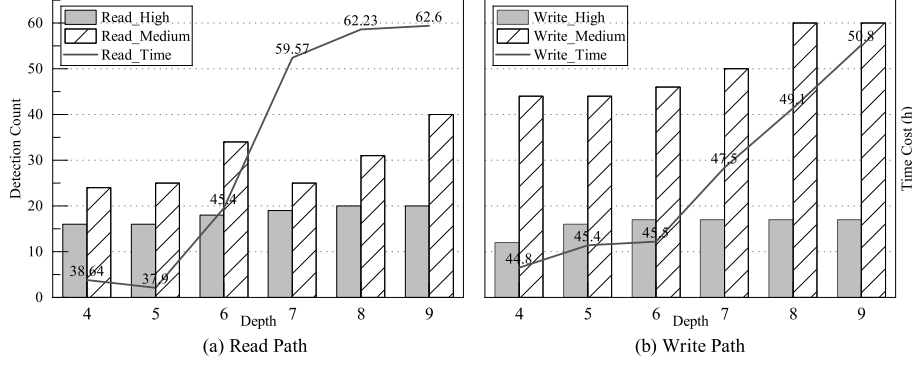


Figure 3. Impact of depth setting on detection capability and time cost. (a) Read-path analysis: increasing depth improves high-confidence detections up to $d = 6$, beyond which the gain is marginal while time cost continues to rise. (b) Write-path analysis: detection counts stabilize after $d = 6$, while analysis time increases significantly beyond $d = 8$. Depth values of $d = 6$ (read) and $d = 8$ (write) are selected as optimal trade-offs.

Table 1. Stage 1: BLE Encryption Detection Summary

Path	#APKs	BLE APIs*	Enc. APIs**	Crypto APIs [#]
Read	67,492	706,466	9,549	3,845
Write	65,743	437,399	8,936	6,485
Total	67,509	1,143,865	18,485	10,330

Note: * BLE APIs: BLE communication API calls (e.g., setValue(), getValue()); ** Enc. APIs: BLE calls preceded by encryption (any confidence); [#] Crypto APIs: Cryptographic API calls used for encrypting BLE communication; The value in “Total” row for #APKs represents the unique number of apps analyzed (i.e., the intersection), while values in other columns are summed across paths.

To determine optimal values, we tested $d = 4$ to $d = 9$ using BWEA and FRDA on a random sample of 1,000 APKs. As shown in Figure 3, $d = 6$ for read paths captures the majority of high-confidence instances; increasing d further yields only minimal additional detections at the cost of a substantial increase in runtime. For write paths, $d = 8$ balances detection capability and performance overhead. Consequently, we adopted $d = 6$ for read paths and $d = 8$ for write paths in our experiments.

Analysis Timeout Configuration. To limit resource consumption caused by long or complex paths, BLECrypto incorporates timeout thresholds during analysis. In Stage 1, we set 600 seconds for write paths and 1,000 seconds for read paths, based on the observed write-to-read BLE API call ratio (6:10). In Stage 2, a 180-second timeout is applied per Cipher instance.

Whitelist Mechanism. Certain code paths invoke common library methods (e.g., kotlin.*, okhttp3.*, okio.*), which are often highly abstracted and lack meaningful semantics for cryptographic analysis. Deep inspection of such methods tends to cause path explosion and false positives. Therefore, BLECrypto maintains a built-in method whitelist to skip in-depth tracking of specified library methods to improve overall stability and efficiency of analysis.

Table 2. Stage 2: Misuse Detection by Confidence Scope (APK-Level)

Path	Enc. (All)*	Mis. (All)**	Enc. (High) [#]	Mis. (High) ^{##}
Read	3,755	3,302 (87.9%)	901	520 (57.7%)
Write	5,299	3,650 (68.9%)	812	661 (81.4%)
Total	7,283	5,332 (73.2%)	1,442	946 (65.6%)

Note: * Enc. (All): Apps with any-confidence BLE encryption; ** Mis. (All): Misuse cases among Enc. (All); [#] Enc. (High): Apps with high-confidence encryption only; ^{##} Mis. (High): Misuse cases in high-confidence apps.

6.3. Results

6.3.1. Overall

We analyzed 67,509 Android apps with BLE APIs to assess the presence and correctness of application-layer encryption using BLECrypto. Table 1 summarizes the detection results of BLE-related encryption behavior across read and write communication paths, while Table 2 further reports the prevalence of cryptographic misuses among the apps identified to apply such encryption.

BLECrypto identified 7,283 APKs performed application-layer encryption over BLE communication data, where 5,332 applications (73.2%) were identified that violate at least one of the defined cryptographic misuse rules.

To avoid overestimating the misuse rate, we further excluded any APK that contained medium-confidence findings in either Stage 1 or Stage 2. Since each APK may include multiple BLE APIs analyzed independently, even a single such result led to exclusion. Under the stricter setting, BLECrypto still reported cryptographic misuse in 946 out of 1,442 cases (65.6%). Our findings suggest that the observed security risks are not artifacts of over-approximation or loose assumptions, but reflect pervasive issues in real-world app implementations.

