From base cases to backdoors: An Empirical Study of Unnatural Crypto-API Misuse

Victor Olaiya
William & Mary

Adwait Nadkarni
William & Mary

Abstract

Tools focused on cryptographic API misuse often detect the most basic expressions of the vulnerable use, and are unable to detect non-trivial variants. The question of whether tools should be designed to detect such variants can only be answered if we know *how* developers use and misuse cryptographic APIs in the wild, and in particular, *what the unnatural usage of such APIs looks like*. This paper presents the first large-scale study that characterizes unnatural crypto-API usage through a qualitative analysis of 5,704 representative API invocations. We develop an intuitive complexity metric to stratify 140,431 crypto-API invocations obtained from 20,508 Android applications, allowing us to sample 5,704 invocations that are representative of all strata, with each stratum consisting of invocations with similar complexity/naturalness. We qualitatively analyze the 5,704 sampled invocations using manual reverse engineering, through an in-depth investigation that involves the development of minimal examples and exploration of native code. Our study results in two detailed taxonomies of unnatural crypto-API misuse, along with 17 key findings that show the presence of highly unusual misuse, evasive code, and the inability of popular tools to reason about even mildly unconventional usage. Our findings lead to four key takeaways that inform future work focused on detecting unnatural crypto-API misuse.

1 Introduction

Cryptography plays a critical role in helping developers protect sensitive data. However, developers have a history of misusing crypto-APIs [6, 14, 15, 41], such as by using weak ciphers or foregoing necessary certificate verification, thereby putting user data at risk. In response, the security community has developed tools such as CryptoGuard [47] and CogniCrypt [28] to detect such crypto-API misuse. Recent work by Ami et al. shows that these popular crypto-API misuse detectors, or *crypto-detectors* in short, are highly ineffective at detecting even the misuse they claim to detect [6]. Particularly, while these tools can detect the simplest expression of a misuse, they often struggle to detect non-trivial *variants* of the same. For example, consider the instantiation of the vulnerable `des` cipher using the `Cipher.getInstance(<parameter>)` API of the Java Cryptography Architecture (JCA). The most basic instance would essentially be `Cipher.getInstance("DES")`, which most tools would detect. However, Ami et al. showed that tools may not detect even a simple variant of the same vulnerable use that specifies the argument in lowercase, i.e., as "`des`", let alone a more complex variant such as in Listing 1.

```
1 Cipher.getInstance("DE$S".replace("$", ""));
```

Listing 1: An unusual use of the `DES` cipher.

Tool designers have argued that such expression of crypto-APIs, as in Listing 1, is *unnatural*, i.e., overly complex and unlikely to be present in real software, and hence, not something tools should be expected to detect [6]. However, a counter-argument can also be made, that such perceived unnaturalness may exist in code in the wild, and hence, tools should look beyond typically conventional usage. This contention exposes a fundamental knowledge gap in the design space of crypto-API vulnerability detection: *we do not know enough about the* ***unnatural, unconventional expression*** *of crypto-API in the wild*, both in terms of the characteristics of such expression, or its prevalence in software. Addressing this gap through systematic study of real-world API usage would not only help address this contention, but also motivate the development of targeted techniques to find highly unnatural but relevant misuse. Therefore, this paper is motivated by a simple but critical ***research question***: *what are the odd, unconventional, ways in which developers (mis)use crypto APIs in the wild?*

Contributions: This paper is the first to develop a qualitative characterization of the unconventional use and misuse of crypto-API in the wild, through a study of 5,704 invocations. We study the use of two broad classes of API invocations defined in prior work [6], (1) *restrictive*, which only takes a certain predefined set of parameters, and (2)

flexible, which can be implemented in an unrestricted number of ways. We study a large number of invocations of one highly relevant and widely-studied API from each category, namely `Cipher.getInstance(<parameter>)` for restrictive, and `checkServerTrusted` for flexible, enabling tractable but in-depth characterization of usage. We also experimentally demonstrate generalizability to other APIs (Section 7). For an effective study grounded in the study of a rich set of crypto-API invocations, we target a security-critical app population that also makes heavy use of crypto-APIs. That is, we study crypto-API invocations in Android apps, and more precisely, a relevant sub-population, *mobile-IoT apps*, i.e., Android apps that serve as companion apps for IoT devices, or represent third-party integrations and automation services. This choice enables an impactful study, as (1) this population of apps is relatively likely to require crypto-APIs to protect sensitive IoT data at rest or in transit, (2) the attack surface mobile-IoT apps expose as the user interfaces (UIs) and controllers of physical IoT systems is critical, and (3) because any findings in these apps are directly relevant to a large body of prior work on crypto-API misuse [20, 28, 29, 32, 36, 44, 47] that has focused on the Java Cryptographic Architecture (JCA). Thus, this study is initialized with 140,431 crypto-API invocations (118,749 restrictive and 21,682 flexible) extracted from 20,508 apps from the mobile-IoT app dataset by Jin et al. [27], and makes the following contributions:

- ***The study.*** This is the first large scale qualitative study (n = 5,704 API invocations) that characterizes the unnatural use of crypto-API in the wild.
- ***Complexity-based stratification for qualitative analysis.*** We develop a *complexity metric* that allows us to obtain a representative sample from the broad population of 140,431 API invocations, using complexity as a proxy for their perceived naturalness. This approach allowed us to obtain a sample of 5,704 invocations (3,599 restrictive and 2,105 flexible) representative of all complexity strata.
- ***Qualitative Analysis and Taxonomies.*** We qualitatively analyzed the 5,704 API invocations, leveraging systematic manual reverse engineering, along with the use of tools such as Ghidra [39] for investigating calls into native code, and minimal working examples to explore parameter encryption and encoding. This extensive analysis led to the identification of 96 unique ways in which developers express and misuse crypto-APIs, organized into *two taxonomies* for the restrictive and flexible category, respectively.
- ***Results and* 17 *key findings* ($\mathcal{F}_1$–$\mathcal{F}_{17}$).** We find that even seemingly conventional API invocations may not always be simple/natural, but instead involve the use of object identifiers (OIDs) and key wrapping algorithms ($\mathcal{F}_2$), encryption ($\mathcal{F}_8$), and unsafe hard-coded defaults ($\mathcal{F}_6$), and ineffective measures instead of security checks ($\mathcal{F}_{12}$–$\mathcal{F}_{14}$). We find several cases of ***developers evading detection tools and manual code reviews***, e.g., appending arguments or manipulating strings in non-intuitive ways ($\mathcal{F}_7$, $\mathcal{F}_{10}$), hiding vulnerable parameters in native code ($\mathcal{F}_3$), and presenting best-practice parameters in code (e.g., `AES` in `GCM` mode) while actually using vulnerable parameters (e.g., `ECB`) ($\mathcal{F}_8$). Finally, we show that popular crypto-API misuse detectors cannot reason about even the most basic cases from our taxonomies ($\mathcal{F}_{17}$), i.e., of the 23 basic cases representing 59k invocations tested against, no tool can reason about 12/23 (representing 12.8k invocations).

To summarize, this paper characterizes the existence, diversity, and prevalence of unconventional, real-world, crypto-API invocations, and demonstrates the ineffectiveness of existing tools in reasoning about them. In doing so, it motivates the need to re-think tool design with a focus on detecting such prevalent and hard-to-find cases. With this objective, we distill our 17 findings into discussion themes that lead to *four key takeaways*, informing future work on the challenges and opportunities in detecting unnatural crypto-API misuse, enabled by the lessons and artifacts from this work.

2 Methodology

Our study is focused on two broad categories of crypto-API identified by prior work [6]: (1) *restrictive* invocations, i.e., where the acceptable parameter values can only fall within a predefined set of constants, and (2) *flexible* invocations, where the APIs are heavily customizable, and can be implemented in any unrestricted number of ways. For our analysis, we choose two APIs that are the canonical representations of the two key usage categories: the `Cipher.getInstance(<parameter>)` API from the restrictive category, and the `checkServerTrusted` method in the `TrustManager` interface for the flexible category. These APIs are some of the most represented in literature, and prior work has extensively studied [6, 12, 15, 21, 27, 38, 45, 49] and designed tools to detect their misuse [14, 15, 29, 31, 36, 37, 44, 47, 61]. Thus, focusing on these two relevant APIs allows for a tractable but impactful analysis.

2.1 Extracting invocations, Generating ASTs

As discussed previously in Section 1, we target mobile-IoT apps due to their critical attack surface, and their potential of using crypto-APIs to protect sensitive IoT data. We were able to successfully download the latest versions of 20,508 mobile-IoT apps from the list of 37k previously compiled by Jin et al. [27], as the rest were unavailable.

We decompile the apps using jadx [24],[1] search the decompiled code for the signatures of the two chosen APIs using regular expressions, and extract the class files with relevant

[1] We experimentally confirmed that decompiled code from jadx has high fidelity with respect to the source (see the online appendix [9]). To summarize, we compiled minimal examples of both natural and unnatural invocations observed in Sections 3 and 4, using R8 and Proguard obfuscation. Decompilation resulted in code with the same characteristics as the source.

instances. We then use the tree-sitter library [5] (particularly its *S-expressions*) to generate the Abstract Syntax Tree (AST) for the entire class. Since tree-sitter generates concrete syntax trees that include tokens such as *if* in if statements, *commas*, and *parentheses* as nodes, we traverse the tree to find and keep *named nodes* while removing anonymous nodes.

We traverse and filter the AST generated for the class using the relevant method signature of the relevant crypto-API. We then extract the selected nodes with the corresponding children, and in effect, obtain the AST for each crypto-API invocation. This approach led to a dataset of 140,431k ASTs representing crypto-API invocations in the 20k apps.

2.2 Extracting a Representative Sample of Invocations using a Complexity Metric

For a feasible qualitative analysis, we need a representative sample from the large dataset of all 140k invocations of crypto-API invocations. Given that our goal is to characterize unnaturalness in crypto-API invocation, we must obtain a sample in which all the unique populations/strata of natural or unnatural invocations are well represented. That is, we need a way to stratify the dataset to precisely group similarly natural/unnatural invocations and sample representative examples from each group. This requirement motivates our metric.

We define ***"natural"*** as the typical/conventional invocation of crypto-API according to the documentation and what tool designers expect, e.g., something as *simple* as `Cipher.getInstance("DES")`. Conversely, ***"unnatural"*** indicates deviation from conventional use, which, in extremes, can be similar to the *complex* code in Listing 1. Hence, it is intuitive to rely on the complexity of the invocation, measured by argument size, for stratifying the dataset along the spectrum of unnaturalness. This does not mean that more complex invocations are misuse, but instead, that *invocations of a similar complexity may be similarly (un)natural, and grouping them allows effectively sampling, study, and characterization*.

Construction of the complexity metric: We use the *S-expression* to query the AST of each crypto-API invocation and return all arguments in the argument list as seen in the source code. This allows us to identify the use of strings, identifiers, method calls, the ternary operator, string methods, *etc.*, in the API invocation.

For simplicity, we design the complexity metric based on number of arguments observed in the children of restrictive invocation nodes, i.e., cipher object instantiation. Similarly, for flexible invocations, we use the size of the method body based on number of element in the block. Equation 1 shows how we calculate complexity score of a method i, S_i, where d_i represents the number of arguments in the method:

$$S_i = \tanh[\log(d_i)] \quad (1)$$

The computation of the metric in Equation 1 can be explained as follows: Since the number of the arguments in a method, i.e., d_i, can not be predetermined, we use the logarithmic function to normalize the argument size to reduce the disparity between the final scores. The second step is to use the hyperbolic tangent function, i.e., tanh to calculate each complexity score between the ranges of -1 for least complex use to $+1$ to most complex use, since hyperbolic tangent function has a range of -1 to $+1$. That is, the tanh function enables a gradual distribution of the complexity score as the argument size increases, resulting in precise stratification.

