Inferring Crypto API Rules from Code Changes

Rumen Paletov*
ETH Zurich, Switzerland
rumen.paletov@gmail.com

Petar Tsankov
ETH Zurich, Switzerland
ptsankov@inf.ethz.ch

Veselin Raychev
DeepCode AG, Switzerland
veselin@deepcode.ai

Martin Vechev
ETH Zurich, Switzerland
martin.vechev@inf.ethz.ch

Abstract

Creating and maintaining an up-to-date set of security rules that match misuses of crypto APIs is challenging, as crypto APIs constantly evolve over time with new cryptographic primitives and settings, making existing ones obsolete.

To address this challenge, we present a new approach to extract security fixes from thousands of code changes. Our approach consists of: *(i)* identifying code changes, which often capture security fixes, *(ii)* an abstraction that filters irrelevant code changes (such as refactorings), and *(iii)* a clustering analysis that reveals commonalities between semantic code changes and helps in eliciting security rules.

We applied our approach to the Java Crypto API and showed that it is effective: *(i)* our abstraction effectively filters non-semantic code changes (over 99% of all changes) without removing security fixes, and *(ii)* over 80% of the code changes are security fixes identifying security rules. Based on our results, we identified 13 rules, including new ones not supported by existing security checkers.

CCS Concepts • **Security and privacy → Systems security**; *Cryptanalysis and other attacks*; *Software security engineering*;

Keywords Security, Misuse of Cryptography, Learning

ACM Reference Format:
Rumen Paletov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Inferring Crypto API Rules from Code Changes. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3192366.3192403

*Currently employed by Twitter, work done while at ETH Zurich

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192403

1 Introduction

Many critical data breaches nowadays are caused by an incorrect use of crypto APIs. Developers often fail to understand and correctly configure cryptographic primitives, such as cryptographic ciphers, secret keys, and hash functions, leading to severe security vulnerabilities that can be abused by attackers [15, 24]. As a result, from 100 inspected Android applications, researchers discovered severe man-in-the-middle attacks in 41 of them and were able to gather a large variety of sensitive data [14]. As another data point, researchers have defined 6 common types of mistakes in using crypto APIs and found that a staggering percentage, 88%, out of thousands of analyzed Android applications have at least one of these mistakes [12].

To resolve this problem, we argue that developers must check their applications against an up-to-date and comprehensive list of security rules regarding potential misuses of crypto APIs. Unfortunately, creating and updating such a list can be quite challenging as security is a constantly moving target: crypto APIs evolve over time as security experts continue to discover new attacks against existing primitives. For example, researchers have recently discovered the first collision against the SHA-1 cryptographic hash function [30], and are now advising developers to shift to safer alternatives, such as SHA-256.

This Work: From Code Changes to API Usage Rules. In this paper, we propose a new approach for learning the correct usage of an API based on code changes, which are readily available in public repositories today. We show that code changes that fix security problems are more common than changes that introduce them, i.e. most problems were introduced in the initial implementation, not in a fix. The immediate benefit of this idea is that we can produce meaningful results even when most developers misuse an API (as it happens with the Java Crypto API). For instance, even if most developers use an outdated, less-secure cryptographic primitive (e.g. SHA-1), our approach can identify, based on a couple of code changes, that developers are switching to a new, more secure primitive (e.g. SHA-256).

Key challenge. Attempting to learn from code changes is difficult because many changes do not semantically affect how the API is used (e.g, they may be a syntactic re-factoring).