Overall, while a subset of developers have adopted BLE application-layer encryption, a majority of them failed to configure it securely. The prevalence of misuse, even under strict analysis constraints, highlights the necessity of specialized detection tools that go beyond simple API usage checks.

Subsequent sections present results for Stage 1, Stage 2, manual validation and baseline comparison (including case studies), and a time efficiency evaluation.

6.3.2. Identify cryptographic use in BLE Communication

We evaluated the Stage 1 analysis of BLECrypto in two settings: benchmark testing using BLE-Benchmark [4], and large-scale deployment across 67,509 real-world APKs.

Evaluation on BLE-Benchmark. A comparison between BLECrypto and BLECryptracer was conducted on the BLE-Benchmark dataset. The results are shown in Table 3. At high confidence, BLECrypto achieved better performance on both read (Precision: 93%, Recall: 73%, F1: 82%) and write operations (Precision: 92%, Recall: 59%, F1: 72%). At medium confidence, BLECrypto showed improved precision but slightly reduced recall, with F1 scores remaining stable. The results validate the accuracy of the encryption identification of BLECrypto and establish a solid basis for the subsequent misuse detection phase.

Large-scale application results. Table 1 summarizes Stage 1 detection results at the API level. For BLE read operations, BLECrypto analyzed 706,466 API calls across 67,492 APKs, identifying 9,549 calls (3,770 high-confidence) as encrypted, from 3,755 apps. For write operations, 437,399 API calls were analyzed across 65,743 APKs, of which 8,936 (1,537 high-confidence) were flagged as encrypted, from 5,299 apps.

BLECrypto: Automatic Detection of Cryptographic API Misuses in BLE Communication

Table 3. Comparison between BLECryptotracer and BLECrypto under different confidence levels on read and write paths.

Metric	Read				Write			
	BLECryptotracer		BLECrypto		BLECryptotracer		BLECrypto	
	High	Med	High	Med	High	Med	High	Med
Dataset Size	92	30	92	32	90	42	90	42
Detected	62	11	60	7	48	22	48	18
TP	58	7	56	6	44	19	44	17
FP	4	4	4	1	4	3	4	1
TN	11	7	11	10	11	8	11	10
FN	19	12	21	15	31	12	31	14
Precision	94%	64%	93%	86%	92%	86%	92%	94%
Recall	75%	37%	73%	29%	59%	61%	59%	55%
F1 Score	83%	47%	82%	43%	72%	72%	72%	70%

Note: TP, true positives; FP, false positives; TN, true negatives; FN, false negatives.

Table 4. Cryptographic misuse statistics per rule, grouped by API and APK count.

Rule	Read API	Read APK	Write API	Write APK
R1	17	17	87	85
R2	2056	2025	1531	1425
R3	2753	2719	2861	2744
R4	477	465	664	662

Note: Read API / Write API: Total number of corresponding BLE API calls across all APKs for read / write operations; Read APK / Write APK: Number of APKs that invoke the corresponding BLE APIs for read / write operations.

These results demonstrate the ability of BLECrypto to identify BLE encryption behavior in realistic mobile applications. While medium-confidence traces offer broader coverage, the high-confidence subset alone provides a solid foundation for the stricter analysis in Stage 2.

6.3.3. Cryptographic Misuse Detection

In Stage 2, we assessed whether the encryption configurations identified in Stage 1 complied with the BLE-specific cryptographic rules defined in Section 4. The following statistics, summarized in Table 2 and Table 4, reflect misuse detection results observed by BLECrypto in our large-scale evaluation:

- *Read operations.* Among 3,755 APKs performing decryption of BLE data, 3,302 (87.9%) exhibited at least one rule violation. The most common were R3 (constant keys) and R2 (ECB mode), indicating frequent flaws in key management and encryption mode selection.
- *Write operations.* Out of 5,299 APKs applying encryption before BLE transmission, 3,650 (68.9%) violated at least one rule. The violation patterns were similar to the read path, again dominated by R3 and R2.

Table 5. Occurrences and security notes of detected cryptographic algorithms.

Algorithm	Occurrences	Security Note
DES	82	Considered insecure; deprecated by NIST due to short key length (56-bit) [34].
DESede	21	Not recommended; still acceptable for compatibility but being phased out by NIST [35].
Blowfish	1	Not recommended; considered insecure due to its 64-bit block size and vulnerability to birthday attacks [5]

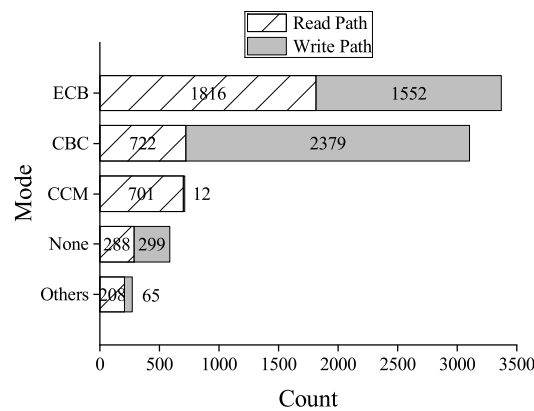


Figure 4. Distribution of encryption modes in BLE application-layer communication across read and write paths