2.3 Qualitative Analyses

We sample both restrictive and flexible invocations using the metric described previously for qualitative analysis. For each invocation being analyzed, we manually examine the invocation as well as it's surrounding code context in the application. In many cases, we came across method invocations that required further analysis and creating minimal working examples to test our hypotheses. Particularly, we frequently observed native method invocation during our qualitative analysis, which we reverse engineered using Ghidra [39] to identify the cipher algorithms and other arguments in native code. Finally, some invocations were more challenging to understand as they involved handcrafted transformations, including encoding and/or encryption. To understand these cases, we used our intuition to develop hypotheses regarding the eventual effect of the transformations, and tested the hypotheses by manually constructing minimal working examples.

2.4 Prevalence Analysis

Our prevalence analysis seeks to obtain a lower bound on the presence of each of the unique unnatural crypto-API invocations found using qualitative analysis across the entire dataset. We do not seek to find all instances of the said usage. Instead, our approach only marks the ones that it is most confident about, giving us a conservative estimate. We used a combination of signatures from argument types retrieved from the AST after parsing and signatures from specific method calls to measure prevalence. For example, the signature: `{'identifier': 3, 'ternary_expression': 1}` indicates the use of three identifiers with the ternary operator, resulting in the identification of `Cipher.getInstance(z2 ? CBC_PADDING : CBC_NOPADDING)` and similar usage.

3 Analysis of Restrictive Invocations

We extracted 118,749 invocations of the restrictive API chosen for this study, i.e., `Cipher.getInstance`. Applying the complexity metric to these invocations resulted in 55 distinct complexity scores. The distribution of the invocations across the complexity scores is shown in Figure 1.

As seen in Figure 1, 79,671 (67%) attain a score of "0" (i.e., henceforth noted as `[SCORE = 0]`). At

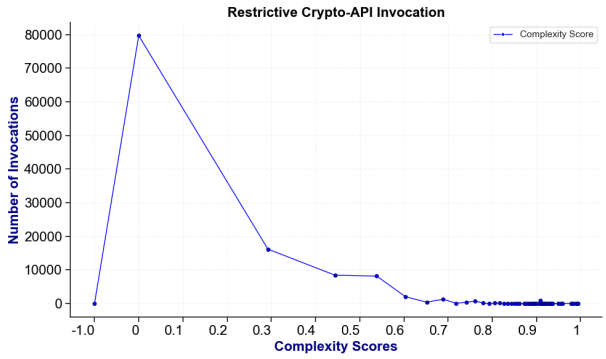


Figure 1: Restrictive Invocation Complexity Score Distribution

face value, this may be interpreted as showing that developers mostly use `Cipher.getInstance` in the most conventional way by specifying a single string, e.g., `Cipher.getInstance("AES/GCM/NoPadding")`. However, our qualitative analysis (Section 3.2) demonstrates otherwise, that even seemingly natural invocations may be quite unnatural.

3.1 Obtaining a Representative, Stratified Sample of Restrictive Invocations

For a feasible but in-depth qualitative analysis, we use stratified random sampling to obtain a representative sample of invocations to analyze for each of the 55 distinct complexity scores. We treat score as an independent population, i.e., a stratum. For each stratum, we calculate the number of samples we need to study for results representative of the entire stratum using the Qualtrics sample size calculator [46], assuming a 5% error margin and 95% confidence, both standard values. For scores with a limited population (e.g., `[SCORE = -1]`), we sampled the entire strata, whereas for others that were most populated, we obtained a relatively small but representative sample (e.g., `[SCORE = 0]`). Our online appendix [9] shows the size of the representative samples for each stratum. Using this approach, we selected a total of 3,599 representative invocations of `Cipher.getInstance` for qualitative analysis.

3.2 Results and Findings from the Qualitative Analysis of Restrictive Invocations

Our analysis of the 3,599 representative invocations allowed us to characterize the unique ways in which developers express restrictive crypto-APIs. We represent this characterization in a *taxonomy of unnatural restrictive crypto-API use*, as shown in Figure 2. The taxonomy shows 15 *basic* ways in which developers express the `Cipher.getInstance` API, and 29 *complex* ways in which they further combine the basic ways, resulting in 44 total taxonomy labels. Table 2 in the Appendix describes the 15 basic labels.

The rest of this section describes the results of our qualitative analysis, wherein we discuss individual scores or score ranges with key results using salient examples.

[SCORE = -1] – No arguments provided: Since the metric was designed using a tanh function which has a range of -1 to $+1$, we get a `[SCORE = -1]` for `Cipher.getInstance` with no arguments provided. There were 16 invocations of this type, expressed as ***type***`Cipher.getInstance()`, where the ***type*** can take the following values: "Block", "AES", and "XML", each value resulting in a unique default cipher configuration.

To elaborate, ***Block***`Cipher.getInstance()` defaults to AES in GCM mode. Similarly, using ***AES***`Cipher.getInstance()` defaults to AES in CBC mode i.e., `AES/CBC/PKCS5Padding`, which is vulnerable in certain situations [3].

Finding 1 ($\mathcal{F}_1$) – When instantiated without arguments, the outcome of `Cipher.getInstance` depends on the *type*, which may lead to vulnerable defaults.

[SCORE = 0] – String Literals, and Identifiers: Using string literals such as `"AES/GCM/NoPadding"` as the parameter is a natural and expected use of API such as `Cipher.getInstance`. Indeed, our prevalence analysis shows $60,060/118,749$ (or 51.41%) invocations of this kind. However, we observe that developers may use non-intuitive literals, potentially leading to undetected misuse.

We observed several cases where developers used object identifiers (OIDs) corresponding to common ciphers, particularly the use of `"1.2.840.113549.3.2"`, which resolves to the vulnerable RC2 cipher (see RFC 2268 [22]). Further, we observed the use of the AES Key Wrap Algorithm without padding (*"AESWrap"*) and with Padding (*"AESKWP"*), both of which default to ECB mode [23] (e.g., in default providers for Java and Android such as SUNJCE [25]), which is vulnerable. Our prevalence analysis establishes a lower bound of 174 uses of OIDs, and 1,487 of AESWrap.

Finding 2 ($\mathcal{F}_2$) – Restrictive invocations with String literals as parameters may seem natural from a syntactic standpoint but use non-intuitive abstractions such as OIDs and key wrapping algorithms with vulnerable defaults. Such semantic unnaturalness may escape detection tools.

After string literals, the use of identifiers is another natural way in which developers specify parameters. Since identifiers can be assigned to any variable and method, we perform additional analysis to characterize their use. We found several unique and non-trivial ways in which developers use identifiers, each of which occupy separate spots in the taxonomy. First, we found 4 cases where a vulnerable block chaining mode was used, i.e., `ECB` in Android API levels < 23, and a secure mode, i.e., `GCM` for higher APIs. The arbitrary cut-off at API 23 is odd as `GCM` has been supported since API 10 [8].

We also found invocations where the identifier holds the result of arbitrary string manipulation, e.g.,

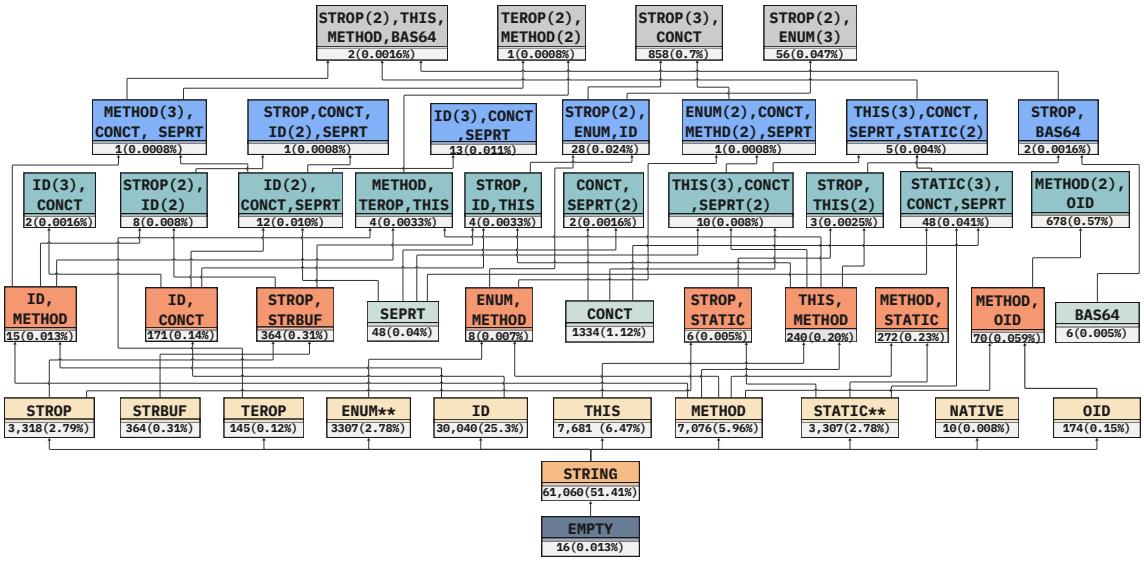


Figure 2: Taxonomy of Unnatural Restrictive Crypto-API Invocations with Prevalence information.

`Cipher.getInstance(<identifier>)`, where `<identifier>` = `String.format("%s/%s/%s","AES","CBC","PKCS7Padding")`, vulnerable to oracle padding attacks [3]. Our prevalence analysis found 44 instances of this type. Further, developers may also assign identifiers to methods that perform complex operations such as `XOR` and `Base64` encoding/decoding on cipher arguments. This may be prevalent in the wild as similar code was found on stackoverflow [54] as early as 2011. Instances of such `Base64` decoding and/or `XOR` use can be seen in Listing 9 (also Listing 33 and Listing 39 in the Appendix). Our prevalence analysis confirmed 6 instances of such complex operations, although this is a conservative estimate as detecting such use is a challenging.

```
1 private static native String nativeGetString(int i);
2 nativeGetString = nativeGetString(1);
3 Cipher.getInstance(nativeGetString);
```

Listing 2: Native Method Call with Identifier 1.

```
1 private final native String getCypherTransformation();
2 f6401b = crypto.getCypherTransformation();
3 Cipher.getInstance(f6401b)
```

Listing 3: Native Method Call with Identifier 2.

Finally, developers frequently assigned identifiers to native method calls, which may conceal misuse. The examples shown in Listing 2 and Listing 3 demonstrate the common patterns of native method calls to retrieve string arguments. There is no performance-based rationale for such calls to native methods that merely retrieve string parameter values to use in a the Java crypto API, as the JNI call results in performance degradation. Our prevalence analysis identified 10 such cases, all of which were misuse, including 5 uses of `AES/ECB/NoPadding`, 1 use of `AES` which defaults to `ECB` mode in JCA, and 4 uses of `AES/CBC/PKCS5Padding`.

Finding 3 ($\mathcal{F}_3$) – While identifier use in place of a string literal in restrictive crypto API may seem natural, we observe several types of non-intuitive usage, such as string manipulation or complex computation, most of which lead to misuse. Importantly, we observe JNI calls to obtain vulnerable parameters from native code to be eventually used in Java API, showing signs of *evasive* API misuse.

[SCORE = 0.4439] – Method calls, enum types, constructors: This score is characterized by the use of enum type, constructors, method calls and "this" keyword to reference the current object in a method. Note that the method calls here are classified distinctly as they were found in `Cipher.getInstance` invocations, unlike the prior discussion in `[SCORE = 0]` of method calls used to initialize identifiers used in the invocations. Out of the 368 invocations analyzed, we were able to identify the precise cipher used in 21 instances as we stopped after two levels of indirection, or if the value was retrieved from the network. 20/21 of these parameters were vulnerable (e.g., using `DES`). Our prevalence analysis showed that referencing the current object using the "`this`" keyword accounts for 6.47% or 7,681 instances.