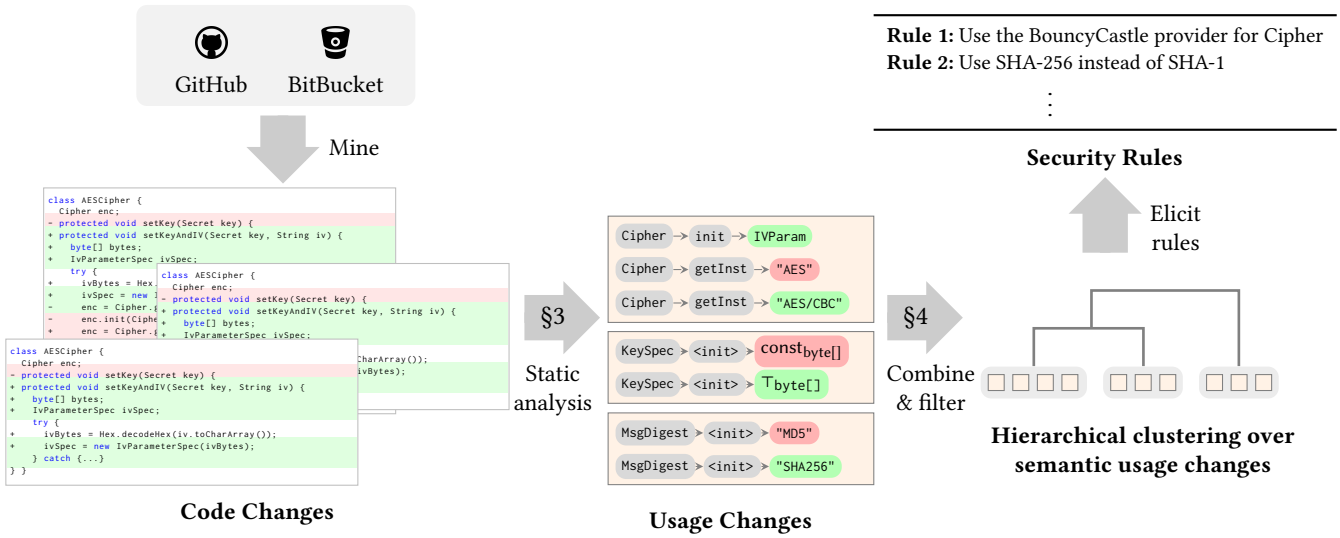


Figure 1. Overview of our approach for learning semantic usage changes and extracting security rules.

To address this challenge, we develop an abstraction tailored to crypto APIs that can capture relevant security properties. Our abstraction captures the semantic features of how a code change affects crypto API usage (e.g., how argument type changes affect the cryptographic mode), which enables filtering of unrelated or non-semantic changes.

Application to the Java Crypto API. We implemented an end-to-end system, called DiffCode, of our approach for learning semantic changes to the Java Crypto API. To demonstrate the effectiveness of DiffCode, we applied it to thousands of code changes collected from GitHub. DiffCode produces only few relevant changes that let us derive new security rules. Based on these, we created a new security checker for the Java Crypto API called CryptoChecker which has more rules than prior security checkers. For instance, a novel rule derived from data is to switch from the default Java provider to BouncyCastle. The reason is that BouncyCastle does not have a 128 bit key restriction [3].

Main Contributions. Our main contributions are:

- A new data-driven approach that learns rules from code changes where the learned rules capture the correct usage of an API (Section 2).
- An abstraction tailored to crypto APIs that captures the semantic structure of code changes while abstracting away syntactic details. This abstraction is essential to distilling thousands of concrete code changes into few semantically meaningful ones (Section 3).
- An end-to-end system, called DiffCode, for discovering relevant code changes. Our system consists of: (i) a lightweight AST-based program analysis that supports (partial) code snippets, (ii) abstraction of crypto API usage changes, and (iii) filtering combined with clustering analysis to ease users in inspecting relevant code changes (Sections 4-5).
- An extensive evaluation of DiffCode on the Java Crypto API. Using DiffCode, we identified 13 security rules, including several previously unknown ones (Section 6).

We remark that while we focus on crypto APIs, the approach is general and can be applied to other types of APIs.

2 Overview of Approach

We now present our approach to extracting information about API usages from thousands of code changes. At a high-level, our method is based on two key insights: Our first insight is to focus on code changes, which identify concrete fixes that developers have applied to the code. The old version of the program (before applying the change) often resembles incorrect (or, insecure) usage of the API. Our approach thus differs from existing statistical "Big Code"-type of approaches, which focus on discovering statistically common API usages (e.g., [27]). Such statistical approaches are bound to produce less meaningful results in settings where the majority of developers misuse the API, as is the case of crypto APIs. In contrast, our method can discover incorrect usage even when only few developers have applied a correct fix. Our second insight is to leverage program abstraction to derive meaningful, semantic information about code changes. This is necessary to avoid irrelevant syntactic code modifications, such as refactoring.

We depict the flow of our learning approach in Figure 1. We now briefly describe its main steps.

Step 1: Mining Code Changes. The first step consists of collecting code changes from open-source repositories.

Since we are usually interested in extracting usage changes for particular API classes, such as `Cipher` and `SecretKeySpec` from the Java Crypto API, we fetch only patches for classes that use the target API classes. We explain how we collected thousands of code changes for the Java Crypto API, which we used in our experiments, in Section 6.