In total, 5,332 of 7,283 apps with BLE encryption (73.2%) failed to comply with at least one rule. To mitigate possible overestimation from indirect traces, we further evaluated a strict setting by excluding any APK containing medium-confidence results. Even under this conservative constraint, 946 of 1,442 high-confidence apps (65.6%) were found to exhibit misuse. These findings provide strong evidence for the widespread and systematic nature of misuses in BLE application-layer encryption. In the following sections, we present detailed findings for R1–R4.

R1: Do not use insecure encryption algorithms. Table 5 summarizes several widely recognized weak encryption algorithms observed in BLE communication paths. Although weak algorithms constitute only a small portion of all encryption usage cases, we still identified instances where deprecated algorithms such as DES, DESede (3DES), and Blowfish were applied in real-world applications.

The risks associated with these algorithms have been extensively documented: DES suffers from insufficient key length and is vulnerable to brute-force attacks; Blowfish is susceptible to key collisions; and DESede, despite being a strengthened version of DES, has been progressively phased out due to its relatively low security strength. Even though security standards, e.g. NIST SP 800-131A [34], explicitly discourage such use, the detection results in BLE encryption implementations highlight a clear gap between secure development practices and actual developer behavior.

R2: Do not use ECB mode for encryption. Figure 4 presents the distribution of encryption

Table 6. Distribution of Rule 3 and Rule 4 violations across different paths and confidence levels.

Path	Confidence	Rule 3	Rule 4
Read Path	High	487	249
	Medium	2266	228
Write Path	High	1290	289
	Medium	1571	375

modes identified across BLE communication paths. R2 was the second most frequently violated among all cryptographic rules and notably exhibited clear directional asymmetry between read and write paths.

Specifically, misuse of ECB mode was significantly more prevalent in the read path, where its frequency far exceeded that of more secure alternatives such as CBC and CCM. We also observed a subset of cases where the encryption mode was not explicitly specified, which likely causes the system to default to ECB mode and thus exacerbates potential risks.

The finding indicates that developers may lack proper awareness of encryption configuration when handling BLE read operations and may instead rely on default, and often insecure, settings. In contrast, write paths showed a much higher adoption rate of CBC mode, yet ECB usage remained non-negligible.

It is worth noting that the total number of configurations labeled with “ECB” or “None” is higher than the number of actual R2 violations. The reason is that some encryption configurations, such as “RSA/ECB/PKCS1Padding”, contain “ECB” or “None” as a syntactic placeholder. Since these modes have no semantic effect in asymmetric encryption, such cases were excluded from our violation counts.

Although more secure modes like CTR, CCM, and GCM are theoretically preferable, they were only sparsely observed in our dataset, reflecting limited developer understanding or experience in configuring encryption schemes.

R3: Do not use constant keys. Table 6 presents the statistics on the usage of constant cryptographic keys. Among all rules, R3 was the most frequently violated, indicating that key management remains the primary weak point in current application-layer encryption practices.

From the perspective of communication paths, BLECrypto demonstrates relatively stronger detection capability at the high-confidence level in write paths, effectively capturing the majority of misuse cases. In contrast, detections in read paths are predominantly concentrated in the medium-confidence category. The observed gap between read and write paths may reflect practical challenges, such as more complex data dependencies and less explicit key propagation in read-side logic, which increase the difficulty of identification by BLECrypto.

In symmetric encryption, the security of the key is highly dependent on its randomness and confidentiality. If a developer hardcodes a constant key within the application, attackers can easily extract it via static analysis or reverse engineering, thereby circumventing the encryption scheme and exposing sensitive communication data. Our analysis revealed that such key misuse occurs frequently across numerous samples, underscoring the urgent need for improved key management practices in BLE-enabled mobile applications.

R4: Do not use constant IVs in CBC mode. BLECrypto detected 1141 instances of constant IV usage, including 538 in high-confidence paths and 603 in medium-confidence paths. In CBC mode, the IV introduces randomness into the encryption process, preventing identical plaintexts from yielding the same ciphertext. The use of a constant IV eliminates the randomness and can

Table 7. Rule-wise Validation Results of BLECrypto with Confidence-Level Breakdown

Rule	Confidence	TP	FP	Precision
Rule 1: Insecure Algorithm	High	1	0	100%
Rule 2: ECB Mode	High	31	0	100%
Rule 3: Constant Key	High	22	1	95.7%
Rule 3: Constant Key	Medium	8	3	72.7%
Rule 4: Constant IV	High	3	0	100%
Rule 4: Constant IV	Medium	2	1	66.7%
Total	–	67	5	93.1%

TP: True Positives. FP: False Positives. Precision is calculated as $TP / (TP + FP)$.