Finding 4 ($\mathcal{F}_4$) – The use of enum types, method calls, and constructors leads to complex call chains that are hard to resolve due to severe indirection and network-derived values. 20/21 of the parameters we could resolve were vulnerable, highlighting the need to analyze such cases.

Among the method calls we analyzed, one in particular led to a ternary operator that used an arbitrary integer value to switch between the `AES` in `GCM` mode, and the default

(which results in ECB mode), as shown in Listing 4 below.

```
Cipher.getInstance(getTransformation())
----------------------------------------
private static String getTransformation() {
 return f882fB >= 2 ? "AES/GCM/NoPadding" : "AES";
```

Listing 4: Use of Ternary Operator via Method Call.

Finding 5 ($\mathcal{F}_5$) **–** Developers may use enum type constant and ternary operator to store cipher algorithm and switch between them based on arbitrary conditions such as constant values. However, the switch between secure and vulnerable parameters indicates that developers recognize the difference, but misuse on purpose.

[SCORE = 0.5385] – Ternary Operator, String Buffer, and Native Method Invocations: While we have seen the *indirect* use of ternary operator via method calls in previous score, this score is characterized by the *direct* use of the ternary operator, native method calls, and string buffers as arguments for cipher object instantiation.

```
Cipher.getInstance(requestTransform(1));
public static native String requestTransform(int i);
```

Listing 5: Direct Native Method Call.

To elaborate, we found direct calls from `Cipher.getInstance` instantiations to native methods that retrieve cipher transformation strings and values. For example, the native method *requestTransform* from Listing 5 is defined as follows: if the value of the parameter $i \neq 0$, it returns `AES/ECB/NoPadding`, otherwise, it returns `AES/CBC/PKCS5Padding`. However, as seen in Listing 5, i is hardcoded to 1, which defaults to `ECB`. We found 5 such cases of misuse in 5 different apps.

Finding 6 ($\mathcal{F}_6$) **–** Developers use conditions based on integers to switch between a vulnerable and good parameters. However, the integer value may be hardcoded, making the vulnerable parameter the default.

Further, we found several direct uses of the ternary operator in a cipher object instantiation. Particularly, we observed instances where the operator was dependent on arbitrary integer values (often from the network), as seen before in Listing 4.

```
StringBuffer stringBuffer = new StringBuffer();
stringBuffer.append("DESede/CBC/");
stringBuffer.append("NoPadding");
this.f13712c =
 Cipher.getInstance(stringBuffer.toString());
```

Listing 6: Use of StringBuffer.

Finally, we find the string buffer being used to append fragments of the cipher algorithm. For example Listing 6 shows a code fragment where the developers split the cipher algorithm and cipher mode separate from the padding mode. This is a relatively simple instance of this type of invocation expressed in a string buffer, as the next score shows more complex cases that involve non-intuitive splitting and combining of values. However, such simple use of string buffers may prove challenging for tools to detect as the manner in which developers may split the algorithm may be non-trivial. Our prevalence analysis found 364 such instances.

[SCORE = 0.6 – 0.69] – Disguising Vulnerable Cipher choices via String Builders, Encoding, Encryption: The general structure of our observed invocations in this score range is similar to prior scores involving ternary operators, method calls, and the use of string builders/buffers. However, their use and parametrization is quite different as the parameter string is irregularly fragmented.

```
StringBuilder sb = new StringBuilder();
sb.append("AES");
sb.append("/EC");
sb.append("B/PKCS7P");
sb.append("adding");
Provider provider = Security.getProvider("BC");
cipher = Cipher.getInstance(sb.toString(), provider);
```

Listing 7: Use of String Builder.

Consider the example in Listing 7, which demonstrates the fragmentation of `AES/ECB/PKCS7Padding` into odd fragments such as `/EC` and `B/PKCS7P`. This type of irregular fragmentation may not only present a challenge to misuse detectors, but may also present additional evidence of developers trying to evade tools and/or code reviews by expressing their choice of vulnerable parameters in non-intuitive ways. Our prevalence analysis found 364 such uses.

Finding 7 ($\mathcal{F}_7$) **–** String operations techniques such as use of `StringBuffer` and `StringBuilder` may be used to append irregular fragments of algorithm name, mode and padding. Developers may use this unconventional design to *evade* detection tools.

We found further complex uses that leveraged encryption, encoding, and XORs to conceal vulnerable algorithm values (e.g., transforming `CBC` to `ECB` at runtime using XOR, see Listing 39 in the Appendix). A salient example is in Listing 8 below, where the secure algorithm appears in code (i.e., `AesGCMUtils`), but a hidden (encrypted), vulnerable, value is eventually used:

```
algo_encode =
 "32Bi2A5oaH61xilScou92x9faAiO0SOBXmb0X/wqAijapt8K";
Cipher.getInstance(AesGcmUtils.decode(algo_encode));
```

Listing 8: Using Secure Cipher to Encrypt Insecure Cipher.

In Listing 8, the developer uses Base64 to decode a string, which is then decrypted using `AES/GCM/NoPadding` (i.e., `AesGcmUtils`) and a hardcoded key. However, this decryption results in `AES/ECB/NoPadding`, which eventually serves as the vulnerable parameter (see full flow in Listing 38 in the Appendix). To automated or and manual security analysis, this usage will appear to be safe, since GCM mode is visible in code, and ECB is not, when in fact it is vulnerable. Our prevalence analysis identified 15 instances of such evasive misuse.

Finding 8 ($\mathcal{F}_8$) **–** Developers use encryption, encoding, and XORs to conceal vulnerable parameter choices. In particular, developers may *use secure parameters to conceal the vulnerable parameters* in an attempt to misdirect automated and manual code reviews.

[SCORE = 0.7 – 0.79] – Even more complex instances of previously identified usage: In this score range, the cryptographic-API *(mis)use* we found were similar to what we saw previously; however, they were more complex, particularly in terms of containing additional arguments, which led to a higher complexity score.

```
Cipher.getInstance(new String
 (Base64.decode("REVTL0NCQy9QS0NTNVBhZGRpb mc=",2)));
```

Listing 9: Directly Decoding the Cipher Algorithm.

For example, we found the use of `Base64` for decoding a string that was then passed as a direct argument to `Cipher.getInstance` as shown in Listing 9 (6 instances of this type), unlike the previous findings, which were either through method calls or identifiers. We observed similar cases including the use of ternary operators (62 instances), string operation techniques such as `format` that may be combined with use of identifiers and other string operations (8 instances), a more fragmented concatenation, i.e., concatenating individual elements including "/" (65 instances), and more complex method chaining.

Finding 9 ($\mathcal{F}_9$) **–** For restrictive invocations, crypto-API usage found indirectly used in lower complexity scores, i.e., as identifiers and method invocations, were observed directly used in invocations with higher complexity.

[SCORE = 0.8 – 0.89] – Nested Ternary Operator & Separator: We found a use of nested ternary operators as direct arguments, where one ternary operator checks if the `Build.Version.SDK_INT` is lower than Marshmallow, after which a string that signifies either case is selected, as seen in Listing 18 in the Appendix. If the resultant string is "M", the next operator selects `RSA/ECB/OAEPWithSHA-1AndMGF1Padding`, whereas if it is "PREM" it selects `RSA/ECB/PKCS1Padding`.

[SCORE = 0.9 – 1.0] – Highly complex usage and obfuscation: On the lower end of this score range i.e. *SCORE #0.908438*, we find a highly complex crypto-API misuse that uses string manipulation. As shown in Listing 10, the `charAt` method is used to select specific characters, along with the addition operator "+" to add the selected characters, which eventually results in `"AES/ECB/NoPadding"`. However, the only value shown in the code is `AES/GCM/NoPadding`, i.e., this particular instance is yet another example of *evading detection tools and manual code reviews*, wherein the developer effectively uses `ECB` mode, but makes it appear as they they use `GCM` mode. Our prevalence analysis identified 858 instances of this pattern, which was also seen in prior work [33].

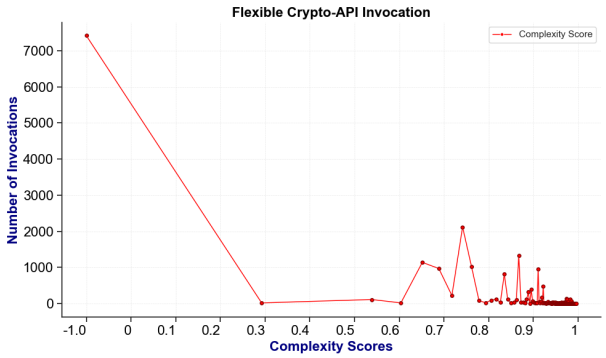


Figure 3: Flexible Invocation Complexity Score Distribution.

```
Cipher.getInstance("AES/" + ((char)
 ("AES/GCM/NoPadding".charAt(4) - 2)) +
 "AES/GCM/NoPadding".charAt(5) + ((char)
 ("AES/GCM/NoPadding".charAt(6) - 11)) + "/NoPadding")
```

Listing 10: Using charAt to Manipulate String.

Finding 10 ($\mathcal{F}_{10}$) **–** Developers may evade detectors and manual reviews using complex string manipulation that presents a benign value, but resolves to a vulnerable one.

On the higher end of this score range, we begin to find signs of obfuscation and parsing errors. Unicode characters in Java can be distinctively identified by use of a leading"\u" and in addition to complex mathematical operations in the arguments, we assume this type of code is heavily obfuscated.

4 Analysis of the Flexible Invocations

We observed 21,682 invocations of the `checkServerTrusted` API in the 20k mobile-IoT apps. Figure 3 shows their distribution across the 339 complexity scores. The `[SCORE = -1]` corresponds to an empty `checkServerTrusted` method body, and was attained by 34.2% (7,417/21,682) invocations. Further, about 20% of the invocations contribute to 309 distinct scores toward the higher end of complexity (i.e., `[SCORE = 0.9]` and above). This indicates that while the majority of the invocations are rather simple, developers may implement such flexible API in several unique and highly complex ways.

4.1 Obtaining a Representative, Stratified Sample of Flexible Invocations

As there are 339 distinct complexity scores for flexible invocations, many due to only one invocation, considering each distinct score as a stratum is infeasible. Therefore, for a viable qualitative analysis, we treated each score *range* as an

individual stratum, starting the first range at [SCORE = 0] and considering ranges at intervals of 0.1 (i.e., 0 – 0.09, 0.1 – 0.19, ..., 0.9 – 1). We consider the special case with [SCORE = -1] (i.e., empty method body) as a unique strata.

We calculate the number of representative samples for each score range (i.e., stratum) using the Qualtrics sample size calculator [46], assuming a margin of error of 5%, and confidence of 95%. Our online appendix [9] provides details on each stratum, its population, and the size of the representative sample for the stratum/score range. Using this approach, we selected a total of 2,105 representative invocations of checkServerTrusted for qualitative analysis.

4.2 Results and Findings from the Qualitative Analysis of Flexible Invocations

Our analysis of the 2,105 representative invocations helped us characterize the unique ways in which developers express flexible crypto-APIs, represented in the form of a *taxonomy of unnatural flexible crypto-API use*, as shown in Figure 4. We identified 30 *basic* ways in which developers implement certificate validation logic via checkServerTrusted API, and 22 *complex* ways in which they further combine the basic ways, resulting in 52 total taxonomy labels. Table 3 in the appendix elaborates on the 30 basic labels.

The rest of this section describes the results and findings from our qualitative analysis of flexible API invocations. We discuss score ranges with key results, providing salient examples along the way.

[SCORE = -1] – Empty Method Body: All implementations with this score have an empty method body, which causes the TrustManager to accept all certificates. Our prevalence analysis shows that this accounts for 7,417 instances.

