MtdScout: Complementing the Identification of Insecure Methods in Android Apps via Source-to-Bytecode Signature Generation and Tree-based Layered Search

Zicheng Zhang
Singapore Management University
Singapore, Singapore
zczhang.2020@phdcs.smu.edu.sg

Haoyu Ma*
Zhejiang Lab
Hangzhou, China
hyma@zhejianglab.com

Daoyuan Wu†
HKUST
Hong Kong SAR, China
daoyuan@cse.ust.hk

Debin Gao
Singapore Management University
Singapore, Singapore
dbgao@smu.edu.sg

Xiao Yi
CUHK
Hong Kong SAR, China
yx019@ie.cuhk.edu.hk

Yufan Chen
Singapore Management University
Singapore, Singapore
yufanchen@smu.edu.sg

Yan Wu‡
Morgan Stanley
Shanghai, China
yan.wu@ms.com

Lingxiao Jiang
Singapore Management University
Singapore, Singapore
lxjiang@smu.edu.sg

***Abstract*—Modern Android apps consist of both host app code and third-party libraries. Traditional static analysis tools conduct taint analysis for API misuses on the entire app code, while third-party library (TPL) detection tools focus solely on library code. Both approaches, however, are prone to some inherent false negatives: taint analysis tools may neglect third-party libraries or face timeouts/errors in whole app-based analysis, and TPL detection tools are not designed for pinpointing specific vulnerable methods. These challenges underscore the need for enhanced identification of insecure methods in Android apps, particularly for app markets addressing open-source security incidents.**

In this paper, we aim to complement the identification of missed false negatives in both TPL detection and taint analysis by directly identifying clones of insecure methods, regardless of whether they are in the host app code or a shrunk library. We propose MtdScout, a novel cross-layer, method-level clone detection tool for Android apps. MtdScout generates bytecode signatures for flawed source methods using compiler-style interpretation and abstraction, and efficiently matches them with target app bytecode using signature-mapped search trees. Our experiment using ground-truth apps shows that MtdScout achieves the highest accuracy among three tested clone detection tools, with a precision of 92.5% and recall of 87.2%. A large-scale experiment with 23.9K apps from Google Play demonstrates MtdScout's effectiveness in complementing both LibScout and CryptoGuard by identifying numerous false negatives they missed due to app shrinking, method-only cloning, and inherent timeouts and failures in expensive taint analysis. Additionally, our experiment uncovers four security findings that highlight the disparities between MtdScout's method-level clone detection and package-level library detection.

*Haoyu Ma and Zicheng Zhang are the co-first authors.
†Corresponding author: Daoyuan Wu. Work conducted while at CUHK.
‡Work conducted by Yan Wu while he was an MSc student at CUHK.

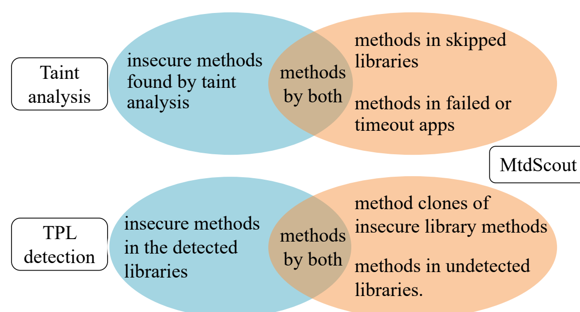


Figure 1: Complementing app-oriented taint analysis and TPL-based methods to address their false negatives.

1. Introduction

Modern Android apps typically comprise not only the host app code but also third-party libraries, such as those for common functionalities and advertisement [51], [39]. However, in the event of open-source security incidents, vulnerabilities such as API misuse tend to be present across a spectrum of insecure methods, subsequently impacting apps that *import* or *replicate* code from these methods. For example, the vulnerability discovered in the Log4j library, which enabled attackers to execute arbitrary code [3], [2], had a widespread impact on numerous global companies [19]. It is thus crucial for app markets [34] as well as companies with the Bring-Your-Own-Device policy [65] to promptly assess the potential impact of insecure methods on their apps. However, both app markets and companies often lack access to the source code of these apps, making it challenging to use source code clone detection tools for identifying those insecure methods.

Current methods for scanning open-source-introduced insecure methods in Android bytecode include app-centric static taint analysis and detection approaches based on third-party libraries (TPL). Specifically, traditional static analysis tools, including FlowDroid [27], Amandroid [67], IccTA [50], JN-SAF [66], CryptoGuard [58], and BackDroid [68], treat the entire app code as a whole and perform control- or data-flow taint analysis for API mis-

uses. In contrast, third-party library (TPL) detection tools, such as the state-of-the-art ATVHunter [72], PHunter [70] LibScan [69] and LibScout [30], focus exclusively on the library code. These TPL detection tools employ library package-level features and relationships to detect vulnerable libraries similar to those in their databases.

However, as illustrated in Figure 1, both approaches are susceptible to some inherent false negatives. Specifically, due to the high overhead associated with Android taint analysis [28], taint analysis tools often neglect third-party libraries [67] or encounter timeouts and internal errors during whole app-based analysis [58]. As a result, any insecure method in a missed library or in a failed app would go undetected, resulting in a false negative. Likewise, while TPL detection tools excel in identifying libraries, they are not designed to pinpoint *specific* vulnerable methods. Hence, when developers repurpose individual methods or classes from libraries – for example, only an insecure library method is cloned into the app's code – a false negative could occur. Additionally, package-level library detection can be affected by Android's default app shrinking [10]. This means that even if an insecure library method is present in an app, other classes and methods from the corresponding library might be shrunk, preventing the library from being identified by TPL detection tools. These challenges underscore the need to improve the identification of insecure methods in Android apps, which is particularly vital for app markets and companies that need to swiftly identify potential risks following open-source security incidents.

In this paper, we aim to address the inherent false negatives in existing app-oriented taint analysis and TPL-based detection methods. Our insight is to *directly identify clones of insecure methods*, ensuring coverage whether they reside in the host app code or within a shrunk library. By doing so, we also eliminate the need for expensive app-oriented taint analysis and enhance the granularity of library-based detection. Based on this insight, we propose MtdScout, a novel cross-layer, method-level clone detection tool tailored for detecting vulnerable open-source method clones in Android apps. MtdScout employs bytecode-level signatures, generated using a compiler-style interpretation and abstraction of the source code, enabling an efficient search for method clones within Android app bytecode. This positions MtdScout as the first[1] tool able to detect method-level code clones in Android app DEX bytecode. Furthermore, MtdScout's cross-layer clone detection differentiates itself from other tools [62], [29], [35], [73], [40] that either transform code into an intermediate representation (e.g., LLVM bit code [29]) or rely on supplemental context (e.g., `git` commit logs [35]). In contrast, MtdScout neither bases its comparisons on intermediate representations nor requires additional context.

To build MtdScout, a notable challenge is how to generate *adequate* bytecode-level signatures from partial Java source code, and in a way that they can be *precisely searched*. We address this challenge by employing two measures proposed in §4. Firstly, to generate adequate signatures that incorporate type information (e.g., a method's return type), which is often absent in standalone library source files, MtdScout begins by constructing a cross-reference dictionary covering all essential details about variables, classes, and methods. Subsequently, an abstract syntax tree (AST) is built to restore the types based on the cross-reference dictionary, followed by converting statements into bytecode-level signatures. Secondly, to facilitate an accurate search of the generated signatures across various Android APK settings, MtdScout identifies and follows major Android compiler optimizations to adjust signatures at the bytecode level.

Another challenge is how to efficiently match hundreds, if not thousands or more, of generated signatures with each target bytecode, which can contain millions to tens of millions of lines of code[2]. Moreover, this matching also needs to be performed across large-scale apps, as it is the designated scenario for MtdScout. To this end, we propose a novel layered search over signature-mapped search trees. Specifically, MtdScout divides each signature into two parts, the order-sensitive and order-independent portions, and organizes the signatures of identified insecure methods into a pair of search trees. MtdScout then conducts a layered search over an app-specific search tree to narrow down the matching space of a target bytecode progressively. Eventually, MtdScout requires only a depth-first search (DFS)-based matching with the limited candidate bytecode. This approach allows MtdScout to directly pinpoint the affected usage of a specific method.

To evaluate MtdScout, we first design a quantitative experiment using non-obfuscated ground-truth apps to assess the accuracy of MtdScout. We compared the performance of MtdScout with two commonly used source code clone detection tools, namely SourcererCC [59] and ReDeBug [48]. Our experiment showed that MtdScout achieves the highest accuracy, with an F1 score of 89.5%, precision of 92.5%, and recall of 87.2%; see §6.2.

We further conduct a large-scale experiment to assess MtdScout effectiveness in complementing the identification of missed false negatives in both TPL detection and taint analysis. For this purpose, we collected a large dataset comprising 23,962 popular apps from all the 51 Google Play categories and applied MtdScout to the specific[3] crypto misuse problem [43]. In this experiment, we compared the results obtained by MtdScout with those from LibScout [30], a widely used open-source library detection tool, and CryptoGuard [58], a dedicated crypto misuse detection tool that employs state-of-the-art crypto-specific slicing techniques [58], [68].

The evaluation results show that MtdScout effectively complements both LibScout and CryptoGuard by identifying the false negatives they missed. Specifically, among the 2,152 library-application pairs detected exclusively by MtdScout from the 18 vulnerable libraries tested by both tools, we sampled and found that around 32% of these false negatives were due to Android app shrinking [10], and 63% were caused by method-only cloning. Only 5% were MtdScout's own false positives. Furthermore, MtdScout identified a considerable number of false negatives missed by CryptoGuard, about 47 apps per category out

1. While there are some Android clone detection tools, e.g., [37], [33], [60], they were for detecting cloned apps using a source app as input. Our input is an insecure method from an open-source project or library.

2. Based on our measurement of the bytecode of 23.9K apps in §6.3.

3. MtdScout can detect various types of source code security issues, but for the purpose of evaluation, we need to focus on a specific problem.

of the total 470 apps in each Google Play category. Many of these were due to CryptoGuard's timeouts and errors, which are difficult to avoid due to the inherently expensive nature of whole app-based analysis like that performed by CryptoGuard. In contrast, MtdScout was able to complement the identification of 5,897 insecure methods from the 2,304 apps that CryptoGuard could not analyze. Beyond its effectiveness, MtdScout also demonstrated significantly improved efficiency compared to CryptoGuard, with a median analysis time of only 52 seconds per app, which is 31.6 times faster than CryptoGuard.

Our large-scale experiment not only identified vulnerabilities in apps from Internet giants like Tencent and Xiaomi, but also revealed four security findings that highlight the disparities between MtdScout's method-level clone detection and traditional package-level library detection. These findings include: (i) the potential removal or modification of class files within the host app package due to Android app shrinking, leading to lower similarity scores in package-level library detection; (ii) the prevalence of code replication in apps instead of library imports, rendering package-level library detection ineffective; (iii) the unreliable nature of package-level library detection in identifying vulnerable library methods, as the presence of a library in an app does not guarantee corresponding vulnerable methods; and (iv) the existence of multiple crypto misuses within single methods, a finding that cannot be revealed by package-level approaches. These findings highlight the importance of method-level detection in effectively uncovering vulnerabilities and security issues, surpassing the limitations of package-level analysis.

Availability. To facilitate future research, we have made the dataset and our evaluation results publicly available on https://github.com/MtdScout/MtdS_Dataset.