Step 2: Abstract Usage Changes via Static Analysis. Program abstraction is a key element in our approach that enables us to distill semantic security fixes from thousands of code changes. Developers often commit patches that refactor the code, e.g. to improve readability and performance, without making semantic changes to the target API classes. Program abstraction can be used to discover that such syntactic modifications do not result in semantic changes to the program and how it uses the API.

DiffCode downloads both the old and the new versions of the program and statically analyzes each version. It first discovers all different usages of a target API and extracts semantic features about each usage. These features capture which methods are invoked on objects of the API as well as information about the arguments passed to these methods. A usage change is then identified by the change in these features. For instance, suppose that in the old version the program creates an object of type `Cipher` by calling `getInstance()` and passing `"AES"` as an argument. Further, suppose that in the new version the program creates the same object with a call to `getInstance()` with arguments `"AES/CBC"` and an initialization vector object. DiffCode would detect this as a semantic change: the program changes the mode of the `AES` cipher from default Electronic Code Book mode (ECB) to the more secure Chain-Block Cipher mode (CBC). We describe the program abstraction we use in Section 3 and how it is derived using static analysis in Section 5.1.

Step 3: Filter and Cluster Usage Changes. The usage changes derived from each project are collected and processed together. Thanks to the abstraction, DiffCode filters out usage changes that are not semantic fixes. For example, if features are neither added nor removed for a particular API usage, then the usage code is likely a refactoring and is thus filtered. We apply additional filters to remove duplicate usage changes and changes that either only add or remove features, as these often correspond to adding a new usage of the API or removing an existing one.

In the beginning, DiffCode starts with tens of thousands of code changes and after the filtering step, there are only 186 remaining usage changes. Yet, we checked that this filtering step does not remove previously known security-related rules. Then, DiffCode uses a classic hierarchical clustering algorithm on these 186 changes (Section 4) and produces clusters that correspond to security-related rules. At this stage, we manually inspected the clusters and devised security checks that we encoded into a tool called CryptoChecker.

The last step in DiffCode is manual for several reasons. First, we did not focus on automating this last step as it involves inspecting only tens of clusters of changes. Second, we manually inspect, document, and explain the derived rules to users. Finally, we remove false positives that introduce security problems as opposed to fix it – in fact, these are easy to filter out, even automatically, because there are fewer commits in clusters that introduce problems than in clusters that fix them.

Overall, we derived 13 security rules, some of which are new rules. The new rules are currently not included in existing security checkers for crypto API misuse. Our rules are described in Section 6.

3 Abstraction for API Changes

We now present our abstraction for representing the semantic structure of security fixes applied to crypto APIs. We first present the terminology, then discuss an abstraction that given (a single version of) a program returns a set of API usage. Finally, given two program versions, we show how to leverage the abstraction in order to capture *API usage changes* (which may correspond to actual fixes). In the sections that follow we show how to leverage this abstraction for learning security rules.

3.1 Example

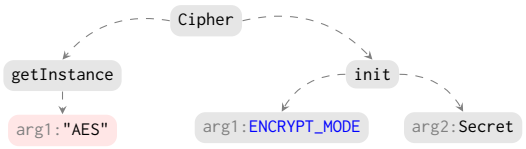
We first present an example which we later use to illustrate our definitions. In Figure 2(a) we show the code patch for a Java class called `AESCipher`. The code lines removed from the old version are shown in red (and marked with -) and the added lines are shown in green (and marked with +). This class creates two objects of type `Cipher`, `enc` and `dec`, which are used for encryption and, respectively, decryption.

Old Version. The old version of `AESCipher` creates the two objects `enc` and `dec` using the method `getInstance` with the string `"AES"` passed as an argument. The instance `enc` is then initialized using `init` with arguments `Cipher.ENCRYPT_MODE`, an integer constant defined in the `Cipher` API, and `key`, which represents the symmetric key to be used for encryption. The object `dec` is also initialized with `key`, but this time the class uses the `Cipher.DECRYPT_MODE` constant.