Finding 11 ($\mathcal{F}_{11}$) **–** Vulnerable TrustManager implementations with empty checkServerTrusted methods are highly prevalent, echoing findings from prior work [15, 47]

[SCORE = 0.6 – 0.7] – Calling another method: This score range is characterized by a method call from checkServerTrusted, such that the called method is responsible for certificate validation.

```
1 public void checkServerTrusted(X509Certificate[]
   x509CertificateArr, String str) {
2  n_checkServerTrusted(x509CertificateArr, str); }
```

Listing 11: Native Method Call from Java Method.

Besides calls to other methods in Java, we also found calls to native methods, such as n_checkservertrusted in Listing 11. Our prevalence analysis reveals that native method calls account for 592 instances and other/Java method calls account for 1199 instances. As such invocations rely on other called methods for the certificate validation, the effectiveness of such method may not be apparent to security tools unless they analyze the called methods.

Finding 12 ($\mathcal{F}_{12}$) **–** Only analyzing checks in the target method (e.g., checkServerTrusted) is inadequate when analyzing crypto-API misuse in *flexible* invocations, as they rely on other native/Java methods.

[SCORE = 0.7 – 0.8] – Certification Expiration: This score range is characterized by calls to the checkValidity method to check the certificate for expiration. Listing 12 shows a typical implementation of this type of invocation.

```
1 public void checkServerTrusted(X509Certificate[]
   x509CertificateArr, String str) throws
   CertificateException {
2         for (X509Certificate x509Certificate :
           x509CertificateArr) {
3                 x509Certificate.checkValidity();}}
```

Listing 12: Checking expiration only.

It is important to note that this type of invocation accepts any unexpired certificate, and is not a substitute for certificate verification. Our prevalence analysis shows that out of the 1,632 instances of the use of the checkValidity method in checkServerTrusted, 62.5% have contain no other checks, and hence, accept all unexpired certificates.

Finding 13 ($\mathcal{F}_{13}$) **–** The check for expired certificates, if present, is commonly the *only check* in such invocations, resulting in *any* unexpired certificates getting accepted.

[SCORE = 0.8 – 0.9] – TrustStrategy, Ineffectual Implementations: We found TrustStrategy.isTrusted method from Apache *httpclient* being used for verifying certificates, often in the absence of any other certificate verification logic. Such use is vulnerable, as the TrustStrategy.isTrusted method does not consult the TrustManager that is configured in the SSLContext [26], which may void the SSL pinning configured by the developer [10]. Our prevalence analysis found 1,045 such instances.

```
1 public void checkServerTrusted(X509Certificate[]
   x509CertificateArr, String str) throws
   CertificateException {
2  try {checkClientTrusted(x509CertificateArr, str);
3  } catch (Exception e10) {
4   throw new CertificateException("Certificate not
     trusted. It has expired", e10);}
5  ...
6  public void checkClientTrusted(X509Certificate[]
    x509CertificateArr, String str) throws
    CertificateException {}
```

Listing 13: Complex implementation without checks.

In this score range, we also observed instances that had relatively complex implementations *that did not do anything*. Listing 13 shows a typical example, where checkServerTrusted calls the checkClientTrusted method, however, checkclienttrusted is empty, which in turn allows any certificate to be accepted.

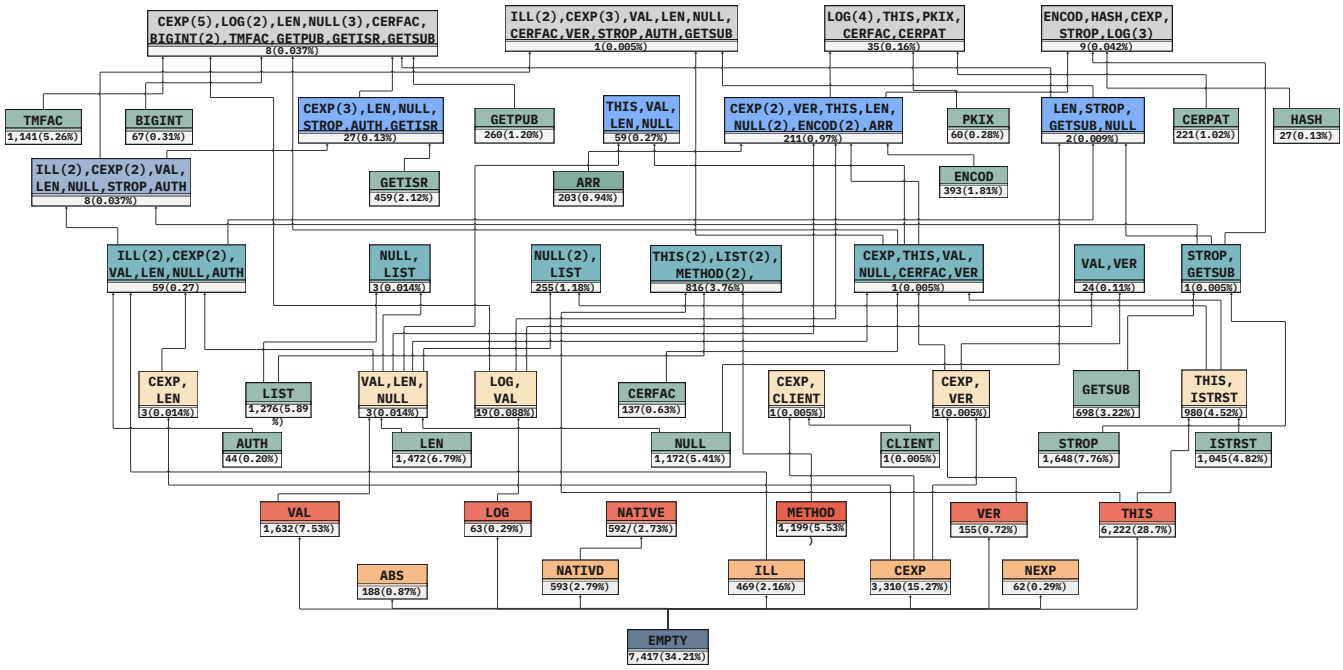


Figure 4: Taxonomy of Unnatural Flexible Crypto-API Invocations with Prevalence information.

```
public void checkServerTrusted(X509Certificate[]
 x509CertificateArr, String str) {
 for (X509Certificate x509Certificate :
  x509CertificateArr) {
  Log.e(TcpClient.TAG, "Certificate:" +
   x509Certificate);}}
```

Listing 14: Logging certificate only.

While specific implementations may vary from the one in Listing 13, we found several such ineffectual implementations that seem non-empty and complex, but do not conduct any relevant checks on the certificate chain. For example, Listing 14 shows a method that simply logs all the certificates in the certificate chain. Our prevalence analysis found 63 instances of such invocation.

Finding 14 ($\mathcal{F}_{14}$) **–** The presence of complex code in `checkServerTrusted` may not translate to effective certificate checks. Instead, developers may simply log the received certificate, or simply call one or more other method that is empty.

[SCORE = 0.9 – 1.0] – String Operations: One of the most prominent observations in samples falling in this score range is the use of string comparison/manipulation methods such as `".equals"`, `"Arrays.equals"`, `"list.add"` and `".contains"`. These methods were commonly found to be used to compare values from the certificate chain, such as the encoded form of a root certificate or a public key, and the distinguished names of the issuer or subject, with values hard-coded in the application. Listing 15 lists the common materials used for such comparisons, whereas Listing 16 shows how these values are compared.

```
x509CertificateArr[0].getencoded()
getIssuerDN()
getIssuerDN().getName()
getSubjectDN().getName()
x509Certificate.getPublicKey().getEncoded()
```

Listing 15: Certificate Materials used for Validation.

```
x509Certificate.getSubjectDN().toString().contains
("EMAIL ADDRESS=sales-usa@extron.com, CN=Quantum
 Ultra, OU=Engineering, O=ExtronElectronics,
 L=Anaheim, ST=CA, C=US")
```

Listing 16: Using ".contains" to validate certificates.

Our prevalence analysis identified 1,648 instances of the use of string operation methods described above. In addition, we also observed the use of list interfaces (1,276 instances) and array methods (203 instances) to manage certificate materials. Thus, we can draw a parallel between restrictive and flexible invocations, as string operations seems to be a common theme in both categories of crypto-API invocations.

Finding 15 ($\mathcal{F}_{15}$) **–** Both restrictive and flexible invocations of crypto-API have the use of string operation methods, method calls, and native code in common.

Finally, our prevalence analysis found 459 instances of IssuerDN and 698 instances of SubjectDN used in such comparisons, even though both are denigrated [2, 4]. Developers also use `SHA1`, which is vulnerable to collisions, to compare values of certificate materials (e.g., an encoded public key or

root certificate) with hard-coded values *to establish trust or pinning*. An example of this is shown in Listing 37 in the Appendix. Our prevalence analysis finds 43 instances of `SHA1` use in similar certificate validation contexts, 21 instances of `SHA1` use by itself (i.e., not in such contexts), and 6 of `SHA256` being used in such certificate validation contexts. Additionally, developers also hard code specific certificate chain material for validation purposes, as shown Listing 16.

Finding 16 ($\mathcal{F}_{16}$) – Developers use denegrated methods such as `getSubjectDN` and `getIssuerDN`, and deprecated algorithms (`SHA1`) in certificate validation.

5 The Effectiveness of Tools at Detecting Misuse in Real-world Crypto-API Invocations

The observations from our analysis regarding real-world crypto-API usage motivate a critical question for vulnerability detection in practice: *how effective are popular detection tools at finding misuse in real-world invocations, particularly those exhibiting the unnaturalness characterized in this study?*

Experimental Setup: To answer the this question, we isolated a set of 23 of the most *basic* cases from a total 96 in our taxonomies, such that if a tool fails to detect these, it will fail to detect any of the other more unnatural cases. These 23 cases represent a total of 59,002 instances of observed crypto-API invocations, i.e., 45,789 restrictive and 13,213 flexible invocations. We developed minimal APKs with these 23 cases implemented using invocations with vulnerable parameters found in our study. For some cases, such as `ID`, we implemented more than one vulnerable invocation as there were multiple possible variants from our dataset. We then tested 6 popular tools: CryptoGuard [16], CogniCrypt [11], SpotBugs [56, 57], CodeQL [17], Semgrep [50], and MobSF [36].

Results: As seen in Table 1, most tools struggle at detecting even the most basic 23 unnatural cases. Only 5/23 cases were detected (✔) by at least one tool. No tool could reason about 12/23 cases, representing over 12,876/59,002 invocations, which highlights a significant gap in the detection capabilities of existing tools when dealing with real-world invocations which are even mildly unconventional. One welcome development is that instead of silently failing, CogniCrypt declares that it cannot reason about the usage in 4 of these 12 cases (representing 7092/59,002 instances), i.e., ✖.

Finding 17 ($\mathcal{F}_{17}$) – Popular tools struggle to reason about basic forms of unnatural invocations uncovered in this study (n=23). No tool could reason about 12/23 cases, representing 12,876/59,002 invocations. In only 5/23 cases was there at least one tool that could reason about it. Finally, there was no case that every tool could detect.

Finally, we marked the remaining 6/23 cases (representing 7,161/59,002 invocations) as *partially* detected (◗), as they represent the following situations where the tool cannot reason about the misuse but still reports a warning: the tool (1) misidentifies a correct use in the invocation as a misuse, while failing to detect the actual misuse, or (2) detects only one misuse in an invocation containing more than one, or, (3) detects only some instances of a specific case, indicating hardcoded rules, and not the ability to detect a type of invocation.

Table 1: Detection of 23 basic cases from the taxonomies by popular crypto-API misuse detection tools.