2. Background and Related Work

DEX Bytecode and dexdump File. While Android apps are developed in Java/Kotlin, they are compiled into Dalvik Executable (DEX) format bytecode when generating the Android application package (APK), the installation package for Android apps. The DEX bytecode is a highly structured data file, and we can use an official Android tool called `dexdump` [1], [14] to convert it into a dexdump file, which is a human-readable text. In this paper, MtdScout's searching process is based on the DEX format instructions in the dexdump file [12].

Cryptographic Misuse. Developers use cryptographic primitives such as block ciphers and message authentication code (MAC) to enhance the security of data and communications [43]. However, despite well-defined cryptographic concepts aimed at ensuring security, developers may incorrectly implement cryptography (i.e., cryptographic misuse) in their apps due to either a lack of cryptographic knowledge or human error, leading to a false sense of security [54]. Existing cryptographic misuse detection tools are typically categorized into two broad groups: static analysis (e.g., CryptoLint [43], Binsight [56], MalloDroid [44], CryptoGuard [58], BackDroid [68], etc.) and dynamic analysis (e.g., SMVHunter [61], AndroSSL [46], etc.). In this paper, although MtdScout is designed to identify a variety of DEX code issues, our specific focus for evaluation is on detecting cryptographic misuse problems.

Detection of (open-source) libraries/projects in apps. Existing library detection tools utilize three main approaches: white-listing, feature matching, and clustering. Book et al. [32] and Grace et al. [47] compile lists of well-known ad libraries. LibScout [30] analyzes the package structure for feature generation. Wukong [63] and LibRadar [55] generate features based on the system APIs used by libraries. LibD [53] utilizes call graphs, inheritance, and inclusion relations for clustering. LibExtractor [74] uses class dependency relations for candidate clustering. StubDroid [26] generates a summary for each library class based on the data flow of the library's bytecode or source code. Similarly, OSSPolice [41] employs project-level features (i.e., normalized class signatures, string constants, and function centroids) to detect license violations from the use of open-source software. These tools primarily focus on the high-level package structure of libraries, comparing features of entire packages. In contrast, MtdScout employs fine-grained, function-level features (i.e., API invocation signatures, string constants and their lengths and concatenations, and exception signatures) to focus on identifying insecure method clones derived from open-source code.

(Cross-layer) Code Clone Detection. Existing code clone detection methods typically require the query and target input to be at the same level. DECKARD [49] generates characteristic vectors from the source code's Abstract Syntax Tree (AST) and detects code clones by calculating the vector similarity. SourcererCC [59] tokenizes source code blocks for clone detection, CCAligner [64] and DroidCC [24] use an order-sensitive token model with a sliding window, and AndroClonium [60] detects bytecode-level code clones using execution traces. AnDarwin [38] utilizes Program Dependency Graphs (PDG) to calculate semantic vectors for individual methods at the bytecode level and employs multiple clustering to efficiently detect method clones across a large number of apps. NiCad [36] compares normalized fragments line by line using an optimized longest common subsequence algorithm to detect clones. CCLearner [52] tokenizes the code of clone and non-clone method pairs to build a deep learning-based classifier for method clone detection.

Some approaches have proposed cross-layer clone detection tools to meet diverse detection needs. FIBER [73] matches fine-grained binary signatures for C++ security patches, OSSPatcher [40] identifies unpatched apps by comparing patch features and applying compiled patches, Feichtner et al. [45] convert ARM binaries to LLVM intermediate representation, and CLCMiner [35] utilizes git commit logs for additional insights. In contrast, MtdScout is the first tool to detect method-level code clones in DEX code using the source code of subject methods as input.

3. MtdScout Overview

3.1. Threat Model

The threat model of this paper is based on three assumptions: (1) Adversaries have the capability to exploit zero-day vulnerabilities within open-source libraries to target vulnerable applications. However, they cannot directly modify the applications owned by the app market;

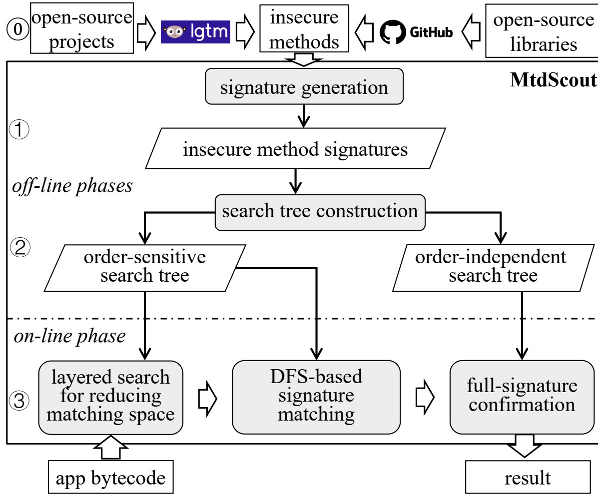


Figure 2: The overall workflow of MtdScout.

(2) Market owners or end users lack access to the source code of the applications but have the ability to acquire the source code of the vulnerable open-source libraries; and (3) App developers import or replicate methods from open-source libraries without disclosing library information to the market owner.

The objective is to facilitate market owners in swiftly identifying insecure methods within affected applications and estimating the impact on their repositories. We address two primary threats: (1) Attackers inject vulnerabilities into some open-source libraries, which are later used by developers in the victim apps; and (2) Developers of open-source libraries unintentionally introduce vulnerabilities into their libraries' methods.

3.2. Workflow

MtdScout follows a workflow depicted in Figure 2, consisting of three offline phases and one online phase.

In phase ⓪, we collect source-level insecure methods from open-source projects and libraries. For example, we can use LGTM [9] to identify insecure methods from open-source projects, or query the GitHub Advisory Database [16] to retrieve vulnerable open-source libraries.

In phase ①, MtdScout employs ANTLR [5] to construct an Abstract Syntax Tree (AST) for each Java source file containing the insecure method(s). Then, according to the description of each insecure method, MtdScout identifies its subtree in the AST. With the AST information, MtdScout generates a bytecode-level signature according to the design described in §4.

In phase ②, MtdScout partitions each signature into an order-sensitive portion and an order-independent portion. These portions are then mapped to a pair of overall search trees, covering both the order-sensitive and order-independent parts of the signatures, as detailed in §5.1.

After completing the offline preparations described above, MtdScout is ready to perform online operations in phase ③ by conducting layered searches on each method within the subject Android app's bytecode file. This process involves identifying nodes from the order-sensitive search tree at specific depths, allowing for efficient elimination of methods that cannot possibly match any signature (§5.2). Then, MtdScout performs signature matching in a depth-first search manner on the order-sensitive tree with pruning, seeking root-to-leaf paths that preserve the proper sequence of nodes within the same method. Finally, MtdScout verifies these potential matches against the order-independent search tree to confirm the presence of the order-independent portion in the matched signatures (§5.3).

4. Source-to-Bytecode Signature Generation

In this section, we first discuss in §4.1 the types of signatures that should be generated to ensure our signature-based matching is both adequate and resilient. Then, from §4.2 to §4.4, we elaborate on how we manage the three types of signatures during the source-to-bytecode signature generation process.

4.1. The Types of Signatures to Generate

Figure 3 illustrates an example signature (in Ⓑ) generated by MtdScout using solely the partial[4] source code information (e.g., a vulnerable method) depicted in Ⓐ. The objective here is that the generated signature could be *precisely searched* in the dexdump file of an app, as shown in the Ⓒ part of Figure 3. As such, our signatures should not only adequately reflect the original code semantics but also be resilient to common obfuscation and compiler optimizations. Specifically, Android apps often shield their code using syntax-level obfuscation tools like ProGuard [8], which has been incorporated into Android's default app shrinking process [10]. Moreover, bytecode compiler optimization further muddles the task of matching constant strings embedded in different apps.

The first and most important type of signature is *invocation signatures*, which depicts the key statements within a method. For these signatures, we focus on standard API methods rather than self-defined custom methods, and there are three reasons for this. Firstly, standard APIs are not subject to ProGuard-like obfuscation because the Android runtime would not recognize an obfuscated API invocation. For example, the standard API `MessageDigest.digest()` (highlighted in the red box) remains intact in the dexdump file, as seen in Ⓒ of Figure 3. In contrast, ProGuard would rename a self-defined method, such as `displayInt()` (shown in the blue box in Ⓐ) to a meaningless name like `b()` (indicated in the blue box in Ⓒ). Secondly, a method invocation in Java source might present only abbreviated class and method names. This makes determining their complete types challenging, particularly when involving non-standard class dependencies. Lastly, self-defined methods might undergo method inlining, where an invocation can be substituted by the body of method being called, rendering the search for this invocation unsuccessful.

Besides invocation signatures, we consider two other types of signatures, namely *constant strings* and *exception signatures*, to enhance the semantics of the generated signatures. Like standard APIs, constant strings remain unobfuscated, even after being processed by ProGuard, as illustrated by the green solid box in Ⓐ

4. As such, MtdScout cannot use a compiler to generate bytecode. Moreover, our signatures must be generalized for matching, whereas the signatures generated by compilers are intended only for execution.

Figure 3: An example of the Java source, the generated method signature, and the matched DEX code. For simplicity, some unimportant instructions are omitted. The legends in the top right corner explain the meanings of different boxes.

and Ⓒ. Moreover, try-catch exception information, such as `NoSuchAlgorithmException` (highlighted in the gray box of Figure 3), is another valuable supplement to differentiate between subtly distinct methods. We will explain the details of these two signatures and their generation in §4.3 and 4.4, respectively.

4.2. Type Recovery for Invocation Signatures

To faithfully generate invocation signatures from the Java source code, a notable challenge is to recover the full type information of the invocation statements. This includes (a) the method name, (b) the complete class name, (c) the full types of the parameters, (d) the modifier (e.g., `invoke-static` or `invoke-virtual`), and (e) the return type. This task is challenging even for standard API statements, especially for statements with multiple method invocations, because our AST parsing only provides the last-level class name, not its full type. For example, the AST node for `MessageDigest` does not capture the full type `Ljava/security/MessageDigest;` found in the dexdump file (see Ⓒ in Figure 3). As a result, when cross-referencing variables, classes, and method invocations to deduce the appropriate bytecode patterns for invocation signatures, the information provided by the AST nodes is incomplete. Gathering and correlating the context for each invocation statement is complex. To solve this problem, we build a cross-ref dictionary to recover the full type of each necessary element and identify the correct method information based on the source code's context.

Recovering full types using a cross-ref dictionary. MtdScout employs a dictionary-based approach to retrieve the aforementioned full type information, namely items (a) to (e), for crafting bytecode-level signatures. Specifically, MtdScout constructs a cross-reference dictionary for the given Java source code. This dictionary creates mappings between all source-level identifiers and their respective full type information. As MtdScout traverses through the method body, it looks up this dictionary to identify the comprehensive method description or class type for every invocation or variable declaration within the AST. While MtdScout can directly refer to the associated AST nodes to obtain item (a) and the abbreviated class/parameter names for items (b) and (c), it primarily leans on the dictionary to acquire the complete type names for items (b) and (c). Using these elements as the method descriptor, MtdScout then identifies the precise method by looking up the dictionary once more. Subsequently, details for items (d) and (e) can be derived from the metadata of the identified method. Throughout this procedure, MtdScout also takes into account the superclass and interfaces of each class for handling Java's polymorphism.