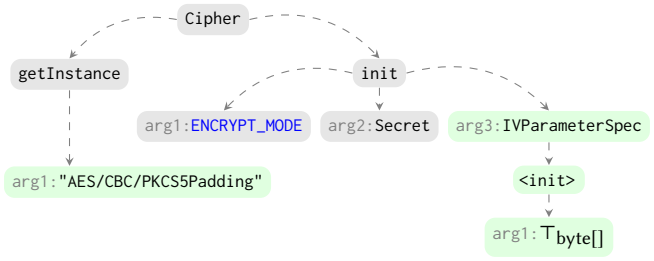
New Version. In the new version, the developer changes the signature of `setKey`, which now also takes as argument the object `iv` of type `String`. Further, the objects `enc` and `dec` are initialized using the string `"AES/CBC/PKCS5Padding"` (instead of `"AES"`). With this change, the developer explicitly expresses that the two ciphers must use the AES cipher in Chain Block Cipher mode (CBC) as well as the `PKCS5` padding scheme. When the two `Cipher` objects are initialized, the developer passes as argument the object `ivSpec` of type `IVParameterSpec` to define the initialization vector that the ciphers must use for the first block they process. The

```
1  class AESCipher {
2    Cipher enc, dec;
3  - final String algorithm = "AES";
4  + final String algorithm = "AES/CBC/PKCS5Padding";
5
6  - protected void setKey(Secret key) {
7  + protected void setKeyAndIV(Secret key, String iv) {
8  +   byte[] bytes;
9  +   IvParameterSpec ivSpec;
10     try {
11 +     ivBytes = Hex.decodeHex(iv.toCharArray());
12 +     ivSpec = new IvParameterSpec(ivBytes);
13       enc = Cipher.getInstance(algorithm);
14 -     enc.init(Cipher.ENCRYPT_MODE, key);
15 +     enc.init(Cipher.ENCRYPT_MODE, key, ivSpec);
16       dec = Cipher.getInstance(algorithm);
17 -     dec.init(Cipher.DECRYPT_MODE, key);
18 +     dec.init(Cipher.DECRYPT_MODE, key, ivSpec);
19     } catch {...}
20   }
21 }
```

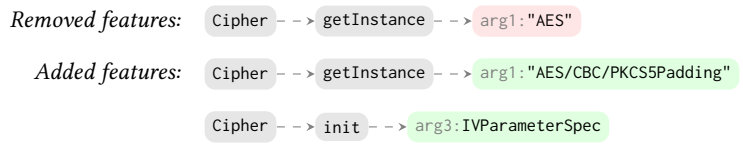
(a) Code changes to two objects (`enc` and `dec`) of type `Cipher`



(b) Usage DAG of the object `enc` before the change



(c) Usage DAG of object `enc` after the change



(d) Removed (red) and added (green) features that capture the usage change of object `enc`.

Figure 2. Code changes to two objects of the type `Cipher` and the usages change derived for object `enc`.

developer also adds lines 11 and 12 to initialize the `ivSpec` object using the string `iv` passed as argument to the method.

The Need for Abstraction. If we consider this example purely syntactically, the lines that call `Cipher.getInstance` remain unchanged. At the same time, most of the syntactic changes are related to renaming the `setKey` method and introducing an extra parameter. However, if we perform the right program analysis before comparing the two versions, we can abstract the semantically relevant changes for each of the `Cypher` objects and concisely capture the semantics of the change.

In later sections, we explain how DiffCode learns from the example in Figure 2(a) by illustrating the steps on Figures 2(b), 2(c), and 2(d).

3.2 Basic Notation and Terminology

Before presenting our abstraction, we describe our notation and terminology and define what we mean by API usage.

Types and Methods. We restrict our attention to crypto APIs for languages such as Java that support base types (e.g., `int`, `byte`, `int[]`, `byte[]`) and object types which are stored in the heap. We consider an API that defines a set of types $Types = \{t_1, \ldots, t_n\}$. For instance, the Java Crypto API defines the type `Cipher`, a cryptographic cipher used for encryption and decryption.

A method signature is given by $m([t_0], t_1, \ldots, t_k) : t_{ret}$ where t_0 is the type of the object on which the method is invoked (the `this` object), k is the method's arity, each t_i is the type of the ith argument, and t_{ret} is the type of object/value returned by the method. Note that t_0 is defined only for non static methods. We write *Methods* to denote the set of all methods.

For a given type t, $Methods_t \subseteq Methods$ denotes the set of all methods that *(i)* accept an instance of type t as an argument or *(ii)* create a new instance of type t. For example, the set $Methods_{\texttt{IVParameterSpec}}$ contains the method `Cipher.init(int, Key, AlgorithmParameterSpec)`, as it accepts objects of type `IVParameterSpec` as the third argument. Further, it contains `IVParameterSpec.<init>(byte[])`, which creates a new instance of type `IVParameterSpec`. Note that in addition to constructor methods, $Methods_t$ may also contain factory methods. For example, $Methods_{\texttt{Cipher}}$ contains factory method `Cipher.getInstance(String):Cipher`.