	Label	CCRYPT	CGUARD	SPOTBUGS	CODEQL	SEMGREP	MOBSF
Restrictive Invocations							
1.	**STRING/OID**	✔	✖	✔	✔	✖	✖
2.	**ID**	✔	✔	✔	✔	◗	✖
3.	**METHOD**	✖	✖	✖	✖	✖	✖
4.	**METHOD***	✖	✖	✖	✖	✖	✖
5.	**NATIVE**	✖	✖	✖	✖	✖	✖
6.	**STROP**	✖	✖	✖	◗	✖	✖
7.	**STRBUF**	✖	◗	✖	✖	✖	✖
8.	**STRBL***	✖	◗	✖	✖	✖	✖
9.	**CONCT**	✔	✔	✔	◗	◗	✔
10.	**BAS64**	✖	✖	✖	✖	✖	✖
11.	**ID,METHOD**	✖	✖	✖	✖	✖	✖
12.	**TEROP**	◗	◗	✖	◗	✖	✖
13.	**STATIC**	✔	✖	✔	◗	✖	✖
14.	**ENUM**	✖	✖	✖	◗	✖	◗
Flexible Invocations							
1.	**EMPTY**	✖	✖	✔	✔	✔	✖
2.	**LOG**	✖	✖	✖	✖	✖	✖
3.	**CLIENT**	✖	✖	✖	✖	✖	✖
4.	**VAL**	✖	✖	✖	✖	✖	✖
5.	**HASH**	✖	◗	◗	◗	◗	✖
6.	**GETSUB**	✖	✖	✖	✖	✖	✖
7.	**LEN/AUTH**	✖	✖	✖	✖	✖	✖
8.	**GETPUB**	✖	✖	✖	✖	✖	✖
9.	**STROP**	✖	✖	✖	✖	✖	✖

✔ = detected, ◗ = partially detected, ✖ = undetected, ✖ = undetected due to error
METHOD* = Multiple method calls, **STRBL***=StringBuilder

6 Discussion

Our characterization of crypto-API usage in the wild demonstrates that developers invoke crypto-APIs in highly diverse, complex, and non-intuitive ways in the wild. Our experiments further demonstrate that popular detection tools cannot reason about many prevalent, non-trivial, usage patterns. Thus, this work motivates the re-thinking of security-analysis techniques to detect vulnerabilities in unnatural invocations, as we can no longer pretend they do not exist. To this end, we distill the 17 key findings into 4 discussion themes that capture the challenges for vulnerability detection in this domain, and

highlight actionable opportunities for future research.

6.1 Even *natural* cases need attention

Tool designers often perceive the use of identifiers and string literals as a natural way for developers to use APIs such as `Cipher.getInstance`, in contrast to more complex usage that is deemed out of scope for detection. However, we observe that even such seemingly natural patterns may exhibit unnaturalness in practice, which tools are unprepared for. For instance, identifiers or string literals may represent OIDs that point to vulnerable ciphers ($\mathcal{F}_2$), *which generally receive little attention from detection tools*.

This gap between perceived naturalness and code in practice holds true for flexible API as well. Conventional wisdom dictates that `checkServerTrusted` method implementations that are empty are vulnerable, but those containing some code generally perform certificate validation checks. Tools designed to detect misuse in these methods follow the same intuition [15], and identify vulnerability with empty/sparse methods, thus ignoring methods with some code. However, we find that methods that score high on our complexity metric, i.e., that contain significant code, exhibit unnatural and often vulnerable behavior, such as simply logging certificates, returning true via a method chain that does not perform any checks ($\mathcal{F}_{14}$), or only checking the expiration date ($\mathcal{F}_{13}$).

To summarize, we observe that broad assumptions by tool designers regarding syntactically conventional usage, e.g., single String literals in `Cipher.getInstance`, or an non-empty `checkServerTrusted` method, may not hold against real-world usage with unexpected semantics.

Takeaway 1

Even if the consensus is that tools should only target conventional/natural misuse, we still need to re-evaluate our assumptions about what such natural expression entails.

6.2 Evasion is Prevalent and Complex

Prior work has discussed how developers may leverage complex patterns to evade vulnerability detectors [6], and developers have described observations of evasion in practice [7, 34]. Our analysis uncovered unique cases of potential evasion of not only automated tools, *but also manual code reviews*.

Particularly, we found several cases where developers would present a benign argument in code, but actually use a vulnerable one ($\mathcal{F}_8$, $\mathcal{F}_{10}$). For instance, we found cases where a vulnerable argument (e.g., `ECB` mode) was used to encrypt data, but was not visible in code, as it was encrypted using a secure algorithm (i.e., `AES` in `GCM` mode) (e.g., Listing 8, and Listing 38 in the Appendix). Similarly, we found code that uses XORs to change secure parameter values to vulnerable ones at runtime (e.g., Listing 39 in the Appendix). Further, we found several cases where developers would use String operations such as `charAt` to present `GCM` use in code, but transform it to `ECB` at runtime, as recent work has observed [33] (e.g., Listing 10). Such use would visually resemble the use of the secure algorithm for encryption, thereby evading both security tools as well as manual code reviews.

Beside these unambiguous attempts at evasion, we also found code with complex expression of vulnerable parameters, which may be potentially to evade automated tools. For instance, we found the use of `StringBuffer` API and simple string transformations to combine arguments in non-intuitive ways (e.g., appending "/EC" and "B/...", which essentially results in ECB mode) ($\mathcal{F}_7$), or through complex encoding and encryption ($\mathcal{F}_8$). Further, we found evidence of developers hiding vulnerable parameters in native code ($\mathcal{F}_3$), i.e., where the API itself would be called in Java, but the parameters would be fetched at runtime via a JNI call.

These findings are particularly concerning because in most cases of such evasive use, we observe that the eventually resolved API parameters often tend to be vulnerable. As we discuss in Ethical Considerations (Appendix 9), we have reported all such usage as potential backdoors to Google, in addition to reporting them to app developers.

Takeaway 2

We can no longer exempt evasive misuse from the design goals of automated vulnerability detection tools, as evasion is present, complex, and may indicate backdoors.

6.3 What motivates unnatural usage?

One resounding question that may come up given our analysis results is: *why do developers implement such complex, non-intuitive, misuse?* One rationale could be performance.

This is particularly true in the case of mobile-IoT applications that need to interface with IoT devices and the cloud in realtime. Using a secure cipher such as `AES/GCM/NoPadding` may cost more runtime relative to a vulnerable option such as `AES/ECB/NoPadding`, motivating developers to use the latter. Alternately, developers may be motivated by tight deadlines, and may take the easy way out of tweaking invocations to finish the job, instead of avoiding or fixing vulnerabilities, as recent work has found [7]. That is, they may use vulnerable options such as empty `checkServerTrusted` methods purely because they work, or write evasive code to get around a quality assurance in the organization, particularly if they are independent contractors with no stake in the product. Developer forums such as StackOverflow certainly do not improve the status quo, e.g., there are several examples of developers struggling to use `GCM` for encrypting and decrypting large files [51–53], with experts recommending `AES/ECB/NoPadding` and `AES/CBC/PKCS5Padding` because they "eliminate the inconvenience".

To summarize, several practical factors, such as performance, deadlines, and a lack of ownership in the product, may

incentivize developers to invoke crypto-APIs in convoluted, unnatural ways, often resulting in misuse and vulnerabilities.

Takeaway 3

As these factors incentivizing unnatural and evasive code are not expected to disappear, the resulting unnatural code will likely proliferate as well. This further motivates proactive re-design of existing analysis techniques to account for unnatural and/or evasive misuse.

6.4 Toward Effective Detection Grounded in a Real-World Characterization

A large-scale qualitative study is the first step in developing an evidence-based understanding of real-world crypto-API usage, and the challenges in detecting it. Particularly, we note that our findings would not have been possible using existing tools (i.e., without the qualitative study), as existing tools fail to detect most basic types of invocations, let alone complex ones (see Section 5) This underscores a broader lesson – that *understanding a problem in the wild is a prerequisite for building informed, effective, tools to detect it*.

The taxonomies and findings from this work lay the foundation for exactly this understanding in the context of crypto-API misuse. In doing so, this paper enables several short and long-term research opportunities for improving the state of the crypto-API misuse detection in practice, as described below.

1. Helping security tools to systematically triage real-world crypto-API invocations: The taxonomies from this work are a precise classification of real-world crypto-API usage, accompanied by real code examples representative of each unique class. Existing crypto-API misuse detectors will be able to leverage our taxonomies to significantly improve their outcomes. At the very least, tools will be able to precisely declare what specific expressions of crypto-APIs from the taxonomies they cannot handle, using the code examples in the taxonomies to test if needed. Such declaration will not only help tools systematically identify gaps and improve, but will also improve tool documentation, and enable tool users to adjust their expectations and understand tool output.

Building on this, tools will be able to triage invocations at runtime, by matching invocations they encounter with classes in the taxonomy, using both complexity scores and the distinctive arguments/variables that exemplify each class. Such triaging will help tools identify invocations they cannot reason about without significant analysis, and thus provide helpful information to the tool user for further analysis.

2. Enabling benchmarks and evaluation frameworks grounded in real usage: The results from our analysis and the taxonomies will provided a basis for testing crypto-API misuse detectors, and particularly when it comes to developing automated frameworks for the same. For instance, frameworks such as MASC [6], which develop synthetic variants of crypto-API misuse using mutation, will be able to directly adopt classes from our taxonomy (and accompanying code) as a base cases for mutation, as well as inspiration for developing new mutation operators. The mutants and resultant benchmark developed from such a framework will be highly relevant, as it will be grounded in systematically classified crypto-API usage in the wild.

3. Enabling find-tuned AI support to complement SAST: As we see in this study, a major challenge for detectors is the interpretation of complex or unnatural usage. However, the detailed taxonomies of code with text descriptions produced in this work, in conjunction with the present developments in large language models (LLMs), exposes a significant research opportunity to address this problem.

To elaborate, we envision LLMs trained using few-shot learning, made possible by the real code examples and accompanying text descriptions of crypto-API usage classes from the taxonomies. Such LLMs can then be used to make the detection of misuse in unnatural crypto-API invocations practical. For instance, the LLMs could be used to resolve the effective arguments, parameters, or other identifying factors relevant to a specific class, which could then be checked to detect misuse. Or, the LLMs could be used to simplify the invocation, i.e., to summarize it further in a more simpler form that existing SAST tools can reason about, thereby pre-processing code for detection by SASTs. While it is unclear how well LLMs will be able to reason about crypto-API invocations in general, or how suitable they would be in detecting vulnerable use (although a very recent evaluation shows promise [35, 60]), the multi-modal ground truth data from this study will certainly give LLMs an edge in this problem space by infusing domain knowledge.

Takeaway 4

The ground-truth characterization of real-world crypto-API misuse from this work enables actionable opportunities for improving, benchmarking, and re-designing crypto-API detection techniques (e.g., using LLMs).