For the statements with multiple method invocations, their items (b) and (c) may be derived from the return values of other method invocations. Considering the third line in Ⓐ of Figure 3, the `digest()` method is invoked by the return value of the invocation `java.security.`

`MessageDigest.getInstance()`, and its parameter is the return value of the method `input.getBytes()`. Thus, accurately identifying an invocation only becomes possible after determining the full types of its caller class and/or parameters. To address this issue, we look to a characteristic of the AST. Within the AST, this statement line splits into two sub-statements: `MessageDigest.getInstance()` and `digest()`. These are both child nodes of the original statement node, with `digest()` positioned on the right side. Moreover, the node for `digest()` will have all nodes related to `input.getBytes()` as its descendants. This breakdown continues until the node represents a variable or class with a full type that can be directly referenced in the dictionary. Therefore, during a Depth-First Search (DFS), the statement nodes for `MessageDigest.getInstance()` and `input.getBytes()` are assured to be processed before the node for `digest()`. Leveraging this characteristic, MtdScout can resolve all method invocations within an AST based on the completion order of the DFS, consequently acquiring all type information for each node traversed.

During traversal, if the parameter type of a standard API method cannot be resolved (for instance, when the parameter is the return value of a non-standard method imported from another file), MtdScout checks if the dictionary contains only one potential match for that method, where the descriptions are consistent except for that parameter. If such a match is found, MtdScout selects that candidate as the identified method and continues the traversal. If not, it skips that method to prevent potential false positives, with one exception: when the invocation has been determined to be a method of a standard Java class but cannot be accurately identified due to parameter types. We handle this differently, as explained below.

Handling corner cases, invocation types, and register allocation using fuzzy signatures. Despite our best efforts, there are times when MtdScout might not be able to accurately identify a standard API method invocation because of difficulties in determining the types of its parameters. This challenge primarily arises because MtdScout's signature generation process is constrained to a file's scope. As a result, parameter types imported from another file might not be identified correctly. To mitigate this, when a standard API invocation slated for inclusion in a method signature has such unresolvable parameters, MtdScout substitutes its modifier, return type, and parameter types with regular expressions while constructing the corresponding bytecode instruction. This creates a *fuzzy pattern* that could still match, though it carries the potential risk of yielding false positives. For example, if the parameters of `String.getbytes()` cannot be resolved, MtdScout converts it into a fuzzy pattern like "`invoke-.* {(v[0-9]+, )*(v[0-9]+)?}, Ljava/lang/String;\.getBytes:\(.*\).*`". Note that this approach is merely a fallback for a minority of instances where method identification is unsuccessful. In the majority of cases, MtdScout is capable of crafting precise bytecode instruction patterns.

Besides the corner cases, MtdScout may not generate the exact bytecode instruction for (1) the modifiers of method invocations (e.g., `invoke-virtual`), (2) the allocation of registers (e.g., {`v0, v3`}), and (3) the actual addresses of exceptions (e.g., `0x0014`). Therefore, MtdScout generates a fuzzy pattern for ambiguous invocation signatures. Within this pattern, each uncertain element is substituted with a regular expression designed to accommodate all potential scenarios. For example, `invoke-(virtual|super|interface)` is used for modifiers, and `\{(v[0-9]+, )*(v[0-9]+)?\}` addresses register allocation, as shown in Ⓑ in Figure 3.

4.3. Addressing String Compiler Optimizations

Operations related to constant strings in a source-level method may undergo various optimizations during compilation to improve program performance. This can notably affect the accuracy of MtdScout's signature generation process and, subsequently, its signature matching. To address this issue, we handle three common types of string optimizations during compilation as follows.

Constant string location and sequence. Firstly, the compiler could relocate all constant strings to the beginning of a method's bytecode in an unpredictable sequence, ensuring each string appears only once, regardless of its usage frequency in the source code. Knowing this feature, MtdScout separates all constant string declarations involved in a signature from the method invocations and omits any repeated constant strings within the same method. This strategy influences the processing of constant strings during signature matching, which will be discussed later in §5.

Constant string concatenation. It is common for compilers to concatenate multiple constant strings, leading to various bytecode instruction versions based on the optimizations applied. For example, the string concatenation "I" + "am" + "good" appears as a single string "Iamgood" in bytecode, or as three separate strings ("I", "am", and "good") accompanied by two `java.lang.String.append()` invocations (see the 'string concatenation' in Figure 3). To address this ambiguity, MtdScout employs a unified approach to transform the constant string into regular expressions that accommodate both scenarios. Specifically, when a constant string is preceded and/or followed by a "+", MtdScout replaces it with a wildcard (representing any characters in the regular expression) and then adds it as an independent constant string into the signature. For instance, the aforementioned string "I" + "am" + "good" is converted to `"I.*"`, `".*am.*"`, and `".*good"`, with each segment required to match at the bytecode level, as described later in §5.3. Furthermore, this approach also accommodates concatenations between constant strings and variables, as illustrated in the second line of *CONSTANT STRING* in Ⓑ of Figure 3.

Length of constant strings. For a source-level statement that calculates the length of a constant string, the compiler might optimize it by directly encoding the string's length into the resulting bytecode as an integer, rather than invoking the API `java.lang.String.length()`. An example is shown in the 'string length' in Figure 3, where the expected return value of `"MD5".length()`, namely `#int 3`, is used directly after compilation. While this optimization eliminates the overhead of an API call, it is uncertain whether a particular app would actually implement it. To avoid this uncertainty during the signature generation process, MtdScout

refrains from generating a bytecode pattern for the API `java.lang.String.length()`.

4.4. Supplementing Signatures with the Try-Catch Exception Information

When a program encounters exceptions, programmers must handle them by catching the exceptions and writing code to address these situations. This information is also an important feature of a method that can help us more accurately identify the correct method in DEX code for two reasons. Typically, these exceptions are associated with specific method invocations and can help capture certain standard APIs that were missed during the previous method invocation conversion process. On the other hand, standard exceptions can also be imported by calls to self-defined methods. Since these self-defined methods are not the focus of the signature generation, including such exceptions in the signature can indicate the presence of some self-defined method invocations.

When traversing the method's AST, MtdScout retrieves exception types from the catch parentheses and converts them into DEX format signatures. In the dexdump file, these exceptions are listed separately after each method body, and the order of the exceptions is not fixed (see the last line of Ⓒ in Figure 3). MtdScout matches them in a manner similar to that used for constant strings, with the detailed matching process explained in §5.

5. Layered Signature Search and Matching

After generating signatures for insecure methods, MtdScout proceeds to the online phase, where it searches large-scale repositories for Android apps that may contain code clones matching these signatures. However, this task is not as straightforward. For example, matching 400 method signatures with an app comprising 30,000 methods using a traditional approach would require 12 million attempts, rendering it impractical. To address this, we propose a novel signature matching approach that significantly enhances the throughput of MtdScout in large-scale method clone detection. This approach maps signature patterns to search trees and employs a layer-based search strategy to efficiently reduce the search space.

5.1. Mapping and Merging Hundreds of Method Signatures onto Two Search Trees

As explained in §4.1, the signature for an insecure method generated by MtdScout includes standard API invocations, constant strings, and try-catch exceptions. Among these, the sequence of standard API invocations is the *order-sensitive* part of the signature, as the order of these instructions is a crucial feature of a method. On the other hand, the other two types are considered to be the *order-independent* part of the signature, as they can be arranged in any order in the bytecode by a compiler without affecting a method's semantics. Therefore, MtdScout divides the signatures accordingly, mapping the order-sensitive and order-independent parts of these signatures to a pair of search trees, respectively. For simplicity, we denote this pair of search trees as $T = \{T_{os}, T_{oi}\}$.

Figure 4 illustrates an example how MtdScout constructs $\{T_{os}, T_{oi}\}$. Specifically, T_{os} is designed with the following features:

- Each node is mapped to a specific API invocation pattern (e.g., `I1` in Figure 4) belonging to one or more method signatures (e.g., `Sig2`, `Sig3`, `Sig4`);
- The depth of a node in T_{os} indicates the position of the corresponding pattern within the respective signature(s). Hence, all nodes on the same path of T_{os} together represent the exact sequence of API invocations in a signature (e.g., `I1`, `I2`, `I3` of `Sig2`);
- The same invocation pattern can be mapped to multiple nodes at different depths for different signatures (e.g., `I2` of `Sig1` at depth 3 and `I2` of `Sig2` at depth 2).

Since different signatures are likely to start with distinct patterns, MtdScout builds a super root for T_{os} that points to all the nodes mapped to the first invocation patterns of the signatures, allowing every node of the tree to be reached with a single traversal. Additionally, for each path of the tree, a special-purpose leaf node is attached to serve as the descriptor of the exact signature indicated by the path, such as the leaf `Sig1`, `Sig2`, `Sig3`, and `Sig4` nodes in T_{os}. This signature descriptor contains the identity of the method and its corresponding project, facilitating the correlation between the two parts of a signature across the two search trees.

Following the design of T_{os}, T_{oi} also has each of its nodes mapped to either a constant string declaration pattern or a try-catch exception pattern. However, this second tree is constructed differently, with the super root pointing directly to signature descriptor nodes, which are duplicates of those in T_{os}. Each signature descriptor then points to a linked list of nodes that are mapped to patterns of the corresponding signature. The linked lists are not arranged in any specific order, as the nodes in T_{oi} are order-independent. Note that some signatures may only involve T_{os}, meaning they contain only API invocation signatures. In such cases, MtdScout sets a mark in the corresponding leaf node of T_{os} so that it can report matches for these signatures without consulting T_{oi}.

5.2. Reducing Bytecode Matching Space via Layered Search

Once the search trees are constructed, it is easy to observe that if a subject method matches with a signature containing an order-sensitive portion of x instruction patterns, then these patterns must appear on a specific path of T_{os} that is of depth x (excluding the super root and signature descriptor). By further relaxing this proposition, a fairly intuitive necessary condition for successful signature matching using the search tree pair T can be inferred: *For a subject method to be considered a potential match to a signature with x patterns in its order-sensitive portion, the method must hit at least one node at each depth of T_{os} until depth x.*

We found that this necessary condition can help MtdScout quickly reduce the search space for its signature-matching process. Specifically, as illustrated in Figure 4, MtdScout groups nodes of the same depth (excluding the super root and signature descriptors) from T_{os} into a *depth-specific search set*, creating a collection of such search sets for all possible depths of the tree. Let S_n denote the search set corresponding to depth n. For any particular method signature generated by MtdScout, if the

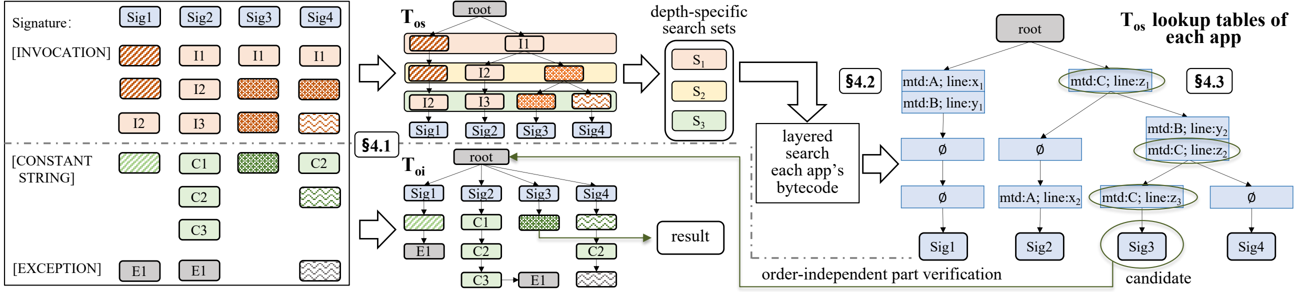


Figure 4: An example illustrating MtdScout's search tree construction and tree-based layered search.

nth line of its invocation pattern sequence exists, it is guaranteed to be included in S_n. Then, given the bytecode of a subject app, MtdScout screens each method using the search sets S_i where $(1 \le i \le m)$, with m being the maximum depth of T_{os}. Throughout this process, if a specific method fails to hit any of the nodes in a search set, say S_j, then the method will no longer be screened with the rest of the search sets S_i where $(j < i \le m)$, as the aforementioned necessary condition is certain to fail for longer signatures. Consequently, the layered search approach allows MtdScout to effectively narrow its focus and safely disregard impossible matches. To illustrate with an extreme example, if a method fails to match any node in S_2, and the depth of the shortest path of T_{os} is 3, then this method can be immediately dismissed since it cannot match any known signatures.