Program State. We assume standard program semantics of an object-oriented language. A program state $\sigma \in States$

is a tuple $\sigma = (objs, \eta, \Delta)$ with:

$$\begin{array}{lcl} objs & \subseteq & Objs \\ Vals & = & objs \cup BaseValues \\ \eta & \in & Heaps\colon objs \times Fields \rightarrow Vals \\ \Delta & \in & Stores\colon Vars \rightarrow Vals \\ States & = & \mathcal{P}(Objs) \times Heaps \times Stores \end{array}$$

where *Objs* is the set of all possible objects, *BaseValues* is the set of all values of base types (such as values of type `int` and `byte`), *Fields* is the set of fields, and *Vars* is the set of all local variables. A program state $\sigma = (objs, \eta, \Delta)$ tracks: *(i)* the set of allocated objects *objs*, *(ii)* the state of the heap η that maps the fields of allocated objects to values (either an allocated object or a value of a base type), and *(iii)* the state of local variables which store values.

Concrete API Usages. A standard way to define a *concrete usage* of a given type t is to collect the set of all method calls to an object of type t together with the program states associated at each method call; cf. [27]. Note that a program usually defines multiple objects of the same type t that are then used in different ways, resulting in multiple concrete usages of type t.

We define the concrete usages as the map:

$$CUses\colon Objs \rightarrow \mathcal{P}(Methods \times States)\ .$$

That is, for a given object $o \in Objs$ of type t, the set of pairs $CUses(o) = \{(m_1, \sigma_1), \ldots, (m_n, \sigma_n)\}$ contains the constructor/factory method $m_i \in Methods_t$ used to create o as well as methods $m_j \in Methods_t$ that take as argument the object o. A method m may be invoked multiple times with object o at different program states, resulting in multiple pairs $(m, \sigma_1), \ldots, (m, \sigma_k)$ in $CUses(o)$.

We note that the concrete usages *CUses* for a given program will typically not be computable in practice as the program may allocate an unbounded set of objects and may have an unbounded number of states. Our abstraction, defined below, allows us to capture these unbounded sets with a finite set of *abstract usages*.

3.3 Abstraction of API Usage

We now present our abstraction which we use to capture the usage of a particular API type. Our abstraction consists of: *(i)* a heap abstraction, to represent the unbounded set of concrete objects with finitely many allocation sites, *(ii)* base-types abstraction, and *(iii)* per-object Cartesian abstraction that keeps track of method calls and abstract states associated with the abstract objects. We define this abstraction below. We will explain how to apply it to programs using static analysis in Section 5.1.

Heap Abstraction. Since a program may instantiate a potentially unbounded number of objects of a given type, we use a per-allocation-site abstraction. That is, each constructor/factory method, such as `Cipher.getInstance(String)`, results in one abstract object identified by the statement's label. We denote by *AObjs* the set of abstract objects. We use $\top_{obj} \in AObjs$ to represent that the allocation of the abstract object is unknown; e.g., the allocation of the method parameter `key` is not defined in Figure 2(a).

Base type	Abstract Domain
`int`	$Ints(P) \cup \{\top_{int}\}$
`int[]`	$IntArrays(P) \cup \{\top_{int[]}\}$
`string`	$Strs(P) \cup \{\top_{str}\}$
`string[]`	$StrArrays(P) \cup \{\top_{str[]}\}$
`byte`	$\{\texttt{const}_{byte}, \top_{byte}\}$
`byte[]`	$\{\texttt{const}_{byte[]}, \top_{byte[]}\}$

Figure 3. Abstract base-type values for a program P.

Base-types Abstraction. In addition to abstracting objects, we also abstract the base-type values as they also range over unbounded sets. In Figure 3 we summarize the abstract domains that we use. Given a program P, we write $Ints(P)$ to denote the set of integer constants that appear in P (e.g., `0` $\in Ints(P)$ if there is a statement `int x = 0` in P), and $IntArrays(P)$ to denote the set of integer array constants that appear in P (extracted from statements such as `int[] arr = {0,1,2,3}`). Integer variables and fields are assigned *(i)* an integer constant from $Ints(P)$ or *(ii)* the symbol $\top_{int}$, which represents the set of all integers. Similarly, we also abstract strings and string arrays. We designed our abstraction to keep the values for integer and string constants as these often represent configuration parameters (e.g. `Cipher.ENCRYPT_MODE`) and configuration strings (e.g. `"AES/CBC/NoPadding"`).