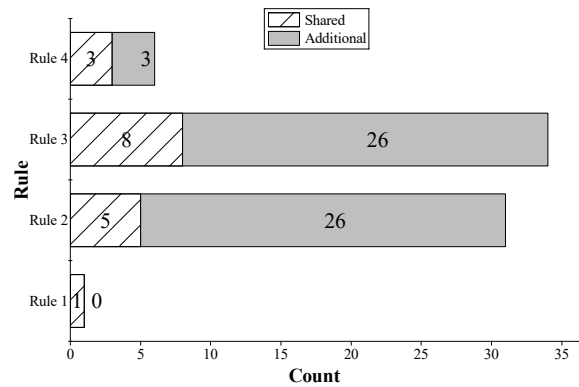


Figure 5. Rule-wise detection coverage comparison between BLECrypto and CogniCrypt_SAST. Each bar shows the number of violations detected by BLECrypto per rule, separated into cases also detected by CogniCrypt_SAST (shared) and additional cases. BLECrypto detects all baseline violations and introduces additional findings across all four rules.

expose ciphertext patterns akin to those seen in ECB mode, thereby compromising confidentiality.

While not as prevalent as violations of R2 or R3, constant IV usage remains a non-negligible threat and reflects a broader lack of awareness about proper IV usage in secure encryption practices.

6.3.4. Manual Validation and Baseline Comparison

To further evaluate the effectiveness of BLECrypto, we conducted a manual analysis on a set of real-world samples. Using a stratified sampling strategy, we randomly selected 30 APKs from all detection results. Specifically, we grouped samples based on the package name prefix of the method containing the cryptographic API, ensuring that at most one APK was selected from each group to enhance diversity and representativeness.

We selected CogniCrypt_SAST for comparison since it can also associate encryption parameters (e.g., key, IV) with the corresponding Cipher instance. Both tools were executed on the same set of 30 APKs, with results manually verified.

Manual verification of 42 BLE-related encryption instances identified in Stage 1 by BLECrypto confirmed 39 true positives (TPs) and 3 false positives (FPs). Based on rule-level de-

tection in Stage 2, a total of 72 violations were reported across these instances, resulting in a precision of 93.1%. The observed false positives were mainly due to path sensitivity limitations and complex data dependencies, which are common challenges in static analysis [36]. Table 7 summarizes the rule-wise results by confidence level.

CogniCrypt_SAST successfully analyzed 22 out of the 30 APKs, with the remainder failing due to issues such as stack overflows. To ensure a fair, context-aligned comparison, only the intersection of encryption instances reported by both tools was considered, yielding 8 shared Cipher instances covering 16 rule violations. All were confirmed as true positives, resulting in a precision of 100% for CogniCrypt_SAST. Notably, BLECrypto also correctly identified all 8 of these instances.

Figure 5 presents a rule-wise breakdown of BLECrypto’s detection results. For each rule, the figure distinguishes violations that are also detected by CogniCrypt_SAST from additional cases uniquely reported by BLECrypto. In total, BLECrypto identifies 55 additional violations, including BLE-specific misuse patterns and medium-confidence paths, with five false positives observed under manual validation.

Finally, we selected three representative applications from the sample set for further case study analysis, to showcase the practical effectiveness of BLECrypto in real-world scenarios.

Case 1: A Smart Access Control Application. This application is involved in controlling physical access devices, and likely transmits authentication credentials or control commands. BLECrypto identified several cryptographic misuses in this app:

- *R1: Encryption algorithm.* The encryption algorithm used is DESede (Triple DES), which, although marginally more secure than DES, has been explicitly marked as “not recommended” by standards such as NIST SP 800-57 [37], and is no longer suitable for protecting sensitive data.
- *R2: Encryption mode.* The encryption mode is ECB, which lacks necessary randomness and may expose plaintext patterns through ciphertext.
- *R3: Encryption key.* The encryption key is hardcoded in the application, making it easily retrievable through reverse engineering.

If an attacker successfully extracts the encryption parameters, it could result in privilege escalation or unauthorized device manipulation, posing significant security risks.

Case 2: An Industrial Monitoring Application. The app is described as supporting remote monitoring and collaborative control of connected infrastructure, but exhibits two misuses:

- *R3: Encryption key.* A constant string composed of repeated digits is used as the encryption key.
- *R4: Initialization vector.* The same constant is reused as the IV in CBC mode, eliminating the required randomness.

If an attacker extracts the hardcoded parameters, they may be able to decrypt or forge BLE messages, potentially interfering with device operations. Notably, this case was detected via medium-confidence paths, demonstrating the capability of BLECrypto to uncover real-world vulnerabilities beyond high-confidence scenarios.

Case 3: A Health Monitoring Application. We observed an application that pairs with medical health devices and applies cryptographic operations to both incoming and outgoing BLE data. However, the implementation suffers from the following problems:

- *R2: Encryption mode.* Both encryption and decryption are performed using the ECB mode.
- *R3: Encryption key.* The key is hardcoded as a constant within the application.