5.3. Matching Signatures in a DFS Manner

During the layered search, MtdScout associates each node of T_{os} with a lookup table, where it records the name of methods in which the very node has been found, along with the exact lines (bytecode addresses) that have been matched within these recorded methods. With this setup, MtdScout proceeds to conduct signature matching using a specially designed Depth-First Search (DFS) on T_{os}, followed by a targeted search on T_{oi}. Specifically, using the lookup tables, MtdScout performs a second round of rapid pruning on T_{os} to remove branches representing signatures that are guaranteed not to match any of the screened methods in the subject app. A node of T_{os} and its subtree will be pruned if any of the following three conditions are met: (1) the lookup table of the current node is empty; (2) there is no common method in the lookup table of the current node and its ancestor; or (3) for the same method in the lookup table of the current node and its ancestor, the order of matched lines is incorrect (e.g., the current node matches with the third line of a method while its ancestor node matches with the fourth line).

If a path toward a signature leaf node in T_{os} survives the aforementioned pruning (i.e., reaching the leaf node during the DFS), then the intersection of methods in the lookup tables on that path is guaranteed to match the signature's invocation part in the correct order. As such, these methods are considered candidates for potentially matching that signature. For example, method C in the T_{os} lookup tables shown in Figure 4 is a candidate for the signature `Sig3`. Recall that in §5.1, a signature may only involve T_{os} if it does not have an order-independent portion. In such cases, MtdScout directly reports the matching cases after the DFS on T_{os}. For other signatures, their potential candidates are sent for further comparison with T_{oi}. As also mentioned in §5.1, T_{oi} is organized differently, with the signature descriptors placed as direct children of the super root. Consequently, MtdScout can quickly obtain the order-independent portions of those signatures that survive the DFS on T_{os} (in the form of linked node lists) and then search for them in the candidate methods to make a final confirmation. For example, method C would be considered a true match only if all nodes in `Sig3`'s linked list on T_{oi} can also be matched.

Finally, considering that the signatures generated by MtdScout might sometimes overlap with each other, a method may be reported as matching multiple signatures, leading to potential false positives. To address this issue, MtdScout adopts a winner-takes-all strategy, where, if a method is identified as matching multiple signatures, only the longest signature among them is considered the true match. This rule is intuitively sensible, as longer signatures convey more information and are less likely to lead to incorrect matches.

6. Evaluation

In this section, we aim to comprehensively evaluate MtdScout by answering the following research questions:

RQ1: How *accurate* is MtdScout in terms of precision, recall, and F1 score?

RQ2: Can MtdScout's method-level clone detection *effectively* complement the identification of missed false negatives in traditional package-level TPL (third-party library) detection?

RQ3: Can MtdScout *effectively* complement the identification of missed false negatives in a state-of-the-art taint analysis tool?

RQ4: Can MtdScout *exclusively* identify new security findings through method-level detection?

RQ5: How *fast* is MtdScout in terms of running performance?

To answer these RQs, we need to put MtdScout into specific problems. While MtdScout can detect any kind of DEX code issues, we target the crypto misuse problem [43] in this paper for two reasons. First, it is the most common security issue in Android apps [43], [67], [54], [58], [68], [57], which allows for collecting a list of vulnerable libraries with cryptographic misuses for comparison with TPL tools. Second, it has a dedicated taint analysis tool called CryptoGuard [58], which leverages state-of-the-art crypto-specific slicing techniques [58], [68]. This allows for a comprehensive comparison of effectiveness and performance using concrete problems.

6.1. Vulnerable Methods Targeted in This Paper

To generate a set of vulnerable method signatures, we analyze 419 popular open-source Java libraries on GitHub using CodeQL [6] dataflow query scripts and LGTM [9]. Our search focuses on library methods with crypto misuses, which serve as input for MtdScout as explained in §3 (see Figure 2). We define six CodeQL query rules based on the specifications in CDRep [54] and CryptoLint [43], which correspond to the six rules of crypto misuses covered by CryptoGuard. These rules are listed below:

1) Encryption using *Electronic Codebook (ECB)* mode
2) Encryption using *Cipher-Block Chaining (CBC)* mode with a static initial vector.
3) Using a constant secret key in encryption.
4) Using constant salt in *Password Based Encryption (PBE)*.
5) The number of iterations for *PBE* is less than 1000.
6) Using a constant seed for *secureRandom()*.

From the LGTM query results, we gather descriptions of vulnerable methods, as well as the Java source files containing these methods. Using this information, MtdScout automatically generates a signature for each vulnerable method. Note that since LGTM outputs only the vulnerable methods, and not the entire call chains, we discard methods with only one or two lines of method invocations, as they can be easily matched with irrelevant methods, leading to false positives. Eventually, we collected 133 vulnerable methods from 44 libraries. From these results, MtdScout successfully generated 129 signatures for the analysis in this paper. The four failed cases all stem from an inability to locate the target method within the source code. Specifically, two of the cases involve constructors of anonymous inner classes that cannot be located with the provided information, and the other two are due to Lambda expressions (which use a "−>" to represent an anonymous method without explicitly indicating the name and return type, making it difficult for MtdScout to locate), a feature since Java 1.8.

6.2. Quantitative Experiment for Accuracy Measurement

To obtain the ground truth for accuracy measurement, we design a quantitative experiment that comprises a controlled number of open-source and closed-source apps (with no obfuscation). By employing state-of-the-art source code clone detection tools on the source code of the insecure methods and the target apps[5], and cross-checking their results with MtdScout's bytecode analysis, we can effectively minimize the manual effort required to confirm all the outputs at the source code level.

For source code clone detection tools, we select the widely-used SourcererCC [59] and ReDeBug [48]. SoucererCC [59] can be directly used with the source code of our experiment set of apps. Specifically, it uses a bag-of-token model to calculate the code similarity and supports different granularity of clone detection, including the method level. ReDeBug, on the other hand, is not a pure code clone detection tool because it relies on security patches as input. We thus adapt ReDeBug's design logic to our problem like BlockScope [71]. Specifically, instead of using patch code for locating contexts, we leverage the crypto APIs as hints for pinpointing the nearby code contexts. After locating the candidate code, our modified version of ReDeBug directly measures its code text similarity with the original library method source code.

Collecting the ground-truth app set. Our ground-truth app set is extracted from our large-scale app set described in §6.3. To facilitate the ground truth collection, this app set should have a controlled number so that manual confirmation is feasible, and it is better to have open-source code or the app code without obfuscation. We thus collect an open-source app subset and a closed-source app subset (with no obfuscation). For the former, we search the package names of all the collected apps on F-Droid [7] and GitHub [15], and select the top ten apps with the most crypto APIs. Due to the page limit and no significant crypto misuses identified in the open-source app subset, we put its details in Appendix A, where we also use one open-source app to demonstrate the obfuscation resiliency of MtdScout.

Similar to how we collect the ten open-source apps, we gather the closed-source app subset by searching for crypto APIs in the dexdump files (i.e., the textual form of bytecode in an APK) of apps in the Business category, considering apps in this category may use crypto APIs more frequently than others. After collecting the APKs containing crypto APIs, we filter out those obfuscated apps by checking their class names and method names. Eventually, we select the top ten apps with the most crypto APIs as our closed-source app subset, as shown in Table 1.

Experimental setup and procedures. With the collected two app subsets, we run them in a standard workstation with an eight-core 2.9GHz CPU, 16GB memory, and 300GB HDD. We first decompile each APK file into Java source code and disassemble the app into a dexdump file. We then run SoucererCC and the adapted ReDeBug separately on the decompiled source code to identify the clones of those vulnerable methods collected in §6.1. We also instruct MtdScout to analyze the dexdump file for our signature-based clone detection. No particular configuration is required for running MtdScout, but we set similarity thresholds for SoucererCC and the adapted ReDeBug. Additionally, we set up the nearby three lines of code (LOC) as the default context LOC for ReDeBug.

To preserve enough results for collecting ground truth, we set up a low similarity threshold, 0.4, for both SoucererCC and our adapted ReDeBug. This particular value is chosen for two reasons. First, it is low enough to allow us to collect many candidate clones since this value indicates that less than half of the two methods are similar. Second, if we further decrease the threshold from 0.4 to 0.3, the number of results outputted by the two tools will increase around ten times, which is nearly impossible to perform manual checking. We choose the upper bound at 0.7 because, at this threshold, the two tools will not output false positives for most of the apps in the two subsets, i.e., they can achieve the best precision at this threshold. Note that no threshold is required for MtdScout.

Accuracy measurement results. Table 1 shows the accuracy measurement result of our quantitative experiment for the closed-source app set (which has much more ground-truth results than the results of the open-source app set in Appendix A). We calculate the precision, recall, and

5. We obtain the source code of closed-source apps by decompilation.

TABLE 1: The accuracy measurement results of the closed-source app subset.

	com.gam_municipality					com.amazon.sellermobile.android					com.weenysoft.word2pdf					com.pms.activity					com.scanbizcards				
	TP|FP	GT	Preci	Rec	F1	TP|FP	GT	Preci	Rec	F1	TP|FP	GT	Preci	Rec	F1	TP|FP	GT	Preci	Rec	F1	TP|FP	GT	Preci	Rec	F1
MtdScout	13|0	13	100%	100%	100%	13|0	13	100%	100%	100%	13|1	13	92.9%	100%	96.3%	13|1	14	92.9%	92.9%	92.9%	14|0	18	100%	77.8%	87.5%
RDB 0.7	0|0	13	-	0%	-	0|0	13	-	0%	-	0|0	13	-	0%	-	0|0	14	-	0%	-	3|10	18	23.1%	16.7%	19.4%
RDB 0.6	0|0	13	0%	0%	-	1|13	13	7.1%	7.7%	7.4%	2|4	13	33.3%	15.4%	21.1%	1|15	14	6.3%	7.1%	6.7%	6|30	18	16.7%	33.3%	22.2%
RDB 0.5	4|217	13	1.8%	30.8%	3.4%	5|257	13	1.9%	38.5%	3.6%	8|164	13	4.7%	61.5%	8.6%	5|452	14	1.1%	35.7%	2.1%	10|493	18	2.0%	55.6%	3.8%
RDB 0.4	13|1,943	13	0.7%	100%	1.3%	12|2,135	13	0.6%	92.3%	1.1%	13|1,384	13	0.9%	100%	1.8%	13|3,681	14	0.4%	92.9%	0.7%	16|3,871	18	0.4%	88.9%	0.8%
SCC 0.7	0|0	13	-	-	-	0|0	13	-	-	-	0|0	13	-	-	-	0|0	14	-	0%	-	4|0	18	100%	22.2%	36.4%
SCC 0.6	0|0	13	-	0%	-	0|0	13	-	-	-	0|0	13	-	0%	-	1|0	14	100%	7.1%	13.3%	5|5	18	50%	27.8%	35.7%
SCC 0.5	0|1	13	0%	0%	-	0|0	13	-	0%	-	0|1	13	0%	0%	-	1|4	14	20%	7.1%	10.5%	7|45	18	13.5%	38.9%	20%
SCC 0.4	0|64	13	0%	0%	-	0|25	13	0%	0%	-	0|11	13	0%	0%	-	1|121	14	0.8%	7.1%	1.5%	7|210	18	3.2%	38.9%	6.0%