Byte values are abstracted to $\texttt{const}_{byte}$ (to represent constant byte values) and $\top_{byte}$ (to represent non-constant byte values). Byte arrays are similarly abstracted to $\texttt{const}_{byte[]}$ and $\top_{byte[]}$. We remark that we represent constant byte arrays as $\texttt{const}_{byte[]}$ to abstract program-specific values, such as hard-coded keys and initialization vectors.

We note that our abstraction is tailored to crypto APIs. To precisely abstract uses of other APIs, one may choose a different abstraction (e.g., Interval or Polyhedra numerical domains [10]).

Abstract State. Let $AVals = AObjs \cup ABaseValues$ be the set of abstract objects and abstract base-type values, where *ABaseValues* is the union of all abstract domains derived from P. An abstract state $\sigma^a = (objs^a, \eta^a, \Delta^a)$ consists of a set of abstract objects $objs^a \subseteq AObjs$, an abstract heap $\eta^a\colon AObjs \times Fields \rightarrow AVals$, and abstract state of local variables $\Delta^a\colon Vars \rightarrow AVals$. We denote by *AStates* the set of all abstract states.

Abstract API Usage. We lift our notion of concrete usages to abstract usages, denoted by *AUses*, where instead of tracking the usage of each object we track the usage of abstract objects. Further, instead of collecting the concrete states at method calls we collect abstract states. Formally, abstract usages are captured with a map:

$$AUses\colon AObjs \to \mathcal{P}(Methods \times AStates)\ .$$

That is, for a given abstract object $o^a \in AObjs$ of type t, we obtain a set $AUses(o^a) = \{(m_1, \sigma_1^a), \ldots, (m_k, \sigma_k^a)\}$ where $m_i \in Methods_t$ is a method and σ_i^a is an abstract state. Each $AUses(o^a)$ defines one abstract usage, while together all abstract objects in a program define the set of all abstract usages. We note that since there are finitely many abstract objects, methods, and abstract states, the abstract usages for a given program are also finitely many.

3.4 Abstract Usages as Directed Acyclic Graphs

Given the abstract usages defined by *AUses* and an abstract object o^a, we construct a rooted directed acyclic graph (DAG) $G = (N, E, r)$ with nodes

$$N \subseteq (Methods \times AStates) \cup (\mathbb{N} \times AVals)\ ,$$

edges $E \subseteq N \times N$, and root $r = (0, o^a) \in N$. Each node in the DAG is either a pair (m, σ^a) of a method m and an abstract state σ^a or a pair (i, a) where $i \in \mathbb{N}$ represents an argument index and $a \in AVals$ is an abstract value (i.e., an abstract object or base-type value).

We label the nodes in the graph as follows. A node (m, σ^a) is labeled by the signature of m. A node (i, a) is labeled by (i, a) if a is an abstract base-type value (e.g., $\top_{\text{int}}$); otherwise, a is an abstract object and the node is labeled by $(i, type(a))$, where $type(a)$ returns the type of a (e.g., `Cipher`).

Examples of these DAGs are given in Figures 2(b) and 2(c). We depict node labels as `arg1: AES` instead of $(1, \texttt{AES})$ to emphasize that 1 represent an argument index. Further, we omit the index 0 in root node labels (as roots always have index 0). Below we explain how these DAGs are constructed.

Constructing a DAG. To construct the graph for an abstract object o^a, we first add the root $(0, o^a)$. Then, starting from the root, the tree is iteratively constructed by performing the following two steps on each node (i, abs):

1. For each $(m, \sigma^a) \in AUses(abs)$, we add a node (m, σ^a) and an edge $((i, abs), (m, \sigma^a))$.
2. For each node (m, σ^a) created in step (1), we add up to k children where k is the arity of m. First, for each parameter p_i of m, we add a node (i, abs_i), where $abs_i = \Delta^a(p_i)$. Then, we add an edge $((m, \sigma^a), (i, abs_i))$ if it does not introduce a cycle in the graph.