Through reverse engineering, we confirmed that these cryptographic operations are embedded in the modules responsible for processing physiological measurement data. If an attacker can intercept and analyze the communication, they may be able to recover private health metrics such as heart rate and blood oxygen levels, resulting in serious privacy breaches.

6.3.5. Time Efficiency

BLECrypto achieves efficient large-scale analysis, with an average runtime of 157.9 seconds per APK. Notably, the system incorporates a timeout control mechanism to prevent excessive resource consumption caused by overly complex analysis paths. Even if the analysis of certain samples is interrupted due to timeouts, BLECrypto is able to retain and return the completed partial results, ensuring the stability and usability of the tool in practical scenarios.

In our evaluation, for the write path analysis, BLECrypto encountered timeouts in 5,988 out of 67,509 APKs, accounting for 8.87% of the dataset. Despite these interruptions, 578 APKs (9.65% of the timed-out cases) still yielded valid misuse reports. For the read path, 1,911 APKs (2.83%) timed out, yet 144 of them (7.53% of the timed-out cases) successfully detected misuse. These results further demonstrate effectiveness of BLECrypto in large-scale scenarios, i.e., even in the presence of complex control flows or deep analysis paths, the tool can still provide meaningful partial outputs rather than failing silently.

6.4. Practical Exploitation Demonstration

To validate the exploitability of cryptographic misuses detected by BLECrypto, we designed an experiment based on the passive attacker model introduced in Section 2.1. The attacker is assumed to have bypassed link-layer protections and focuses solely on application-layer traffic.

6.4.1. Experiment Setup

The BLE communication testbed involved three components: (1) a **central device**, i.e., an Android smartphone running a custom companion app to initiate BLE communication; (2) a **peripheral device**, i.e., an Android tablet running a custom app that emulates BLE-enabled IoT firmware, exposing GATT services and applying symmetric encryption; and (3) a **sniffing device**, i.e., a PC with an nRF52840 sniffer and Wireshark for BLE packet capture.

The peripheral application was deployed on a tablet that supports BLE Peripheral mode, verified through the `isMultipleAdvertisementSupported()` API and successful initialization of both `BluetoothLeAdvertiser` and `BluetoothGattServer`. With these capabilities, the tablet functioned as a BLE GATT server, communicating with the central device over BLE in a manner that is functionally equivalent to commercial IoT peripherals at the protocol level. We adopted this controlled simulation setup to ensure full observability of both communication ends while avoiding ethical concerns associated with reverse-engineering or attacking real-world third-party devices.

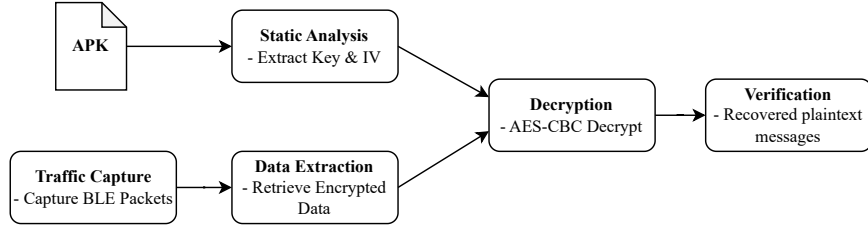


Figure 6. Simulated workflow demonstrating the recovery of BLE plaintext messages by exploiting cryptographic misuses detected by BLECrypto.

6.4.2. Encryption and Communication Design

To reflect typical misuses observed in real-world apps, the demo apps implemented application-layer encryption using the AES/CBC/PKCS5Padding algorithm, with a hardcoded 128-bit key ("MySecretKey12345") and a fixed IV ("MyFixedIV1234567").

The BLE link-layer security mechanisms (such as pairing and bonding) were explicitly disabled by setting all characteristics to PERMISSION_READ and PERMISSION_WRITE, which simulates a scenario in which the attacker has already bypassed the built-in BLE protections and can freely intercept application-layer traffic. The apps exchanged encrypted messages representing random identifiers and timestamps, formatted as Device [XXXX] @HH:MM:SS.

6.4.3. Simulated Attack Workflow

To evaluate whether the vulnerabilities detected by BLECrypto could be practically exploited, we conducted a simulated attack under the threat model defined in Section 2.1. The experiment reproduced the essential steps that a passive adversary might follow to recover plaintext BLE communication from an unmodified Android app exhibiting cryptographic misuses. Figure 6 illustrates the five-step workflow executed in our simulated attack, which we elaborate below.