	com.delivery.india.client					com.infraware.office.link.lg					de.idnow					com.coolmobilesolution.fastscannerfree					com.redarbor.computrabajo				
	TP|FP	GT	Preci	Rec	F1	TP|FP	GT	Preci	Rec	F1	TP|FP	GT	Preci	Rec	F1	TP|FP	GT	Preci	Rec	F1	TP|FP	GT	Preci	Rec	F1
MtdScout	13|2	18	86.7%	72.2%	78.8%	13|2	16	86.7%	81.3%	83.9%	13|0	14	100%	92.9%	96.3%	11|4	18	73.3%	61.1%	66.7%	12|1	12	92.3%	100%	96.0%
RDB 0.7	3|11	18	21.4%	16.7%	18.8%	0|0	16	-	0%	-	0|0	14	-	0%	-	3|9	18	25.0%	16.7%	20%	1|0	12	100%	8.3%	15.4%
RDB 0.6	5|31	18	13.9%	27.8%	18.5%	1|29	16	3.3%	6.3%	4.3%	0|5	14	0%	0%	-	4|29	18	12.1%	22.2%	15.7%	1|21	12	4.5%	8.3%	5.9%
RDB 0.5	5|601	18	0.8%	27.8%	1.6%	7|918	16	0.8%	43.8%	1.5%	4|279	14	1.4%	28.6%	2.7%	8|803	18	1.0%	44.4%	1.9%	4|455	12	0.9%	33.3%	1.7%
RDB 0.4	15|5,212	18	0.3%	83.3%	0.6%	15|7,936	16	0.2%	93.8%	0.4%	13|2,722	14	0.5%	92.9%	0.9%	16|6,615	18	0.2%	88.9%	0.5%	11|4,189	12	0.3%	91.7%	0.5%
SCC 0.7	4|0	18	100%	22.2%	36.4%	0|0	16	-	-	-	0|0	14	-	0%	-	4|0	18	100%	22.2%	36.4%	0|0	12	-	0%	-
SCC 0.6	4|0	18	100%	22.2%	36.4%	0|0	16	-	0%	-	0|4	14	0%	0%	-	5|8	18	38.5%	27.8%	32.3%	0|2	12	0%	0%	-
SCC 0.5	5|40	18	11.1%	27.8%	15.9%	0|2	16	0%	0%	-	0|15	14	0%	0%	-	5|68	18	6.8%	27.8%	11.0%	0|7	12	0%	0%	-
SCC 0.4	5|228	18	2.1%	27.8%	4.0%	0|55	16	0%	0%	-	0|213	14	0%	0%	-	5|371	18	1.3%	27.8%	2.5%	0|172	12	0%	0%	-

F1 score for each app. The blue color indicates the highest number of each metric of an app. As shown in Table 1, MtdScout has the highest F1 score on all the ten apps in this subset, the highest precision on seven apps, and the highest recall on six apps. On average, MtdScout achieves 92.5% precision, 87.2% recall, and 89.5% F1 score.

In contrast, the adapted ReDeBug can only achieve a high recall when at the 0.4 threshold (similar to that on the open-source app subset). Its recall drops quickly when the threshold increases, and it even drops to 0 for five apps when the threshold increases to 0.7. On the other hand, SourcererCC does not output any result for six apps at four thresholds, and on the other four apps, it can only get less than 40% recall, which is the worst of the three tools. Although it can achieve nearly 100% precision, its total number of true positives is only 18, while the total ground truth is 150, which makes it less practical under our scenario. Note that the low recall is not caused by incomplete decompiled source code, since ReDeBug is conducted on the same input yet outputs true positives.

The metrics on both subsets show that MtdScout can achieve a high precision while maintaining a high recall without the need to set a threshold. Moreover, it outputs zero or very few false positives when an app does not contain a vulnerable method clone. Compared with other tools, they either achieve a high recall yet with a low precision or a high precision yet with a low recall.

Answer for RQ1: MtdScout demonstrates superior accuracy with 89.5% F1 score for apps in the closed-source set, averaging 92.5% precision and 87.2% recall.

6.3. Large-scale Experiment for Effectiveness Comparison

In this section, we conduct a large-scale experiment with three objectives. First, this experiment enables us to present the distinct results of MtdScout's method-level clone detection. Second, by comparing MtdScout's findings with those of LibScout [30], we can assess MtdScout's effectiveness in complementing the identification of missed false negatives in TPL detection. Third, we compare MtdScout's result with those of CryptoGuard [58] to assess its capability to complement the identification of missed false negatives in static taint analysis.

A large dataset. To build a large app dataset, we select the top apps from all the 51 categories on Google Play, such as Communication, Business, Social, etc. For each category, we collected the top 500 apps based on the ranking list from Androidrank [4]. We then collected the most recent Google Play version of these apps from AndroZoo [25]. Since AndroZoo does not have some of the apps, or the ranking itself contains less than 500 top apps, we were able to obtain, on average, 470 apps per category. Eventually, we collected a total of 23,962 APKs from Google Play for the experiment.

Experimental setup. We ran this large-scale experiment on a more powerful machine with 40×2.2GHz CPUs, 32GB memory and 2TB storage for parallel execution with multiple threads. Moreover, as mentioned in §6.1, MtdScout utilizes 129 insecure method signatures from 44 libraries for the analysis in this section.

MtdScout's results. MtdScout outputs a set of matched clone pairs for each app, i.e., $R_{MS} = \{<m_{sig}, m_{dex}>\}$, where m_{sig} represents the signature of the original insecure method and m_{dex} represents the detected method clone in the app's DEX bytecode. Among the 44 source libraries, 17 were detected by MtdScout with insecure method clones in our app set. Table 2 provides detailed method-level clone detection results, showing the 18,944 clone pairs between 45 method signatures of 17 libraries and the 7,618 methods found in 2,990 apps. This means that more than one-third of the total 129 method signatures have been cloned, with a single signature being matched with up to 1,464 method clones. Upon further examination of the method signatures across these 17 libraries, it was observed that three of them share two method signatures. Fortunately, these identical signatures do not affect the fairness of the comparison between MtdScout and LibScout, as none of the 18 libraries used for comparison share the same signature (refer to §6.3.1). Also, these signatures do not impact the fairness in comparing MtdScout with CryptoGuard, since having the same method signature does not alter the number of detected apps (see §6.3.2).

Out of the total 17 libraries in the result, *four* (24%) have method clones detected by MtdScout across all 51 categories of apps. Despite the maximum number of method signatures cloned from each library being only six, on average, each signature is associated with 421 clone pairs and involves 66 apps. Additionally, we observe that even libraries with a single vulnerable method signature can influence apps across all 51 categories. For example, the signature of Rule 3 from `apache/cloudstack` affects apps across all categories (see Table 2). These

TABLE 2: Detection results of MtdScout and LibScout (shown in bold font within brackets).

Library Name	# of unique items				# of method clone pairs						
	Sig	App	Mtd	Ctg	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Total
songxiaoliang/EncryptionLib	6	192	200	40	400	0	0	0	0	0	400
bwssytems/ha-bridge	6	**23 (0)**	24	**16 (0)**	25	0	0	25	25	0	75
zaproxy/zaproxy	4	1,583	4,000	51	5,424	0	0	0	0	0	5,424
AsyncHttpClient/async-http-client	4	1,583	3,443	51	4,866	0	0	0	0	0	4,866
MyCATApache/Mycat-Server	4	**27 (0)**	27	**18 (0)**	54	0	0	0	0	0	54
apache/httpcomponents-client	3	**1583 (2)**	3,002	**51 (2)**	4,425	0	0	0	0	0	4,425
aa112901/remusic	3	189	192	40	192	0	178	0	0	0	370
apache/poi	3	**17 (17)**	17	**8 (12)**	51	0	0	0	0	0	51
aws/aws-sdk-java	2	**210 (626)**	210	**42 (51)**	420	0	0	0	0	0	420
mike-ensor/aes-256-encryption-utility	2	169	169	41	0	169	169	0	0	0	338
alibaba/druid	2	**3 (0)**	3	**3 (0)**	6	0	0	0	0	0	6
apache/cloudstack	1	1,380	1,381	51	0	0	1,381	0	0	0	1,381
igniterealtime/Openfire	1	681	698	50	0	0	698	0	0	0	698
alibaba/nacos	1	**425 (0)**	425	**50 (0)**	0	0	425	0	0	0	425
HussainDerry/secure-preferences	1	1	1	1	1	0	0	0	0	0	1
matrix/Burp-JCryption-Handler	1	7	7	6	0	7	0	0	0	0	7
justauth/JustAuth	1	**3 (0)**	3	**3 (0)**	0	0	3	0	0	0	3
Union	45	2,990	7,618	51	15,864	176	2,854	25	25	0	18,944

The rows displayed in bold font indicate that the corresponding library was also tested by LibScout, with LibScout's number listed in brackets. "Sig" and "Ctg" stand for "signatures" and "categories" respectively. The number of signatures refers to those with a method-level clone result.

findings highlight the prevalence of vulnerable code clones across different types of applications, emphasizing the urgent and critical need to detect and address these clones.

Regarding the rules of crypto misuses (as outlined in §6.1), we identified 2,650 apps, each of which exhibits multiple types of crypto misuses. Furthermore, among the clone pairs, we identified 371 methods, each of which is linked to a single signature associated with multiple rules. A more comprehensive explanation of this finding can be found in §6.4. The most frequent issue, making up 83.7% (15,864) of the clone pairs, is related to encryption using ECB mode. No matching results were found for Rule 6, which involves the use of a constant seed for `secureRandom()`. This is primarily due to the limited number of signatures collected for Rule 6.

6.3.1. Comparison with LibScout. To demonstrate MtdScout's effectiveness in complementing the identification of missed false negatives in TPL detection, we conduct a comparison between MtdScout and a representative library detection tool based on the results obtained by MtdScout. Among the various library detection tools available [69], [55], [30], [53], [74], [72], LibScan [69] is a recent state-of-the-art tool; however, in our testing, it struggled to generate meaningful results in our dataset unless we set its similarity threshold as low as 0.4 (at which point the results become unstable). ATVHunter [72] is integrated into a non-open-source online service, making it unsuitable for use with our own library dataset. LibRadar [55], [17] relies on its own outdated library database that has not been updated in over five years. Additionally, LibExtractor [74] and LibD [53] employ clustering-based methods and do not rely on feature matching with a given set of libraries. Therefore, we have chosen LibScout [30], the most widely used open-source library detection tool, for our comparison. LibScout also provides a script [18] that allows us to generate library profiles ourselves.

To use LibScout for matching APKs with the input libraries, it is necessary to first generate library profiles using the libraries' Jar files. Out of the 44 libraries collected in §6.1, we were able to obtain 18 Jar files from libraries' Maven and corresponding GitHub websites. For the remaining 26 libraries, either the specific version of compiled Jar files was not released, or we failed to compile the specific version of the library source code. Therefore, we utilized LibScout to generate profiles for the 18 available libraries and conducted matching on the 23,962 apps using these profiles. It is worth noting that while MtdScout utilized all 44 libraries for the large-scale experiment, in this comparison, we only present the results related to the 18 libraries to ensure a fair comparison.