7 Threats to Validity

We now describe the threats to the validity of our study. **1. Manual Analysis, exploitability:** Given that we seek to find the very unnatural variants of vulnerabilities that automated tools are incapable of detecting [6], using automated tools or building a detector, was incompatible with our research goal. Instead, our objective required us to meticulously construct the ground truth taxonomies, which may future research on building automated tools to detect such latent vulnerabilities. That said, due to the manual approach, we do not claim completeness in terms of capturing all unnatural patterns. Further, we consider validating the exploitability of the misuse out of the scope of this work, as the focus is un-

derstanding the nature of vulnerable crypto-API invocations in order to improve their detection. That said, prior work shows the detection of well-known types of misuse generally resolves to true positives (e.g., 24.7% false alarms for `ECB` alarms from CryptoREX, as per Chen et al [12]). **2. Generalizability with respect to non-IoT apps:** Our decision to focus on mobile-IoT apps was influenced by the *impact* from the critical attack surface they represent, and their likelihood of using crypto-APIs. This decision resulted in a rich and large set of invocations for qualitative analysis, leading to significant results and findings. That said, to explore if any new patterns would be found in an analysis of the broader Android app population, we extended our analysis to all non-IoT apps from Jin et al.'s dataset [27] that are now available on Google Play, i.e., a sample of 409 confirmed non-IoT Android apps. We extracted 2,728 crypto-API invocations (2,451 restrictive and 277 flexible) from these apps. We found that the distribution of complexity scores in non-IoT apps, for both the flexible and restrictive case, are strikingly similar to those seen for mobile-IoT apps in Figure 1 and Figure 3 (see Appendix A for non-IoT app figures), except for the lack of empty `Cipher.getInstance` invocations (`[SCORE = -1]`) in the non-IoT apps. Moreover, our qualitative analysis of the 2,728 invocations did not reveal any new patterns. **3. Generalizability with respect to other crypto-APIs:** Our results reflect the usage of two highly popular and well-studied crypto-APIs, `Cipher.getInstance` and `checkServerTrusted`, which have also been historically well-represented in vulnerability research in this domain [6,12,15,21,27,38,45,49]. To explore generalizability to other crypto-APIs, we repeated our study on two additional APIs that are highly relevant in mobile application security: (1) `SecretKeySpec`, which is used to handle the construction of cryptographic keys from byte arrays, and mostly falls under the *restrictive* invocation category, and (2) `HostnameVerifier`, from the *flexible* category. We extracted 10,568 invocations of both APIs, of which 3,459 were sampled for manual analysis. On analyzing these invocations, we found no new patterns; instead, we found repeats of patterns that form our taxonomies. The distributions of complexity scores for `SecretKeySpec` and `HostnameVerifier` were also similar to those for `Cipher.getInstance` (i.e., Figure 1) and `checkServerTrusted` (i.e., Figure 3) respectively. Thus, while we do not claim generalizability, these additional insights demonstrate that characterization is representative of a dominant subset of interesting patterns of crypto-API invocation in the wild. Detailed results from this additional analysis can be found in Appendix B.

8 Related Works

This paper presents the first study of the unnatural use of crypto-API in the wild. As a result, we discuss closely related work pertaining to SAST tools & frameworks for finding crypto-API misuse [14,15,29,31,36,37,44,47,59,61], existing benchmarks & evaluation frameworks [6, 13, 47–49, 55] and qualitative studies [7, 18, 19, 21, 30, 38, 40, 58].

MalloDroid by Fahl et al. [15] was one of the first SASTs to evaluate Android apps for SSL/TLS-misuse vulnerabilities. Some of our findings resonate with MalloDroid, as even 12 years later, empty `checkServerTrusted` methods remain extremely common and as shown in Table 1, MalloDroid still performs significantly better than state-of-the-art crypto-detector tools. Egele *et al.* [14] developed a tool called CryptoLint which is based on six basic rules which Android apps should adhere to ensure indistinguishability under chosen plaintext attacks (IND-CPA). They found that 88% of mobile apps violate at least one rule. Similar rules were employed by Zhang *et al.* [61] to design a tool called CRYPTOREX and applied the rules in the context of IoT firmwares, finding similar problems, i.e., close to 25% of IoT firmware violates at least one rule. In order to better support developers, Krüger *et al.* developed CogniCrypt using 23 rules to ameliorate the problem with use of low level cryptographic primitives [31]. Similarly, Cryptoguard tool and benchmark were designed based on 16 rules [47]. These types of techniques used by state of the art crypto-detector tools may be effective against simple types of misuse, and even perform well on benchmarks from tool designers, such as CryptoAPI-Bench [47] and CamBench [49].

Finally, some of our observed invocations bear significant resemblance to the synthetic mutants generated by MASC [6]. Particularly, the cases with string operations and concatenation observed in our analysis of restrictive crypto-API use are similar to the mutation operators built for MASC. However, it is important to note that our work is focused on finding relevant real world examples in the wild, and is the first qualitative characterization of unnatural misuse in this space.

9 Conclusion

This paper performed the first large-scale qualitative study (n=5,704 API invocations) of odd/unnatural use of cryptographic-APIs in the wild. The resultant taxonomies capture this characterization in 96 unique classes of invocation, many of them non-trivial. Our 17 findings effectively resolve the initial contention posed in the paper, and show that a diverse spectrum of unnatural usage is prevalent in the wild, and should be within the scope of work of crypto-API misuse detection tools. More importantly, we show that existing detectors do not detect even most of the basic cases of unnatural uses from the taxonomies, thereby motivating the re-design of tools with a focus on such hard-to-detect vulnerabilities. The results, findings and discussion from this paper emphasize the practical challenges for vulnerability detection in the face of unnatural API invocation, and describe actionable opportunities to leverage the insights and artifacts from this study for improving vulnerability detection in this domain.

Ethical Considerations

Stakeholders: We identified two primary stakeholders that may be impacted by our work, these include the application developers and the end user. In order to reduce harm to users and app developers, have followed a responsible vulnerability disclosure process, described later in this section. Moreover, it is also important to note that we do not disclose application names in the paper, as again, the goal is not to study specific apps, but the nature of certain misuse.

Responsible Disclosure: Following the principle of beneficence, we conducted responsible disclosure prior to publicizing our work. We have reported all the misuse found in our qualitative study (i.e., analysis of 5,704 crypto-API invocations) to the appropriate vendors/app developers, and in some cases, also to Google. To elaborate, we submitted reports containing one or more discovered misuse to 241 app developers. In the case of 161 applications, where we found the expression of the misuse to be highly evasive, we also reported the misuse as a potential crypto backdoor to Google, in order to protect users. We noted that at the time of our reporting, some of the applications with evasive use cases had already been taken down from Google Play, including three applications that hid vulnerable parameters in native method calls and one that used base64.

Potential Harm Mitigation: It is important to highlight that this research did not experiment with live systems, as we only statically analyzed code obtained from mobile apps. However, we automatically downloaded the APKs corresponding to our list of mobile apps (obtained from Jin et al. [27]) from Google Play. To prevent any adverse impact on Google Play, we did not run multiple scripts concurrently, and only only ran a single downloader, with a 5 secs sleep timer after every APK searched/downloaded. We did not collect developer information such as name, affiliation or/and emails. We also did not collect user reviews and/or users PII associated with apps from Google Play. Finally, we followed a responsible vulnerability disclosure process, which mitigates harm to users (in the form of undisclosed vulnerabilities) and app developers (in the form of harm to reputation from irresponsible disclosure of vulnerabilities).

Decision: Our research aims to measure the unnaturalness of cryptographic-API (mis)use in the wild. This work uncovers several critical cryptographic-API misuse which may have eluded research and industry security tools. We carefully weighed the potential benefits of the research against any possible harm, and concluded that our research provides a net positive to the community.

References

[1] Android String. https://developer.android.com/reference/java/lang/String. Last accessed on August 26, 2024.

[2] Android X509Certificate. https://developer.android.com/reference/java/security/cert/X509Certificate. Last accessed on August 26, 2024.

[3] Cipher is susceptible to Padding Oracle. https://find-sec-bugs.github.io/bugs.htm#PADDING_ORACLE. Last accessed on February 05, 2023.

[4] Java X509Certificate. https://docs.oracle.com/javase/8/docs/api/java/security/cert/X509Certificate.html. Last accessed on August 26, 2024.

[5] Tree-Sitter. https://tree-sitter.github.io/tree-sitter/. Last accessed on June 18, 2024.

[6] Amit Seal Ami, Nathan Cooper, Kaushal Kafle, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. Why crypto-detectors fail: A systematic evaluation of cryptographic misuse detection techniques. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 614–631. IEEE, 2022.

[7] Amit Seal Ami, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. "False negative - that one is going to kill you" - Understanding Industry Perspectives of Static Analysis based Security Testing. In *Proceedings of the 2024 IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, May 2024. IEEE Computer Society.

[8] Android. Cipher. https://developer.android.com/reference/kotlin/javax/crypto/Cipher, 2024. Last accessed on June 23, 2024.

[9] Anonymous. Unnaturalness in crypto-api misuse - online appendix. https://sites.google.com/view/unnaturalness/, 2025. Last accessed on Jan 6, 2024.

[10] Apache. The apache software foundation. https://github.com/apache/httpcomponents-core/blob/master/httpcore5/src/main/java/org/apache/hc/core5/ssl/TrustStrategy.java, 2024. Last accessed on June 23, 2024.

[11] Collaborative Research Center CROSSING at TU Darmstadt. Cognicryptsast. https://github.com/CROSSINGTUD/CryptoAnalysis, 2024. Last accessed on Jan 1, 2024.

[12] Yikang Chen, Yibo Liu, Ka Wu, Duc Le, and Sze Chau. Towards precise reporting of cryptographic misuses. 01 2024.

[13] Yikang Chen, Yibo Liu, Ka Lok Wu, Duc V Le, and Sze Yiu Chau. Towards precise reporting of cryptographic misuses. 2024.

[14] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84, 2013.

[15] 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 *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61, 2012.

[16] Miles Frantz. Cryptoguard. https://github.com/CryptoGuardOSS/cryptoguard, 2024. Last accessed on Jan 1, 2024.

[17] GitHub. Codeql. https://github.com/github/codeql, 2024. Last accessed on Jan 1, 2024.

[18] Peter Leo Gorski, Yasemin Acar, Luigi Lo Iacono, and Sascha Fahl. Listen to developers! a participatory design study on security warnings for cryptographic apis. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[19] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[20] David Sounthiraraj Justin Sahs Garret Greenwood and Zhiqiang Lin Latifur Khan. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *Network and Distributed System Security Symposium (NDSS). Internet Society, San Diego, CA*, pages 1–14, 2014.

[21] Mohammadreza Hazhirpasand, Mohammad Ghafari, and Oscar Nierstrasz. Java cryptography uses in the wild. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, 2020.

[22] IETF. RFC 2268 - A Description of the RC2(r) Encryption Algorithm. https://datatracker.ietf.org/doc/html/rfc2268#section-6, 2024. Last accessed on June 23, 2024.

[23] IETF. RFC 3394 - Advanced Encryption Standard (AES) Key Wrap Algorithmm. https://www.ietf.org/rfc/rfc3394.txt, 2024. Last accessed on June 23, 2024.

[24] J. Developers,. jadx - Dex to Java decompiler. https://github.com/skylot/jadx/, 2022.

[25] Java. Java cryptography architecture sun providers documentation. https://docs.oracle.com/javase/6/docs/technotes/guides/security/SunProviders.html#SunJCEProvider, 2024. Last accessed on June 23, 2024.

[26] Javadocs. Interface truststrategy. https://www.javadoc.io/doc/org.apache.httpcomponents/httpclient/4.3.2/org/apache/http/conn/ssl/TrustStrategy.html, 2024. Last accessed on June 23, 2024.

[27] Xin Jin, Sunil Manandhar, Kaushal Kafle, Zhiqiang Lin, and Adwait Nadkarni. Understanding iot security from a market-scale perspective. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1615–1629, 2022.

[28] Stefan Krüger, Karim Ali, and Eric Bodden. Cognicryptgen: generating code for the secure usage of crypto apis. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 185–198, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. Cognicrypt: Supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–936. IEEE, 2017.

[30] Stefan Krüger, Michael Reif, Anna-Katharina Wickert, Sarah Nadi, Karim Ali, Eric Bodden, Yasemin Acar, Mira Mezini, and Sascha Fahl. Securing your crypto-api usage through tool support-a usability study. In *2023 IEEE Secure Development Conference (SecDev)*, pages 14–25. IEEE, 2023.

[31] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering*, 47(11):2382–2400, 2021.

[32] MalloDroid. MalloDroid. https://github.com/sfahl/mallodroid, 2013.

[33] Prianka Mandal, Amit Seal Ami, Victor Olaiya, Sayyed Hadi Razmjo, and Adwait Nadkarni. "Belt and suspenders" or "just red tape"?: Investigating Early Artifacts and User Perceptions of IoT App Security Certification. In *Proceedings of the 2024 USENIX Security Symposium (USENIX)*, August 2024.