1. **Static analysis:** In practice, an attacker may reverse-engineer the central APK using off-the-shelf tools such as JADX to extract hardcoded cryptographic parameters. In our experiment, we used BLECrypto to automatically detect and retrieve the encryption key and IV from the app, demonstrating that the misuses reported by our tool can directly facilitate real-world attacks.
2. **Traffic capture:** BLE packets were captured using an nRF52840 sniffer during communication between the central and peripheral apps.
3. **Data extraction:** Encrypted attribute values were extracted from ATT Write Requests within the pcapng traces.
4. **Decryption:** A Python script leveraging the PyCryptodome library performed AES-CBC decryption using the key and IV obtained from BLECrypto.
5. **Verification:** The recovered plaintext messages were matched against the original application-side logs to confirm correctness.

6.4.4. Results and Analysis

Every captured BLE message was successfully decrypted using the key and IV reported by BLECrypto, confirming the exploitability of the detected misconfigurations. For example, the

ciphertext 4cda00fcda7942d1... (truncated for brevity) was decrypted to Device[4475]@16:40:57.

These results demonstrate that hardcoded credentials and insecure encryption modes, as detected by BLECrypto, can lead to full plaintext recovery under realistic threat assumptions. The attack was conducted without any access to the BLE peripheral's firmware or source code. In particular, we successfully decrypted messages originating from the peripheral device using only the key and IV extracted from the Android companion app, which strongly supports the assumption of behavioral symmetry in BLE application-layer encryption and confirms that BLECrypto's findings are not merely theoretical. They are directly exploitable in practice using low-cost hardware and standard tools, as further exemplified by real-world cases such as CVE-2025-60250 [38].

7. Limitations and Discussion

In this section, we outline several limitations of our tool and experimental setup that may have affected our results.

Data relaying beyond BLE endpoints. While BLE devices transmit data to Android applications, the companion application is not necessarily the final destination. In practice, an app may serve as a relay, forwarding BLE data to other apps, devices, or remote servers. In such cases, BLECrypto may be unable to observe the full encryption process, potentially leading to false negatives. However, this does not imply that the data is transmitted in plaintext; rather, the encryption may occur outside the observable scope of our static analysis.

Mitigate false positives by the confidence mechanism. Mitigating false positive is a common challenge for static analysis tools [36]. Since static analysis does not execute actual program paths, it will fail to capture cases with control-flow branches, ambiguous data dependencies, and indirectly propagated encryption logic. BLECrypto adopts a confidence-based mechanism to mitigate the false positive problem. The two confidence levels are explicitly reported that users can interpret the findings based on their own security settings.

Limited detection scope for custom cryptographic logic. Our detection targets cryptographic operations implemented through the Java Cryptography Architecture (JCA), including direct API usage and third-party libraries that internally delegate to JCA. However, encryption logic that bypasses JCA entirely, e.g. custom cryptographic implementations, is outside the scope of our analysis, and will be our future work.

No runtime validation of exploitability. While BLECrypto statically detects cryptographic misuses, its exploitability validation is currently demonstrated through a representative proof-of-concept scenario (Section 6.4), rather than across arbitrary real-world firmware or hardware configurations. Comprehensive runtime validation typically requires access to proprietary firmware or hardware instrumentation, which remains beyond the current scope. In future work, we plan to explore selective runtime tracing or emulation-based validation to broaden the practical assurance.

Ethical considerations. This study was conducted solely for academic purposes, aiming to assess the prevalence of cryptographic misuses in BLE-related Android applications. All analysis results are anonymized, and no findings related to specific apps or vendors are disclosed. We plan to report relevant security issues to the respective stakeholders through appropriate and responsible channels.

Justification for app-side proxy analysis. BLECrypto assumes that cryptographic behavior in BLE devices is often reflected in the corresponding Android companion apps. This assumption

is supported by prior studies [13, 14], which demonstrate that BLE security configurations are frequently mirrored in app code. Moreover, practical evidence, such as our own proof-of-concept experiment (Section 6.4) and real-world cases [38], shows that cryptographic misuses in BLE ecosystems can lead to full message decryption, even when only one endpoint is analyzed.

8. Conclusion

In this paper, we presented BLECrypto, a static analysis tool designed to detect cryptographic misuse in BLE application-layer encryption paths within Android applications that utilize BLE communication. By combining semantics-guided data-flow analysis with control-flow reasoning, BLECrypto incorporates five dedicated analyzers that collectively trace the propagation of BLE-related data and identify potential cryptographic operations and misuses. The project repository of BLECrypto is available at <https://github.com/Zhu-Hacken/BLECrypto-artifact>.

We conducted a large-scale evaluation on **67,509** real-world Android applications. Our results show that, on average, approximately **73.2%** of APKs employing encryption for BLE communication exhibit at least one instance of typical cryptographic misuse. Even under a strict setting that considers only high-confidence traces, **65.6%** of apps still violate at least one rule. These findings demonstrate the effectiveness and practicality of BLECrypto and further highlight the widespread misuse and lack of secure design awareness in application-layer encryption for BLE communications. Beyond large-scale prevalence, we further confirmed the practical exploitability of such misuses through a proof-of-concept attack.

Author Contributions

Haiqin Zhu carried out the research work, including the design and implementation of the experiments, data analysis, and manuscript drafting.