Unlike MtdScout, which outputs a set of method pairs, i.e., $R_{MS} = \{<m_{sig}, m_{dex}>\}$, LibScout's result consists of library-application pairs, i.e., $R_{LS} = \{<lib, apk>\}$. To make the results comparable, we transform the output of MtdScout into a set of library-application pairs as well. We represent the method signatures using the corresponding library names and indicate the application name where MtdScout identifies the target method clones. Note that MtdScout is designed for detecting method clones rather than third-party libraries. Therefore, to ensure a fair comparison, we also checked the method signatures of the 18 libraries and found that none of them share the same signature, ensuring that these signatures serve as unique identifiers for the libraries.

Comparison results. Out of the 18 libraries analyzed by both MtdScout and LibScout, only three of them were detected by LibScout in the tested 23,962 apps. In contrast, MtdScout detected method-level clones in eight libraries across the same set of apps, and the three libraries detected by LibScout are a subset of the eight libraries detected by MtdScout. In Table 2, we highlight these eight libraries and present the detection numbers of MtdScout and LibScout using bold fonts, with the number of LibScout listed in brackets. The result shows that LibScout only has comparable or higher detection numbers on two libraries, (i.e., `aws-sdk-java` and `apache/poi`). In the following paragraphs, we analyze in more detail from the perspective of library-application pairs to explain the differences in results between the two tools. More security findings are available in §6.4.

For the 18 libraries analyzed, MtdScout identified 2,291 $<lib, apk>$ pairs, while LibScout identified only 645 pairs. Among these, 139 pairs were detected by

both tools, accounting for only 6.1% of all MtdScout's results. This implies that there are 2,152 pairs detected exclusively by MtdScout. Given this substantial number, manually inspecting all these pairs is impractical. Hence, we randomly sampled 100 pairs to investigate why LibScout failed to detect them. Since each $<lib, apk>$ pair may have one or more combinations of signatures and target methods, for each $<lib, apk>$ pair, we randomly selected one $<m_{sig}, m_{dex}>$ pair for analysis. For each sampled method pair, we compare the package names of m_{dex} and m_{sig} to determine whether m_{dex} is an imported library method or a cloned method. If the package name of m_{dex} is obfuscated, we compare its hierarchy to identify the imported method. Through our analysis, we identified three reasons for the discrepancies:

- Among 32% of the sampled pairs, the vulnerable library methods indeed exist in the corresponding apps, but certain library class files within the host app package have been either removed or modified by Android's default app shrinking [10]. Further details on this observation are discussed in §6.4.
- In 63% of the sampled pairs, the methods are indeed cloned instead of being imported from the libraries, making the corresponding packages in the apps differ from the libraries. This implies that the APKs do not utilize the corresponding libraries, rendering these pairs undetectable by LibScout. A more comprehensive explanation of this finding is provided in §6.4.
- Only 5% of the sampled pairs are due to the false positives of MtdScout, which is consistent with the measurement in §6.2.

Out of the 506 pairs detected exclusively by LibScout, which accounts for approximately 78.4% of all the pairs detected by LibScout, we also sampled 50 pairs. For each pair, we identified the method signatures of the corresponding library and investigated why MtdScout did not identify them in the corresponding APK. In all the sampled pairs, it was observed that although LibScout detected the presence of the library in an app, the app itself did not contain the target vulnerable methods. This absence leads to MtdScout's inability to detect them. The reason LibScout can detect the library even in the absence of target methods is that other packages or class files contribute to the similarity score that exceeds LibScout's threshold. This discovery emphasizes that the detection of a library does not necessarily imply the existence of a specific vulnerable method. Further details on this observation are provided in §6.4.

> Answer for RQ2: MtdScout effectively complements LibScout by identifying its missed false negatives related to Android app shrinking and method-only cloning, with a sampled accuracy of 95%.

6.3.2. Comparison with CryptoGuard. In real-world scenarios, app markets often employ dataflow analysis tools like CryptoGuard [58] to scan their apps for crypto misuses. CryptoGuard is a dedicated dataflow analysis tool that utilizes state-of-the-art crypto-specific slicing techniques [58], [68]. As such, we evaluated the effectiveness and the running performance of MtdScout by comparing it with CryptoGuard on our large dataset of 23,962 APKs. Both MtdScout and CryptoGuard were executed in parallel using ten threads. Note that we configured CryptoGuard with the same set of six rules as MtdScout (outlined in §6.1) to ensure a fair comparison. Additionally, we set a timeout of 30 minutes for each APK when running CryptoGuard, as it may encounter infinite loops when encountering cycles in its dataflow graph. MtdScout, on the other hand, does not face this issue since our search trees are acyclic, as explained in §5.1.

Effectiveness results. Figure 5 provides a visual representation of the apps detected by CryptoGuard and MtdScout. In summary, CryptoGuard identified 23,761 insecure methods within 5,065 APKs, while MtdScout found 7,618 methods in 2,990 apps. CryptoGuard, which detects general crypto misuses in apps as opposed to specific insecure method clones, naturally reports a higher number of findings compared to MtdScout. The bars labeled *CG_apps* in the figure represent the insecure apps identified exclusively by CryptoGuard, amounting to around 149 apps per category. Meanwhile, MtdScout exclusively identified approximately 47 apps, as indicated by the bars labeled *MS_app*, *MS_App_(CG_timeout)*, and *MS_app_(CG_error)*. Unlike other taint analysis tools, CryptoGuard does not skip the analysis of third-party libraries. However, this comprehensive approach leads to timeouts for 8,086 apps (attributed to the expensive call graphs generated by FlowDroid [27], as discussed in [68]) and internal errors for 7,590 apps (due to the whole app-based analysis for call graphs). As a result, CryptoGuard could only process 34.6% (8,286) of the input apps.

MtdScout successfully scanned all the apps that CryptoGuard failed to analyze due to errors or timeouts, complementing the identification of insecure methods. Specifically, MtdScout supplemented the detection in 2,304 apps that CryptoGuard could not analyze (see *MS_App_(CG_timeout)* and *MS_app_(CG_error)*), identifying 5,897 insecure methods. It is worth noting that even if CryptoGuard had been able to process these apps, it might not have identified all the methods that MtdScout did. Furthermore, MtdScout also exclusively identified 262 insecure methods that CryptoGuard missed in 74 processed apps (see *MS_app*). This highlights the value of using MtdScout to complement the identification of insecure methods by taint analysis tools like CryptoGuard.

> Answer for RQ3: MtdScout effectively complements CryptoGuard by identifying a considerable number of false negatives, which are hardly avoided due to inherent timeouts and failures in the expensive taint analysis.

6.4. Security Findings and Case Study

Following a brief introduction to some of the findings in §6.3, this section gives more elaboration about the security findings obtained from the large-scale experiment.

Finding 1: Class files within the host app package could be either removed or modified by Android app shrinking, leading to a lower similarity score in package-level library detection. As mentioned in §6.3, among 32 of the 100 sampled library-application pairs, the vulnerable library methods do indeed exist in the corresponding apps and can thus be detected by MtdScout. However, some class files from the library within the host app package have been either removed or modified by Android's default app shrinking [10], leading to a lower

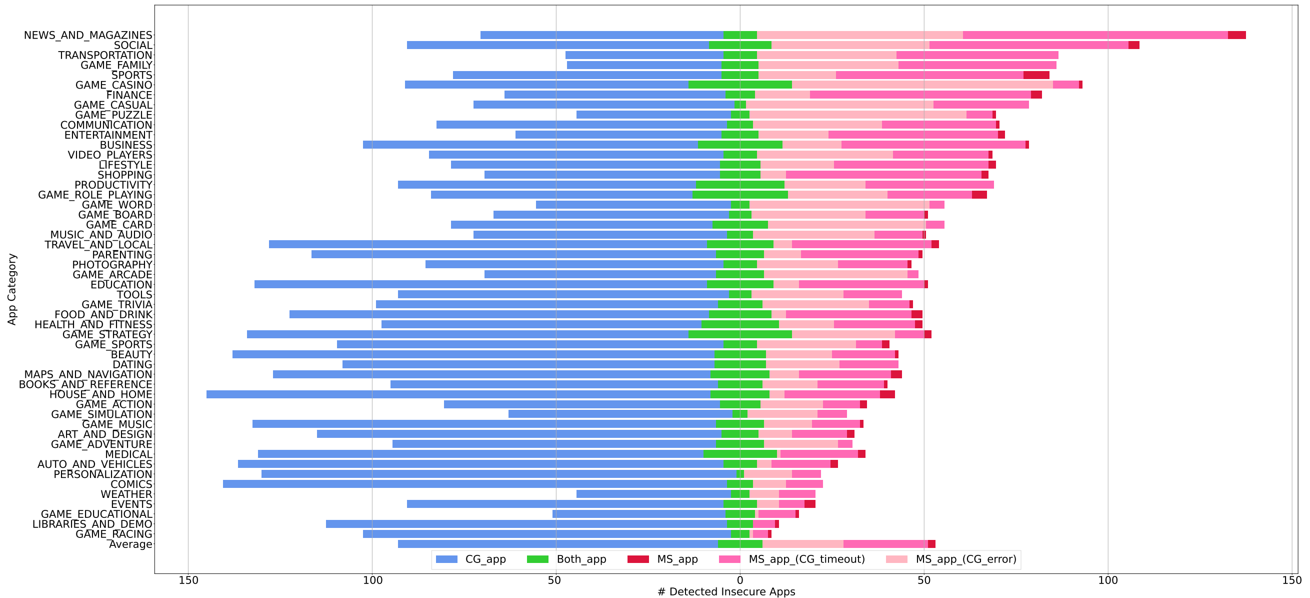


Figure 5: The number of detected apps in each category by CryptoGuard and MtdScout.

similarity score that falls below LibScout's threshold. For example, in the `br.com.intermedium` app, the `com.amazonaws.util` package originally contained 65 classes, but due to app shrinking, only 27 classes remain in the package, leading to the detection discrepancy.

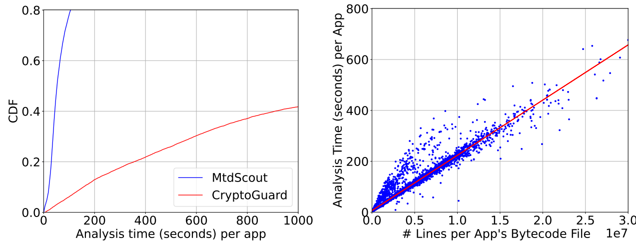
Finding 2: Many apps copy code from libraries instead of importing them, rendering package-level library detection ineffective. Among the 100 sampled pairs, we found that 63% of them contained copied code from another library. For instance, `com.amazonaws.auth` library copied a method from the `com.alibaba.nacos` library to generate a message authentication code (MAC) with a constant salt value. Indeed, 858 methods in the results implemented the same salt and MAC generation process. Further analysis revealed that about 77.6% (666) of these methods were copied code rather than imported code. These copied methods inherit the same vulnerabilities as the original code. However, package-level detection cannot identify copied methods solely by matching the original package, as an app may only copy specific methods without including other parts of the package. Moreover, while CryptoGuard can detect these methods, it cannot determine them as copied code.