The above steps are iteratively performed in a breath-first manner by first expanding the root node (depth 0 of the rooted DAG). Then, we process the nodes (i, abs) at depth 2 of the DAG where $abs \in AObjs$ is an abstract object such that $abs \neq \top_{\text{obj}}$. Note that we skip the nodes at depth 1 as they contain only method nodes. We continue this process until a fixed depth n (in our experiments, we set n to 5).

Example. To illustrate the graph construction we refer to the example in Figure 2. The code after the change (green and white lines in Figure 2(a)) has two abstraction objects of type `Cipher` — one allocated at line 13 and another one at line 16. In Figure 2(c) we depict the graph constructed for the abstract object l_{13} (i.e., the `enc` object). The root node of the graph is $(0, l_{13})$, and it is labeled by $(0, \texttt{Cipher})$ because the type of l_{13} is `Cipher`. The abstract usage of l_{13} is given by

$$AUses(l_{13}) = \{(\texttt{getInstance}, \sigma_{13}^a), (\texttt{init}, \sigma_{15}^a)\}\ .$$

The root node therefore has two children nodes labeled by `getInstance` and `init`, respectively. Node `getInstance` has one child $(1, \texttt{AES/CBC/PKCS5Padding})$ because

$$\Delta_{l_{13}}^a(\texttt{algorithm}) = \texttt{AES/CBC/PKCS5Padding}\ .$$

Node `init` has three children. The first one is $(1, \texttt{ENCRYPT\_MODE})$, where `ENCRYPT_MODE` is an integer constant defined in `Cipher`. The second child is $(2, \top_{\text{obj}})$ and is labaled by $(1, \texttt{Secret})$. This child has no further children as $\Delta_{l_{13}}^a(\texttt{key}) = \top_{\text{obj}}$ (the allocation of object `key` is not defined in the code). Finally, the third child is $(3, l_{12})$ and is labeled by $(3, \texttt{IVParameterSpec})$. The abstract object l_{12} is recursively expanded with the constructor method `<init>` and it's argument $\top_{\text{byte[]}}$.

3.5 From DAGs to Usage Changes

To capture the semantic meaning of a code change with respect to a type t, we derive the abstract usages $AUses_1$ and $AUses_2$ for the old and, respectively, the new version of the program. We proceed in three steps, as depicted in Figure 4. First, for each version, we derive all rooted DAGs for all abstract objects of type t, as explained in Section 3.4. We note that we may obtain multiple DAGs for each version (determined by the number of allocation sites of objects of type t in a version). Second, we pair the DAGs of the old version with those of the new version based on a distance metric (defined below) that captures the similarity between two DAGs. Finally, given a pair of DAGs, we derive features that describe the semantic change between the two usages. We refer to these features as *usage change*. The result of the three steps above is a set of usage changes.

Distance Between DAGs. We first define a metric that reflects the distance between two DAGs $G_1 = (N_1, E_1, r_1)$ and $G_2 = (N_2, E_2, r_2)$. The distance between the DAGs is given by a intersection-over-union measure over the sets of nodes:

$$dist(G_1, G_2) = 1 - \frac{|N_1 \cap N_2|}{|N_1 \cup N_2|}\ .$$

This measure reflects the change in terms of nodes in the DAGs that differ in the two graphs while also respecting