Weijing You supervised the overall study, provided the research direction, guided the experimental design and interpretation, and revised the manuscript critically for important intellectual content.

Wenqing Li contributed to discussions on methodology and offered constructive suggestions for improving the work.

All authors read and approved the final manuscript.

Acknowledgement

This work was supported in part by the Young Scientists Fund of the National Natural Science Foundation of China (No. 62202102), the Natural Science Foundation of Fujian Province, China (No. 2024J08162), and the CCF-NSFOCUS ‘Kunpeng’ Research Fund (No. CCF-NSFOCUS2024004).

Availability of Data and Materials

The APKs used in this study were obtained from the AndroZoo [33] dataset, which is available to academic researchers upon request. Due to licensing restrictions and the large-scale nature of the dataset (over 67,000 apps), we do not redistribute the APKs directly. The dataset filtering criteria are described in Section 6.1.

References

- [1] G. Koulouras, S. Katsoulis, and F. Zantalis, "Evolution of bluetooth technology: Ble in the iot ecosystem," *Sensors*, vol. 25, no. 4, p. 996, 2025.
- [2] J. Wu, Y. Nan, V. Kumar, D. J. Tian, A. Bianchi, M. Payer, and D. Xu, "{BLESAs}: Spoofing attacks against reconnections in bluetooth low energy," in *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.
- [3] Y. Zhang, J. Weng, R. Dey, Y. Jin, Z. Lin, and X. Fu, "On the (in) security of bluetooth low energy one-way secure connections only mode," *arXiv preprint arXiv:1908.10497*, 2019.
- [4] P. Sivakumaran and J. Blasco, "A study of the feasibility of co-located app attacks against {BLE} and a {Large-Scale} analysis of the current {Application-Layer} security landscape," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1–18.
- [5] W. Li, S. Jia, L. Liu, F. Zheng, Y. Ma, and J. Lin, "Cryptogo: Automatic detection of go cryptographic api misuses," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 318–331.
- [6] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. Yao, "Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2455–2472.
- [7] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 2013, pp. 73–84.
- [8] C. Zuo, H. Wen, Z. Lin, and Y. Zhang, "Automatic fingerprinting of vulnerable ble iot devices with static uuids from mobile apps," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1469–1483.
- [9] Q. Zhao, C. Zuo, J. Blasco, and Z. Lin, "Periscope: Comprehensive vulnerability analysis of mobile app-defined bluetooth peripherals," in *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security*, 2022, pp. 521–533.
- [10] L. Zhang, J. Chen, W. Diao, S. Guo, J. Weng, and K. Zhang, "{CryptoREX}: Large-scale analysis of cryptographic misuse in {IoT} devices," in *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 151–164.
- [11] Y. Zhang, J. Weng, R. Dey, Y. Jin, Z. Lin, and X. Fu, "Breaking secure pairing of bluetooth low energy using downgrade attacks," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 37–54.
- [12] J. Wang, F. Hu, Y. Zhou, Y. Liu, H. Zhang, and Z. Liu, "Bluedoor: breaking the secure information flow via ble vulnerability," in *Proceedings of the 18th ACM/IEEE International Conference on Mobile Systems, Applications, and Services*, 2020, pp. 286–298.
- [13] D. Schmidt, C. Tagliaro, K. Borgolte, and M. Lindorfer, "Iotflow: Inferring iot device behavior at scale through static mobile companion app analysis," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 681–695.
- [14] J. Du, F. Xu, C. Zhang, Z. Zhang, X. Liu, P. Ren, W. Diao, S. Guo, and K. Zhang, "Identifying the ble misconfigurations of iot devices through companion mobile apps," in *Proceedings of the 2022 19th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, 2022, pp. 343–351.
- [15] F. Draschbacher and J. Feichtner, "Cryptoshield-automatic on-device mitigation for crypto api misuse in android applications," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, 2023, pp. 899–912.
- [16] W. Zhou, Y. Jia, Y. Yao, L. Zhu, L. Guan, Y. Mao, P. Liu, and Y. Zhang, "Discovering and understanding the security hazards in the interactions between {IoT} devices, mobile apps, and clouds on smart home platforms," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1133–1150.
- [17] Y. Nan, X. Wang, L. Xing, X. Liao, R. Wu, J. Wu, Y. Zhang, and X. Wang, "Are you spying on me?(Large-Scale) analysis on {IoT} data exposure through companion apps," in *Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6665–6682.
- [18] P. Sivakumaran, C. Zuo, Z. Lin, and J. Blasco, "Uncovering vulnerabilities of bluetooth low energy iot from companion mobile apps with ble-guuide," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, 2023, pp. 1004–1015.
- [19] A.-K. Wickert, L. Baumgärtner, F. Breitfelder, and M. Mezini, "Python crypto misuses in the wild," in *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2021, pp. 1–6.
- [20] S. Rahaman, H. Cai, O. Chowdhury, and D. Yao, "From theory to code: identifying logical flaws in cryptographic implementations in c/c++," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 6, pp. 3790–3803, 2021.