Finding 3: The detection of a library in an app does not guarantee the presence of corresponding vulnerable methods in that app, rendering package-level library detection unreliable for identifying vulnerable library methods. As described in §6.3, after sampling 50 library-application pairs from the 506 pairs detected exclusively by LibScout, we found that none of the vulnerable methods existed in the corresponding applications, making them undetectable by MtdScout. This discrepancy can be attributed to optimization mechanisms [10] during compilation, where only the utilized components of a library are included in the APK file, while the rest are removed. For example, LibScout detects the library `aws/aws-sdk-java` in the app `com.Slack`, whereas MtdScout does not output the pair. Upon examining the app's bytecode and the library's Jar file, we discovered that the vulnerable method exists in the package `com.amazonaws.services.s3.crypto`. However, while the app contains the package `com.amazonaws.services`, the subpackage `s3.crypto` is absent. This demonstrates the inadequacy of relying on package-level library detection tools for identifying vulnerable library methods in apps.

Finding 4: Many single methods contain multiple crypto misuses, a finding that can not be revealed by package-level approaches. We found that 371 methods in our result can be matched with the signatures of two or more crypto misuse rules. Among these methods, 178 (48%) utilize ECB mode and a constant secret key for encryption (Rule 1 and 3), while 169 (45.6%) employ CBC mode with a static initial vector and a constant secret key (Rule 2 and 3). Among the remaining 25 methods, 24 of them perform password-based encryption with ECB mode and a constant secret key, with iterations below 1000 (Rule 1, 4 and 5). The presence of multiple crypto misuses within a single method exacerbates the insecurity of encryption. Notably, previous studies did not explore this scenario at the method level and did not reveal that a single method could contain multiple crypto misuses, as their focus was on package-level or app-level detection.

Case study. 2,624 apps (87.8% of the apps in the result) were found to contain methods that utilize constant keys for encryption, which may result in privacy breaches if attackers use the key to decrypt transmitted information. For example, the gaming app `com.tencent.godgame`, developed by Tencent, includes a method using a constant string as the symmetric encryption key. The app `com.xiaomi.hm.health`, developed by Xiaomi, utilizes a method that uses a constant salt when generating a secret key. An attacker can easily obtain this constant salt by decompiling the APK, and conduct a dictionary attack.

Ethics. We have responsibly reported the identified issues to the respective vendors for appropriate action. For example, we reported the issue in `com.tencent.godgame`, where constant keys were used for encryption, to the Tencent Security Response Center [20]. Similarly, we reported the issue in `com.xiaomi.hm.health`, where constant salts were used for secret key generation, to the Xiaomi Security Center [22].



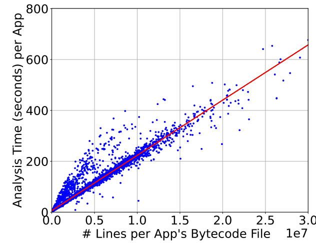


Figure 6: CDF plot of per-app running time of MtdScout and CryptoGuard.

Figure 7: The relation between MtdScout's analysis time and LOC.

> Answer for RQ4: Our analysis reveals four security findings that shed light on the root causes behind the greater effectiveness of MtdScout's method-level clone detection compared to package-level library detection.

6.5. Running Performance

We evaluate the running performance of MtdScout and demonstrate its efficacy by comparing it with CryptoGuard [58]. Overall, MtdScout requires only about 52 hours to complete the analysis of the entire dataset. This translates to an average running time of approximately 8.6 seconds per app when executed in parallel. On the other hand, CryptoGuard took around one month (specifically, 27 days) to analyze the same dataset, despite being configured with a 30-minute timeout and parallelized with ten threads. In contrast, MtdScout encountered failures for only 89 APKs, which account for less than 0.4% of the entire dataset. To ensure reliable performance comparison, we excluded the 7,590 APKs that encountered internal errors, as their time does not accurately represent the actual analysis duration.

Figure 6 presents the CDF (cumulative distribution function) plot of *per-app* running time for the 16,372 APKs analyzed by both tools. As shown in the blue curve, MtdScout completes the analysis for 80% of the apps within ~100 seconds. In contrast, the red curve shows that only 35.7% of the apps could be analyzed by CryptoGuard within 750 seconds, within the time frame of which MtdScout can analyze all the apps. Furthermore, at the 80% time point of MtdScout (105 seconds), only 6.6% of the APKs could be analyzed by CryptoGuard. In terms of median time, MtdScout performs at 52 seconds, while CryptoGuard takes significantly longer at 1,645 seconds. Thus, MtdScout is 31.6 times faster than CryptoGuard in terms of analysis speed.

We conducted additional measurements to evaluate the impact of bytecode size (lines of dexdump file) on the analysis time of MtdScout. The results, shown in Figure 7, indicate a nearly linear relationship between bytecode size and analysis time. This finding suggests that as the bytecode size increases, the running time of MtdScout does not exponentially grow. Additionally, MtdScout performs matching directly on the dexdump file, whereas traditional source code clone detection tools require decompilation of APKs into source code. Disassembling an APK into a dexdump file typically takes only a few seconds or less, whereas decompilation can take several minutes or longer. Therefore, MtdScout exhibits greater efficiency compared to source code clone detection tools.

> Answer for RQ5: MtdScout is quite fast, with the median time for analyzing each app 31.6 times faster than CryptoGuard. Moreover, the running time of MtdScout scales linearly with the size of the app's bytecode.

7. Discussion

Obfuscation and packers. Currently, MtdScout is resilient to identifier obfuscation (e.g., ProGuard) and certain code optimizations (e.g., function inlining). MtdScout can also adapt to Java reflection [21] because standard reflection APIs such as `Class.forName()` and `Method.invoke()` can be translated into invocation signatures, with their parameters recorded as constant strings representing the reflected names. However, MtdScout's performance may be affected by more complex obfuscation techniques, such as VM-based obfuscation [31], [13], [11] that modify the order of instructions or encrypt the DEX code. We also did not test MtdScout with advanced obfuscators like DexGuard [23]. Exploring solutions to counteract sophisticated obfuscation could be considered for future work. Additionally, MtdScout is unable to analyze packed APKs [42] due to the inability to access the original DEX code. Unpacking APKs is a separate problem that falls outside the scope of this work.

Lack of semantic information. During signature generation, MtdScout does not take into account the control flow graph (CFG) information when converting method invocation statements into bytecode instruction sequences. Consequently, if a vulnerability is patched with a minor instruction-level modification that does not change the overall instruction order, MtdScout may incorrectly identify the patched method as its vulnerable version, resulting in false positives. For example, if a patched method only modifies the trigger condition of a crucial branch, the patched version may be mistakenly flagged. In our future work, we aim to enhance the order-sensitive portion of our signatures by incorporating additional information to address such scenarios and improve accuracy.

8. Conclusion

In this paper, we introduced MtdScout, a cross-layer, method-level clone detection tool designed to detect insecure open-source method clones in Android apps at the bytecode level, aiming to address the inherent false negatives in existing TPL and taint analysis methods. MtdScout generates bytecode signatures for flawed source methods using compiler-style interpretation and abstraction and matches them efficiently with target app bytecode using layered searches over signature-mapped trees. Our quantitative experiment showed that MtdScout outperforms three tested clone detection tools, achieving a precision of 92.5% and a recall of 87.2%. A large-scale experiment with 23.9K Google Play apps further demonstrated MtdScout's ability to complement LibScout and CryptoGuard in identifying insecure methods and highlighted its superior efficiency compared to CryptoGuard. Additionally, our experiment revealed four security findings, highlighting the disparities between MtdScout's method-level clone detection and traditional package-level library detection.

References

[1] Disassemble Android DEX files. http://blog.vogella.com/2011/02/14/disassemble-android-dex/.

[2] CVE-2021-44228, Log4j2 vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44228, 2021.

[3] CVE-2021-44832: Apache Log4j2 are vulnerable to a remote code execution (RCE) attack. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44832, 2021.

[4] AndroidRank. https://www.androidrank.org/android-most-popular-google-play-apps, 2022.

[5] Antlr. https://www.antlr.org/, 2022.

[6] CodeQL. https://codeql.github.com/, 2022.

[7] F-droid. https://f-droid.org/, 2022.

[8] Java Obfuscator and Android App Optimizer — ProGuard. https://www.guardsquare.com/proguard, 2022.

[9] LGTM. https://lgtm.com//, 2022.

[10] Shrink, obfuscate, and optimize your app. https://developer.android.com/studio/build/shrink-code, 2022.

[11] Allatori. http://www.allatori.com/features/android-obfuscation.html, 2023.

[12] Dalvik executable instruction formats. https://source.android.com/docs/core/runtime/instruction-formats, 2023.

[13] DashO. https://www.preemptive.com/products/dasho, 2023.

[14] Dexdump on Ubuntu. https://manpages.ubuntu.com/manpages/xenial/man1/dexdump.1.html, 2023.

[15] GitHub. https://github.com/, 2023.

[16] GitHub Advisory Database. https://github.com/advisories, 2023.

[17] LibRadar on Github. https://github.com/pkumza/LibRadar, 2023.

[18] LibScout on Github. https://github.com/reddr/LibScout, 2023.

[19] Log4j affected vendor and software lists. https://github.com/cisagov/log4j-affected-db, 2023.

[20] Tencent Security Response Center. https://en.security.tencent.com/, 2023.

[21] Using Java Reflection. https://www.oracle.com/technical-resources/articles/java/javareflection.html, 2023.

[22] Xiaomi Security Center. https://sec.xiaomi.com/, 2023.

[23] DexGuard. https://www.guardsquare.com/dexguard, 2024.

[24] Junaid Akram, Zhendong Shi, Majid Mumtaz, and Ping Luo. DroidCC: A scalable clone detection approach for Android applications to detect similarity at source code level. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 100–105. IEEE, 2018.

[25] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: Collecting millions of Android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016.

[26] Steven Arzt and Eric Bodden. Stubdroid: automatic inference of precise data-flow summaries for the android framework. In *Proceedings of the 38th International Conference on Software Engineering*, pages 725–735, 2016.

[27] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM PLDI*, 2014.

[28] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *2015 IEEE/ACM 37th IEEE international conference on software engineering*, volume 1, pages 426–436. IEEE, 2015.

[29] Arutyun Avetisyan, Shamil Kurmangaleev, Sevak Sargsyan, Mariam Arutunian, and Andrey Belevantsev. Llvm-based code clone detection framework. In *2015 Computer Science and Information Technologies (CSIT)*. IEEE, 2015.

[30] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in Android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–367, 2016.

[31] Vivek Balachandran, Darell JJ Tan, Vrizlynn LL Thing, et al. Control flow obfuscation for android applications. *Computers & Security*, 61:72–93, 2016.

[32] Theodore Book, Adam Pridgen, and Dan S Wallach. Longitudinal analysis of Android ad library permissions. *arXiv preprint arXiv:1303.0857*, 2013.

[33] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186, 2014.

[34] Mengjie Chen, Xiao Yi, Daoyuan Wu, Jianliang Xu, Yingjiu Li, and Debin Gao. AGChain: A blockchain-based gateway for trustworthy app delegation from mobile app markets. *arXiv preprint arXiv:2101.06454*, 2021.

[35] Xiao Cheng, Zhiming Peng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. Mining revision histories to detect cross-language clones without intermediates. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 696–701, 2016.

[36] James R Cordy and Chanchal K Roy. The NiCad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220. IEEE, 2011.

[37] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In *Computer Security–ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings 17*, pages 37–54. Springer, 2012.

[38] Jonathan Crussell, Clint Gibler, and Hao Chen. Andarwin: Scalable detection of android application clones based on semantics. *IEEE Transactions on Mobile Computing*, 14(10):2007–2019, 2014.