[34] Prianka Mandal and Adwait Nadkarni. "We can't change it overnight": Understanding Industry Perspectives on IoT Product Security Compliance and Certification. In *Proceedings of the 2025 IEEE Symposium on Security and Privacy (S&P 2025)*, May 2025.

[35] Zohaib Masood and Miguel Vargas Martin. Beyond static tools: Evaluating large language models for cryptographic misuse detection. *arXiv preprint arXiv:2411.09772*, 2024.

[36] MobSF. Mobile Security Framework (MobSF). https://github.com/MobSF/Mobile-Security-Framework-MobSF, 2022.

[37] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. Source attribution of cryptographic api misuse in android applications. ASIACCS '18, page 133–146, New York, NY, USA, 2018. Association for Computing Machinery.

[38] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering*, pages 935–946, 2016.

[39] National Security Agency. Ghidra Software Reverse Engineering Framework. https://github.com/NationalSecurityAgency/ghidra, 2024. Last accessed on June 23, 2024.

[40] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A DeLong, Justin Cappos, and Yuriy Brun. {API} blindspots: Why experienced developers write vulnerable code. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 315–328, 2018.

[41] Marten Oltrogge, Nicolas Huaman, Sabrina Amft, Yasemin Acar, Michael Backes, and Sascha Fahl. Why eve and mallory still love android: Revisiting tls (in) security in android applications. In *USENIX Security Symposium*, pages 4347–4364, 2021.

[42] Oracle. Certificate. https://docs.oracle.com/javase/8/docs/api/java/security/cert/Certificate.html. Last accessed on August 29, 2024.

[43] Oracle. Certificate factory. https://docs.oracle.com/javase/8/docs/api/java/security/cert/CertificateFactory.html. Last accessed on August 29, 2024.

[44] 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)*, pages 1972–1989, 2021.

[45] Amogh Pradeep, Muhammad Talha Paracha, Protick Bhowmick, Ali Davanian, Abbas Razaghpanah, Taejoong Chung, Martina Lindorfer, Narseo Vallina-Rodriguez, Dave Levin, and David Choffnes. A comparative analysis of certificate pinning in android & ios. In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, page 605–618, New York, NY, USA, 2022. Association for Computing Machinery.

[46] Qualtrics. Sample size calculator. https://www.qualtrics.com/blog/calculating-sample-size/, 2024. Last accessed on June 23, 2024.

[47] 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.

[48] Michael Schlichtig, Steffen Sassalla, Krishna Narasimhan, and Eric Bodden. Introducing fum: A framework for api usage constraint and misuse classification. 2023.

[49] Michael Schlichtig, Anna-Katharina Wickert, Stefan Krüger, Eric Bodden, and Mira Mezini. Cambench–cryptographic api misuse detection tool benchmark suite. *arXiv preprint arXiv:2204.06447*, 2022.

[50] semgrep. semgrep. https://github.com/semgrep/semgrep, 2024. Last accessed on Jan 1, 2024.

[51] StackExchange. Plain text size limits for aes-gcm mode at 2gb with sun jce implementation? https://crypto.stackexchange.com/questions/71712/plain-text-size-limits-for-aes-gcm-mode-at-2gb-with-sun-jce-implementation, 2019. Last accessed on August 30, 2024.

[52] StackExchange. Aes/gcm/pkcs5padding giving issues while aes/cbc/pkcs5padding works fine for my use case. https://crypto.stackexchange.com/questions/89030/aes-gcm-pkcs5padding-giving-issues-while-aes-cbc-pkcs5padding-works-fine-for-my?rq=1, 2021. Last accessed on August 30, 2024.

[53] StackOverflow. Best way to aes encrypt large files (approx. 5gb). https://stackoverflow.com/questions/72949189/best-way-to-aes-encrypt-large-files-approx-5gb, 2022. Last accessed on August 30, 2024.

[54] StackOverflow. XOR operation with two strings in java. https://stackoverflow.com/questions/5126616/xor-operation-with-two-strings-in-java, 2024. Last accessed on June 23, 2024.

[55] Cong Sun, Xinpeng Xu, Yafei Wu, Dongrui Zeng, Gang Tan, Siqi Ma, and Peicheng Wang. Cryptoeval: Evaluating the risk of cryptographic misuses in android apps with data-flow analysis. *IET Information Security*, 17(4):582–597, May 2023.

[56] SpotBugs Team. Owasp find security bugs. https://github.com/find-sec-bugs/find-sec-bugs/, 2024. Last accessed on Jan 1, 2024.

[57] SpotBugs Team. Spotbugs. https://github.com/spotbugs/spotbugs, 2024. Last accessed on Jan 1, 2024.

[58] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 109–126, 2020.

[59] Anna-Katharina Wickert, Lars Baumgärtner, Florian Breitfelder, and Mira Mezini. Python crypto misuses in the wild. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, 2021.

[60] Yifan Xia, Zichen Xie, Peiyu Liu, Kangjie Lu, Yan Liu, Wenhai Wang, and Shouling Ji. Exploring automatic cryptographic api misuse detection in the era of llms. *arXiv preprint arXiv:2407.16576*, 2024.

[61] Li Zhang, Jiongyi Chen, Wenrui Diao, Shanqing Guo, Jian Weng, and Kehuan Zhang. {CryptoREX}: Large-scale analysis of cryptographic misuse in {IoT} devices. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 151–164, 2019.

A Non-IoT Application Analysis

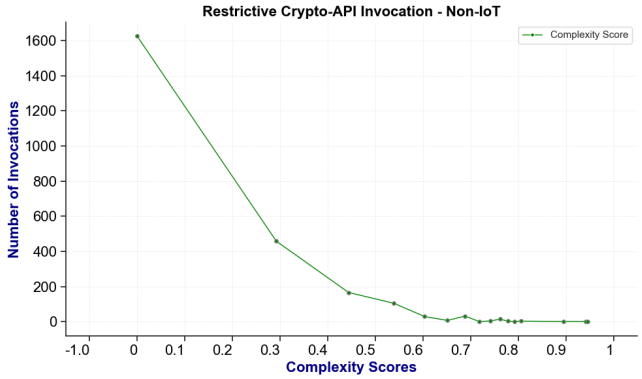
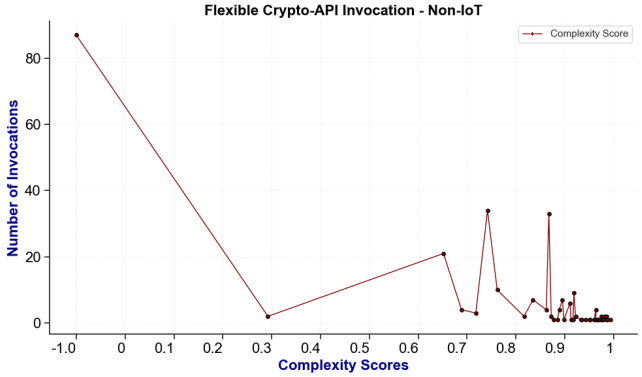


Figure 5: Complexity Score Distribution for Restrictive Invocation for Non-IoT Apps.



Note: Score -1 has 87 number of invocations and is not in the graph.

Figure 6: Complexity Score Distribution for Flexible Invocation for Non-IoT Apps.

We decompiled 409 mobile apps (non-IoT) and found 2,451 Cipher.getInstance restrictive invocations and 277 checkServerTrusted flexible invocations. Figure 5 & 6 show the distribution of complexity scores vs number of invocations for restrictive and flexible invocations for non-IoT apps respectively. Both figures can be observed to be highly similar to their respective counterpart for mobile-IoT apps which indicates that the trends observed in mobile-IoT apps is similar to that in non-IoT apps. For restrictive invocations, `[SCORE = 0]` had 1,625 invocations with the use of identifiers only being 462 and use of string only being 1,163 invocations. `[SCORE = 0.2923]` had 535 invocations, `[SCORE = 0.4439]` had 89 invocations, `[SCORE = 0.5385]` had 105, and the rest from `[SCORE = 0.6 - 0.9363]` having a total of 97 invocations with 0.9363 being the maximum score. For flexible invocations, our analysis found 277 invocations for checkServerTrusted with 31.4% (87/277) having a `[SCORE = -1]`i.e. implementing an empty Trust manager. This is similar to the 34.2% of invocations implementing an empty Trust manager for mobile-IoT apps. `[SCORE = 0]` accounts for 3.61% (10/277) of the dataset which consists of abstract methods (4/277) and native method declarations (6/277). This is very similar to `[SCORE = 0]` for mobile-IoT apps.

We did not observe any new patterns on qualitative analysis of the 2,728 invocations from non-IoT apps. Instead, we observed two additional salient examples of two key types in the restrictive taxonomy (Figure 2), namely the `ID` and `METHOD` type, which we discuss as follows:

Evasive Example, Identifier use: One key finding for identifier use as shown in Listing 39 is assigning it to a method call. This method call takes a cipher algorithm as an argument i.e. "AES/CBC/PKCS5Padding". The key idea here is to get the byte value of the string 'C', 'B', 'C', and perform bit-wise XOR with the byte value of '6','1','1' i.e., 'C' byte value is XOR-ed with '6' and so on. This operation produces 'ECB' when converted to string. We developed minimal working examples using code found in live applications. We found 6 instances of this used for encryption and decryption across multiple applications.

Example of Method calls: As shown in Listing 34, method calls were used to extracts specific byte values from a hardcoded list of bytes, offsets the byte values by a specific number, and appends each character to a StringBuilder before converting the output to string. This type of code led to 3 instances of using "AES" and 2 instances of using "RSA/ECB/PKCS1PADDING" cipher algorithms. We found 5 instances of this use in the same mobile application.

B Generalizability with respect to other Crypto-APIs – Additional Details

We find similar trends as regards complexity scores for `SecretKeySpec` (see online appendix [9]), except for missing `[SCORE = 0]` and `[SCORE = -1]`. `[SCORE = 0]` is missing because the baseline use case for this API requires two arguments, unlike `Cipher.getInstance`, which has an optional argument to specify a provider. We find the use of hardcoded keys (Listing 36), use of base64 (Listing 35), method call, stringBuffer, use of identifiers to store hardcoded keys, use of enum, concatenation, which reveals that developers often use similar techniques we observed in our analysis of `Cipher.getInstance`. The only difference we observed is in one case, where the developers stored the generated key material in a file, which is not an essential aspect of the crypto-API invocation itself.

The trends observed for `HostnameVerifier` are similar to those in `checkServerTrusted`. We found empty implementations or those that always return true, allowing all hostnames. We observed more complex implementations of `HostnameVerifier` that call another method that returns true

Table 2: Restrictive Taxonomy Labels

Label	Label Name	Description of Labels
STROP	String Operation	This refers to the use of any of the string methods such as `charAt`, `format`, `toString` etc. as depicted in Android developers documentation [1]. We associate every use of such method with this label. Listing 17
TEROP	Ternary Operator	This refers to the use of the conditional operator that accepts three operands. This mimics the if-else statement such that based on a conditional (first operand), any of the two other operand could be the return value. Listing 19
ENUM	Enum Type Constant**	Enum Type is a class used to store predefined constant in code such as various cipher algorithms, mode, padding etc. Enum type offers more functions such as use with switch case statements, type and value safety etc. Listing 20
ID	Identifier	Identifiers refers to unique variable name particularly assigned to cipher algorithm, mode, and/or padding.Listing 21
THIS	Reference Current Object	Referencing the current object refers to using the "this" keyword to refer to the current object instance variable or method. Listing 22
METHOD	Method Call	Used to indicate the presence of a *function* call that returns the cipher algorithm., mode, and/or padding. Listing 23
STATIC	Static Field Constant	Similar to enum type, it is used to store predefined values, however, unlike enum type it does not offer additional functions.Listing 24
NATIVE	Native Method Call	This refers to the accessing methods or variables implemented in C/C++ and stored in native libraries. Listing 25
BAS64	Base64-Decoder	This refers to decoding byte data encoded using Base64 encoding scheme. Listing 26
STRBUF	String Buffer	This refers to the use of "append" method to add strings in a sequence. StringBuilder is broadly similar to StringBuffer however, StringBuilder is faster under most implementation. There are many more methods associated with this class and should be considered for future investigations. Listing 27
CONCT	Concatenation	This refers to the use of the addition operator "+" to sequentially append string i.e. cipher algorithm, mode and padding, to each other. Listing 28
SEPRT	Separator	This refers to "/" which typically separates the cipher algorithm, mode and padding, used as a standalone value either as a string, identifier, static field class etc. Listing 29
OID	Object Identifier	These are numeric strings used to refer to cipher algorithms. Listing 30
STRING	String	This refers a series of characters i.e. string literals that refers to the cipher algorithm, mode and/or padding. Listing 31
EMPTY	Empty	This refers to an empty arguments i.e., cipher algorithms `Cipher.getInstance()`. Listing 32

(i.e., $\mathcal{F}_{14}$), which essentially turns hostname verification off. Most complex implementations of `HostnameVerifier` often call other methods to handle individual components of the verification. e.g., one method checks the host name format (IPV4, IPV6, URL), and another component uses elements of the X509Certificate, such as its getSubjectAltName, and compares that with the hostname or IP address.