- [21] L. Piccolboni, G. Di Guglielmo, L. P. Carloni, and S. Sethumadhavan, “Crylogger: Detecting crypto misuses dynamically,” in *Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1972–1989.
- [22] C. An, D. Zhang, X. Gao, and X. Zhu, “Cryptodetection: A cryptography misuse detection method based on bi- lstm,” in *Proceedings of the 2022 IEEE 8th International Conference on Computer and Communications (ICCC)*. IEEE, 2022, pp. 1244–1249.
- [23] L. Wang, J. Wang, T. Sui, L. Kong, and Y. Zhao, “Intelligent detection of cryptographic misuse in android applications based on program slicing and transformer-based classifier,” *Electronics*, vol. 12, no. 11, p. 2460, 2023.
- [24] Y. Zhang, B. Li, J. Lin, L. Li, J. Bai, S. Jia, and Q. Wu, “Gopher: High-precision and deep-dive detection of cryptographic api misuse in the go ecosystem,” in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 2978–2992.
- [25] S. Ma, D. Lo, T. Li, and R. H. Deng, “Cdrep: Automatic repair of cryptographic misuses in android applications,” in *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security*, 2016, pp. 711–722.
- [26] I. Muslukhov, Y. Boshmaf, and K. Beznosov, “Source attribution of cryptographic api misuse in android applications,” in *Proceedings of the 2018 ACM Asia Conference on Computer and Communications Security*, 2018, pp. 133–146.
- [27] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler *et al.*, “Cognicrypt: Supporting developers in using cryptography,” in *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 931–936.
- [28] Q. Wang, J. Li, Y. Zhang, H. Wang, Y. Hu, B. Li, and D. Gu, “Nativespeaker: Identifying crypto misuses in android native code libraries,” in *Proceedings of the International Conference on Information Security and Cryptology*. Springer, 2017, pp. 301–320.
- [29] A. Desnos, “Androguard: Reverse engineering and pentesting for android applications,” <https://github.com/androguard/androguard>, 2025, accessed May 30, 2025.
- [30] F. E. Allen, “Control flow analysis,” *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, 1970.
- [31] F. E. Allen and J. Cocke, “A program data flow analysis procedure,” *Communications of the ACM*, vol. 19, no. 3, p. 137, 1976.
- [32] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, “Androzoo: Collecting millions of android apps for the research community,” in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR ’16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: <http://doi.acm.org/10.1145/2901739.2903508>
- [33] M. Alecci, P. J. R. Jiménez, K. Allix, T. F. Bissyandé, and J. Klein, “Androzoo: A retrospective with a glimpse into the future,” in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 389–393.
- [34] E. Barker and A. Roginsky, “Transitioning the use of cryptographic algorithms and key lengths,” National Institute of Standards and Technology (NIST), Tech. Rep., 2018.
- [35] E. Barker and N. Mouha, “Recommendation for the triple data encryption algorithm (tdea) block cipher,” National Institute of Standards and Technology (NIST), Tech. Rep., 2017.
- [36] Y. Chen, Y. Liu, K. L. Wu, D. V. Le, and S. Y. Chau, “Towards precise reporting of cryptographic misuses,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS 2024)*, 2024.
- [37] E. Barker and Q. Dang, “Nist special publication 800-57 part 1, revision 5: Recommendation for key management: Part 1–general,” National Institute of Standards and Technology (NIST), Tech. Rep., 2020.
- [38] National Vulnerability Database, “Cve-2025-60250,” <https://nvd.nist.gov/vuln/detail/CVE-2025-60250>, 2025, use of hard-coded cryptographic key and IV in BLE communication; Accessed February 3, 2026.

Author Biography

Haiqin Zhu is a master’s student at the College of Computer and Cyberspace Security, Fujian Normal University. His research interests focus on cryptographic API misuse detection and mobile application security.

E-mail: qsx20231390@student.fjnu.edu.cn; ORCID: 0009-0003-3548-0382

Weijing You received her Ph.D. degree in information security from the University of Chinese Academy of Sciences in 2021. She is currently an associate professor with the College of Computer and Cyber Security at the Fujian Normal University. Her research interests are in applied cryptography and data security. She has published several referred research papers

in conferences and journals, including NDSS, ESORICS, IEEE TIFS, Cybersecurity. She is a senior member of China Computer Federation (CCF).

E-mail: youweijing@fjnu.edu.cn; ORCID: 0000-0001-7060-5820

Wenqing Li is a Research and Development Engineer at Venustech Group Inc. She earned her Master's degree in Cyberspace Security from the University of Chinese Academy of Sciences in 2023. Currently, she is engaged in cybersecurity-related work, primarily responsible for the engine development of web vulnerability scanning products. Her technical expertise is centered on the Go (Golang) programming language, and she maintains a focused interest in security detection products and technologies.

E-mail: li_wenqing@venusgroup.com.cn; ORCID: 0009-0001-4970-4872