[39] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *Proc. ACM CCS*, 2017.

[40] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. Automating patching of vulnerable open-source software versions in application binaries. In *NDSS*, 2019.

[41] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*, pages 2169–2185, 2017.

[42] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and XiaoFeng Wang. Things you may not know about android (un) packers: A systematic study based on whole-system emulation. In *NDSS*, 2018.

[43] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In *Proc. ACM CCS*, 2013.

[44] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *Proc. ACM CCS*, 2012.

[45] Johannes Feichtner, David Missmann, and Raphael Spreitzer. Automated binary analysis on ios: A case study on cryptographic misuse in ios applications. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 236–247, 2018.

[46] François Gagnon, Marc-Antoine Ferland, Marc-Antoine Fortier, Simon Desloges, Jonathan Ouellet, and Catherine Boileau. Androssl: A platform to test Android applications connection security. In *International Symposium on Foundations and Practice of Security*, pages 294–302, 2015.

[47] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112, 2012.

[48] Jiyong Jang, Abeer Agrawal, and David Brumley. ReDeBug: Finding unpatched code clones in entire OS distributions. In *Proc. IEEE Symposium on Security and Privacy*, 2012.

[49] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proc. ACM ICSE*, 2007.

[50] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291. IEEE, 2015.

[51] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in Android apps. In *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 403–414. IEEE, 2016.

[52] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260. IEEE, 2017.

[53] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. Libd: Scalable and precise third-party library detection in Android markets. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 335–346, 2017.

[54] Siqi Ma, David Lo, Teng Li, and Robert H Deng. Cdrep: Automatic repair of cryptographic misuses in Android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016.

[55] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: fast and accurate detection of third-party libraries in Android apps. In *Proceedings of the 38th international conference on software engineering companion*, pages 653–656, 2016.

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

[57] Luca Piccolboni, Giuseppe Di Guglielmo, Luca P Carloni, and Simha Sethumadhavan. Crylogger: Detecting crypto misuses dynamically. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[58] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng 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*, pages 2455–2472, 2019.

[59] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[60] Bahman Sistany. Androclonium: Bytecode-level code clone detection for obfuscated android apps. In *ICT Systems Security and Privacy Protection: 37th IFIP TC 11 International Conference, SEC 2022, Copenhagen, Denmark, June 13–15, 2022, Proceedings*, page 379. Springer Nature.

[61] David Sounthiraraj, Justin Sahs, Garrett Greenwood, Zhiqiang Lin, and Latifur Khan. SMV-Hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In *Proc. ISOC NDSS*, 2014.

[62] Chenning Tao, Qi Zhan, Xing Hu, and Xin Xia. C4: Contrastive cross-language code clone detection. 2022.

[63] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: A scalable and accurate two-phase approach to Android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 71–82, 2015.

[64] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. Ccaligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1066–1077, 2018.

[65] Xueqiang Wang, Kun Sun, Yuewu Wang, and Jiwu Jing. DeepDroid: Dynamically enforcing enterprise policy on Android devices. In *Proc. ISOC NDSS*, 2015.

[66] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1137–1150, 2018.

[67] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proc. ACM CCS*, 2014.

[68] Daoyuan Wu, Debin Gao, Robert H Deng, and Chang Rocky KC. When program analysis meets bytecode search: Targeted and efficient inter-procedural analysis of modern Android apps in BackDroid. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 543–554, 2021.

[69] Yafei Wu, Cong Sun, Dongrui Zeng, Gang Tan, Siqi Ma, and Peicheng Wang. LibScan: Towards more precise Third-Party library identification for android applications. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3385–3402, 2023.

[70] Zifan Xie, Ming Wen, Haoxiang Jia, Xiaochen Guo, Xiaotong Huang, Deqing Zou, and Hai Jin. Precise and efficient patch presence test for android applications against code obfuscation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 347–359, 2023.

[71] Xiao Yi, Yuzhou Fang, Daoyuan Wu, and Lingxiao Jiang. BlockScope: Detecting and investigating propagated vulnerabilities in forked blockchain projects. In *Proc. ISOC NDSS*, 2023.

[72] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1695–1707. IEEE, 2021.

[73] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 887–902, 2018.

[74] Zicheng Zhang, Wenrui Diao, Chengyu Hu, Shanqing Guo, Chaoshun Zuo, and Li Li. An empirical study of potentially malicious third-party libraries in Android apps. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 144–154, 2020.

TABLE 3: The accuracy measurement results of the open-source app subset.

	org.coolreader					com.wire					org.telegram.messenger					com.fsck.k9					org.connectbot				
	TP|FP	GT	Preci	Rec	F1	TP|FP	GT	Preci	Rec	F1	TP|FP	GT	Preci	Rec	F1	TP|FP	GT	Preci	Rec	F1	TP|FP	GT	Preci	Rec	F1
MtdScout	14|0	18	100%	77.8%	87.5%	0|0	1	-	0%	-	0|0	0	-	-	-	1|1	1	50%	100%	66.7%	0|1	0	0%	-	-
RDB 0.7	3|9	18	25.0%	16.7%	20%	0|0	1	-	0%	-	0|0	0	-	-	-	0|0	1	-	0%	-	0|1	0	0%	-	-
RDB 0.6	5|12	18	29.4%	27.8%	28.6%	0|0	1	-	0%	-	0|1	0	-	-	-	0|0	1	-	0%	-	0|4	0	0%	-	-
RDB 0.5	8|219	18	3.5%	44.4%	6.5%	1|22	1	4.3%	100%	8.3%	0|35	0	0%	-	-	0|25	1	0%	0%	-	0|156	0	0%	-	-
RDB 0.4	16|1,460	18	1.1%	88.9%	2.1%	1|305	1	0.3%	100%	0.7%	0|439	0	0%	-	-	1|280	1	0.4%	100%	-	0|1,050	0	0%	-	-
SCC 0.7	4|0	18	100%	22.2%	36.4%	0|0	1	-	0%	-	0|0	0	-	-	-	0|0	1	-	0%	-	0|0	0	-	-	-
SCC 0.6	4|6	18	40%	22.2%	28.6%	1|0	1	100%	100%	100%	0|0	0	-	-	-	0|0	1	-	0%	-	0|0	0	-	-	-
SCC 0.5	4|48	18	7.7%	22.2%	11.4%	1|0	1	100%	100%	100%	0|1	0	0%	-	-	0|0	1	-	0%	-	0|2	0	0%	-	-
SCC 0.4	4|162	18	2.4%	22.2%	4.3%	1|45	1	2.2%	100%	4.3%	0|10	0	0%	-	-	0|14	1	0%	0%	-	0|4	0	0%	-	-

	com.ichi2.anki					de.schildbach.wallet					fr.gouv.android.stopcovid					net.nurik.roman.muzei					org.xbmc.kore				
	TP|FP	GT	Preci	Rec	F1	TP|FP	GT	Preci	Rec	F1	TP|FP	GT	Preci	Rec	F1	TP|FP	GT	Preci	Rec	F1	TP|FP	GT	Preci	Rec	F1
MtdScout	12|0	12	100%	100%	100%	1|0	2	100%	50%	66.7%	0|0	0	-	-	-	0|1	0	0%	-	-	0|0	0	-	-	-
RDB 0.7	0|0	12	-	0%	-	1|1	2	50%	50%	50%	0|0	0	-	-	-	0|0	0	-	-	-	0|0	0	-	-	-
RDB 0.6	0|4	12	0%	0%	-	1|6	2	14.3%	50%	22.2%	0|0	0	-	-	-	0|8	0	0%	-	-	0|0	0	-	-	-
RDB 0.5	4|109	12	3.5%	33.3%	6.4%	1|148	2	0.7%	50%	1.3%	0|0	0	-	-	-	0|123	0	0%	-	-	0|16	0	0%	-	-
RDB 0.4	8|686	12	1.2%	66.7%	2.3%	1|926	2	0.1%	50%	0.2%	0|17	0	0%	-	-	0|847	0	0%	-	-	0|382	0	0%	-	-
SCC 0.7	0|0	12	-	0%	-	0|0	2	-	0%	-	0|0	0	-	-	-	0|0	0	-	-	-	0|0	0	-	-	-
SCC 0.6	0|0	12	-	0%	-	2|0	2	100%	100%	100%	0|0	0	-	-	-	0|0	0	-	-	-	0|0	0	-	-	-
SCC 0.5	0|1	12	0%	0%	-	2|13	2	13.3%	100%	23.5%	0|0	0	-	-	-	0|0	0	-	-	-	0|0	0	-	-	-
SCC 0.4	0|8	12	0%	0%	-	2|93	2	2.1%	100%	4.1%	0|117	0	0%	-	-	0|11	0	0%	-	-	0|8	0	0%	-	-

Appendix

In §6.2, we also tried to find open-source apps from the large-scale app set mentioned in §6.3. Specifically, we searched the package names of all the apps on F-Droid [7] and GitHub [15] to determine whether they are open-sourced, which allows us to identify 71 open-source apps. However, some apps may not contain the vulnerable library methods we are interested in. We thus conduct a pre-search (by searching both the source code and dexdump files to count the number of crypto APIs used in these 71 apps and choose the top ten apps with the most crypto APIs as our open-source app subset, since they have a higher chance to contain vulnerable library methods. The detailed package names of these ten apps are shown in the first row of Table 3. The blue color indicates the highest number of each metric of an app, and the green color of Table 3 means this tool outputs no result when the ground truth is zero.

In the open-source app subset, only two apps, `org.coolreader` and `com.wire`, have been affected by multiple clones of vulnerable library methods. Other apps contain only less than two pairs of ground truth or even zero. This helps us understand how MtdScout would perform if an app does not contain any clone. The lack of ground truth in this subset is another reason why we need one more subset of apps for evaluation. In the close-source app subset, the average number of ground truth is 15, which can better evaluate the accuracy of MtdScout and make a fairer comparison with other tools.

The result of open-source app subset. According to Table 3, SourcererCC has a high precision at a high threshold yet with a very low recall, while our adapted ReDeBug has a high recall at a low threshold yet with thousands of false positives. Neither of them achieves a good F1 score, except for `com.wire` and `de.schildbach.wallet` which have only one or two ground truths. This indicates that under our problem scenario, both tools cannot find a good balance between precision and recall, which makes it hard to choose a suitable threshold. In contrast, MtdScout achieves the highest recall in `com.ichi2.anki` and the second-highest recall in `org.coolreader`, are the two apps with the most ground truth. On both apps, MtdScout also has the highest precision and F1 score. Moreover, MtdScout also does not output any false positive for the three apps with no ground truth, as shown in the green color of MtdScout row in Table 3. For other apps with zero or few vulnerable clones, the difference between the true positives outputted by MtdScout and the ground truth is less than one.

Obfuscation Resiliency. To check the obfuscation resiliency of MtdScout, we use the open-source app `com.ichi2.anki` and compile it into an APK file with the obfuscation by R8 compiler [10]. Specifically, we set the rule that all identifiers will be obfuscated, and the R8 compiler generates a table that maps the obfuscated identifiers to their original names. Based on this one-to-one relationship, we can recover the obfuscated matched methods outputted by MtdScout and determine whether it achieves the same accuracy.

We found that MtdScout still outputs 12 clone pairs. After recovering them to the original method names, the result pairs are the same as the results generated by the non-obfuscated version. This confirms that MtdScout is resilient to identifier obfuscation, which mainly benefits from non-obfuscated key statements used in our signature generation.