C Restrictive Invocations – Code Snippets

```
Cipher.getInstance(String.format("%s/%s/%s", "RSA",
 "ECB", "PKCS1Padding"));
```

Listing 17: String Operation

```
Cipher.getInstance(Intrinsics.areEqual(this.
 sharedPreferences.getString("cipher.used",
 Build.VERSION.SDK_INT >= 23 ? "M" : "PREM"), "M") ?
 "RSA/ECB/OAEPWithSHA-1AndMGF1Padding" :
 "RSA/ECB/PKCS1Padding")
```

Listing 18: The use of nested ternary operators in `Cipher.getInstance(<parameter>)`

```
Cipher.getInstance(z ? "AES/GCM/NoPadding" :
 "AES/CBC/PKCS5Padding");
```

Listing 19: Ternary Operator

```
Cipher.getInstance(aVar.f13865d);
```

Listing 20: Enum Type

```
Cipher.getInstance(nativeGetString);
```

Listing 21: Identifier

```
Cipher.getInstance(this.mTransformation);
```

Listing 22: Referencing Current Object.

```
Cipher.getInstance(getCipherAlgorithm());
```

Listing 23: Method Call

```
Cipher.getInstance(SecurityConstants.AES_MODE);
```

Listing 24: Static Final Class.

```
Cipher.getInstance(requestTransformation(1));
```

Listing 25: Native Method Call.

```
Cipher.getInstance(new
 String(Base64.decode("REVTL0NCQy9QS0NTNVBhZGRpbmc=",
 2)));
```

Listing 26: Base64 for Decoder

Table 3: Flexible Taxonomy Labels

Label	Label Details	Description of Labels
ABS	Abstract Method	This is a method declared without a method body i.e. curly braces – "{}" and terminted with a semicolon.
NATIVD	Native Method Declaration	This refers to checkServerTrusted method defined in native code. It is distinctively identified by the use of keyword "native".
ILL	Throwing Illegal Argument Exception	This refers to throwing the Illegal Argument Exception, if certificate chain and auth type values are null or zero-length.
CEXP	Throwing Certificate Exception	This refers to throwing the Certificate Exception, if certificate chain is not trusted by the TrustManager.
NEXP	Throwing Null and Assertion Error	This refers to unconditionally throwing Null or Assertion Error only in the method body.
LOG	Logging Certificates	This refers logging certificates in the method body of checkServerTrusted method.
NATIVE	Native Method Call	This refers to the accessing methods or variables implemented in C/C++ and stored in native libraries
METHOD	Method Call	Used to indicate the presence of a *function* call that returns the verifies the certificate
THIS	Reference Current Object	Referencing the current object is using the "this" keyword to refer to the current object instance variable or method.
ISTRST	TrustStrategy → isTrusted Method	This refers to the use of an interface particularly a method named "isTrusted" which determines whether the certificate chain can be trusted without consulting the trust manager configured in the SSL context [26].
VAL	Checking Certificate Validity	This refers to one of the x509Certificate method used to check if certificate is currently valid i.e. not expired [2]
LEN	Checking Certificate Chain Length	This refers checking if certificate is zero-length.
NULL	Non-Null Value Check	This refers checking if certificate material is non null.
LIST	Using List Methods	Lists is often used to handle certificate materials, this refers to the use of list methods to i.e. add, remove etc while handling certificate.
CERFAC	Certificate Factory	This is used to generate certificate object and initializes it with the data read from an input stream [43].
VER	Verify Method	This is the use of one of X509Certificate methods which verifies that a certificate was signed using a private key that corresponds to public key specified as input [2]
STROP	String Operation	This refers to the use of any of the string methods such as `contains`, `equals`, etc.
AUTH	AuthType Value Check	This refers checking Auth Type string for type of algorithm used, checking for zero-length and/or null value.
ENCOD	getEncoded Method	This refers to the use "getEncoded" method associated with an abstract class for managing different types of certificates with common functionality i.e. X.509, PGP etc [42]
ARR	Using Array Methods	This refers to the use of arrays methods in handling or verifying certificates.
BIGINT	BigInteger	This is the use of BigInteger.for managing large integer numbers. It is often use to handle certificate materials such as public keys.
TMFAC	TrustManager Factory	This is acts a factory for trustmanagers which are responsible for handling trust materials used in establishing trust.
GETPUB	Certificate.getPublicKey	It is often used to return the public key of the root certificate. Used possibly for certificate validation purposes.
GETISR	Certificate.getIssuerDN (Deprecated)	This is used to return the issuer distinguished value which is the entity that signed and issued the certificate. This method is deprecated and replaced with "getIssuerX500Principal" [2].
GETSUB	Certificate.getISubjectDN (Deprecated)	This method returns the subject (subject distinguished name) value from the certificate. This method is also deprecated and replaced with "getSubjectX500Principal".
HASH	Hash Algorithm	This refers to the use of hash algorithms such as SHA1 and SHA256 for certificate validation purposes.
CERPAT	CertPathValidator	This is used for validating certificate chains or path. This is used to return a CertPathValidator object that implements the specified algorithm i.e. "PKIX"
PKIX	PKIXParameters	This is used to populates the set of most-trusted CAs from the trusted certificate entries contained in the specified KeyStore.It is often used to disable the revocation enable flag to enable faster validation.
CLIENT	Calling checkClientTrusted	This refers calling an empty checkClientTrusted from checkServerTrusted method for certificate validation.
EMPTY	Empty Method Body	This refers to a `checkServerTrusted` method with empty body i.e. no certificate validation logic implemented.

```
1 Cipher.getInstance(stringBuffer.toString());
```

Listing 27: String Buffer/String Builder

```
1 Cipher.getInstance("AES/ECB/" + padding);
```

Listing 28: Concatenation

```
1 Cipher.getInstance(str5 + RDMConstants.SLASH + str2 +
  RDMConstants.SLASH + str4, provider);
```

Listing 29: Separator.

```
Cipher.getInstance("1.2.840.113549.3.2");
```

Listing 30: Object Identifier.

```
Cipher.getInstance("Blowfish");
```

Listing 31: String.

```
AESCipher.getInstance();
```

Listing 32: Empty Argument

```
Cipher cipher = Cipher.getInstance(cipherAlgorithm);
----------
cipherAlgorithm =
Utils.base64DecodeAndXor("Iio+ASgjKE4/ZSIjXDMOCUoCDww=");
-----------
static String base64DecodeAndXor(String str) {
 return xorMessage(new
  String(Base64.getDecoder().decode(str)));
--------------
Note: xorMessage in Line 7 calls two more methods not
 shown
cipherAlgorithm = "AES/CBC/PKCS5Padding"
```

Listing 33: XOR and Base64 with up to 4 method calls

```
Cipher cipher = Cipher.getInstance(m7713c());
-------------------------
public static String m7713c() {
int[] iArr = {55, 49, 59, 54, 47, 45, 36, 43, 41};
StringBuilder sb = new StringBuilder();
for (int i2 = 8; i2 < 9 && i2 >= 0; i2 -= 3) {
 sb.append(Character.toChars(iArr[i2] + 24));
}
return sb.toString();
```

Listing 34: Extracting AES from byte array

```
new
 SecretKeySpec(Base64.decode("oik6PdDdMnOXemTbwvMn9de/h9
 lFnfBaCWbGMMZqqoSaQaqUOqjVGm5NqsmjcBI1x+sS9ugjB55HEJWR
 iFXYFw==", 2), "HmacSHA256")
```

Listing 35: Base64 Encoded Key

```
new
 SecretKeySpec("oejkdirztefhnvscxhdmdzedfotuabje".get
 Bytes("UTF-8"), "AES");
```

Listing 36: Direct Hard Coded Key

```
public void checkServerTrusted(X509Certificate[]
 x509CertificateArr, String str2) throws
 CertificateException {
try {
if(AdjustBridgeUtil.byte2HexFormatted(MessageDigest
 .getInstance("SHA1").digest(x509CertificateArr[0]
 .getEncoded()))
 .equalsIgnoreCase("7BCFF44099A35BC093BB48C5A6B9A5
16CDFDA0D1")) {
return;
}
```

Listing 37: SHA1 and getEncoded to check hard coded root cert.

```
private static final String KEY_AES = "AES";
private static final String KEY_CIPHER =
 "AES/GCM/NoPadding";
public static final String KEY_GCM =
 "OGEseetime201800";
private static String TAG = "AES2Utils";

algorithmStr_encode =
 "32Bi2A5oaH61xilScou92x9faAiOOSOBXmb0X/wqAijapt8K"
String AES_ECB_PADDING_encode =
 "32Bi2A5oaH6r4jpgI7+10Rwi6u+aWTgnrWUjLeHbiJK5";

Cipher.getInstance(AesGcmUtils.decode(algorithmStr_encode));
Cipher.getInstance(decode(AES_ECB_PADDING_encode);

--------------------------------------
public static String decode(String str) {
 try {
  return str.isEmpty() ? "" : new
   String(decrypt(getbase64ToBytes(str), KEY_GCM));
   //Calls the function below with
   Base64.getDecode().decode
 } catch (Exception e) {
  e.printStackTrace();
 return ;
}
}

--------------------------------
public static byte[] decrypt(byte[] bArr, String str)
 {
 try {
  SecretKeySpec secretKeySpec = new
   SecretKeySpec(getKey(KEY_GCM), KEY_AES);
  byte[] bytes = str.getBytes(StandardCharsets.UTF_8);
  Cipher cipher = Cipher.getInstance(KEY_CIPHER);
  cipher.init(2, secretKeySpec, new
   GCMParameterSpec(128, bytes));
 return cipher.doFinal(bArr);
 } catch (Exception e) {
  e.printStackTrace();
 return new byte[0];
}
}
```

Listing 38: Using GCM mode with Hard Coded Key to conceal use of ECB mode

```
Qhi = Qhi("AES/CBC/PKCS5Padding");
Cipher.getInstance(Qhi);
-------------
public static String Qhi(String str) {
 int[] iArr = new int[str.length()];
 iArr[4] = 6;
 iArr[5] = 1;
 iArr[6] = 1;
 return new String(Qhi(str.getBytes(), iArr));}
-------------
public static byte[] Qhi(byte[] bArr, int[] iArr) {
 if (bArr == null || bArr.length == 0 || iArr == null
  || iArr.length == 0) {
  return bArr;}
 byte[] bArr2 = new byte[bArr.length];
 for (int i = 0; i < bArr.length; i++) {
  bArr2[i] = (byte) (bArr[i] ^ iArr[i %
   iArr.length]);}
 return bArr2; }
```

Listing 39: Use of XOR to change CBC to ECB