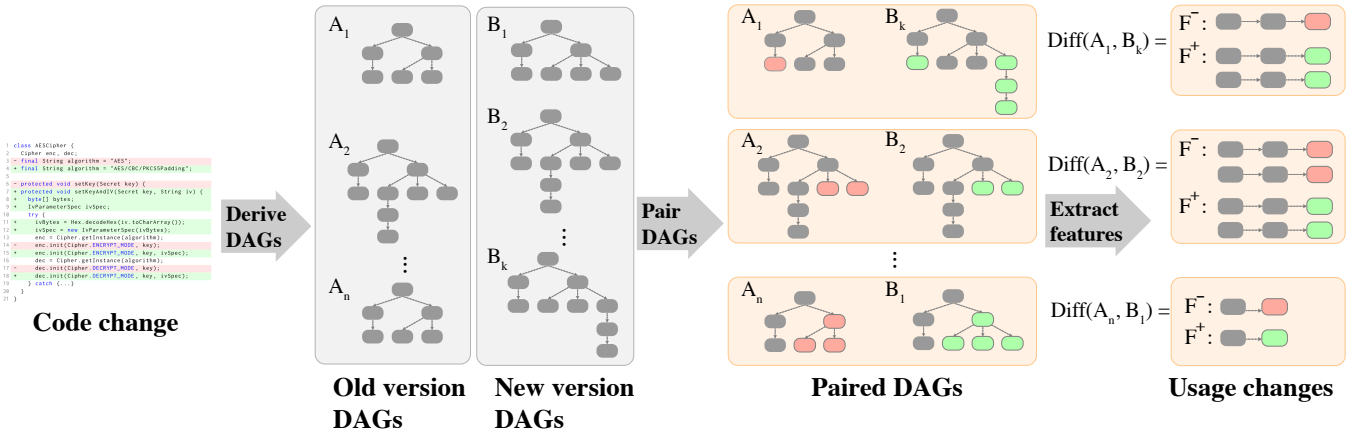


Figure 4. Step-by-step derivation of usage changes from a code change defining two versions of the program

the edges. For instance, for the DAGs G_1 and G_2 depicted in Figures 2(b) and 2(c), respectively, we get $dist(G_1, G_2) = \frac{1}{2}$.

Pairing DAGs. Suppose that the DAGs derived from the old version are $V_{\text{old}} = \{A_1, \ldots, A_n\}$ and those derived from the new version are $V_{\text{new}} = \{B_1, \ldots, B_k\}$; we depict these as old/new version DAGs in Figure 4. For simplicity, we assume that $|V_{\text{old}}| = |V_{\text{new}}|$, i.e. the two versions have an equal number of DAGs; if this is not the case, we extend the version with fewer DAGs with DAGs of the form $G = (\{r\}, \emptyset, r)$, which only contain a root node r labeled with the type t. We solve a maximum matching problem to map the DAGs in V_{old} to unique DAGs in V_{new} (and vice versa), such that we minimize the sum of the distance metrics of the paired graphs. The mapping is a bijection $m \subseteq V_{\text{old}} \times V_{\text{new}}$ whenever $|V_{\text{old}}| = |V_{\text{new}}|$.

Formally, let M be the set of all possible mappings. The distance for a given mapping $m \in M$ is given by:

$$mdist(m) = \sum_{(G_1, G_2) \in m} dist(G_1, G_2) \ .$$

To pair the DAGs in the two versions we find a minimum distance mapping according to $mdist(m)$. Note that there may be multiple such mappings.

For the example provided in Figure 4, the mapping produces the pairs (A_1, B_k), (A_2, B_2), and so forth. We color the nodes in green/red to emphasize which nodes in the paired DAGs are different.

From DAG Pairs to Usage Changes. We use the following notation to define the derivation of usage changes. Let $G = (N, E, r)$ be a rooted DAG. Given two paths p and p', we write $p \prec p'$ if p is a strict prefix of p', i.e. the length of p is strictly smaller than that of p' and p is a prefix of p'. For a set of paths P, we define

$$Shortest(P) = \{p \in P \mid \neg \exists p' \in P.\ p' \prec p\} \ .$$

That is, $Shortest(P)$ contains a path p if and only if no other path is a strict prefix of p. For example, for the set of paths

$$P = \{a \rightarrow b, a \rightarrow b \rightarrow c, b \rightarrow c\} \ ,$$

we get $Shortest(P) = \{a \rightarrow b, b \rightarrow c\}$.

Given two DAGs G_1 and G_2, we define the *shortest-removed paths* of G_1 and G_2, denoted by $Removed(G_1, G_2)$, as:

$$Removed(G_1, G_2) = Shortest(Paths(G_1) \setminus Paths(G_2)) \ .$$

That is, $Removed(G_1, G_2)$ contains the shortest prefixes in G_1 that are not in G_2.

We define the *usage change* between two DAGs G_1 and G_2 as a pair $Diff(G_1, G_2) = (F^-, F^+)$ where $F^- = Removed(G_1, G_2)$ and $F^+ = Removed(G_2, G_1)$. That is, the set of paths F^- contains the shortest prefixes removed from G_1 while F^+ contains those that are added to G_2. In Figure 2(d) we show in detail the usage change derived from the DAGs depicted in Figures 2(b) and 2(c).

4 Clustering Semantic Usage Changes

In this section, we describe our approach for filtering and clustering the obtained usage changes, that is, the output of Figure 4 described earlier. Our filters will aim to eliminate any irrelevant, non-semantic usage changes. Then, the remaining semantic usage changes will be clustered so to ease users in inspecting them and eliciting security rules.

4.1 Extract Usage Changes

The input to the first step is a set of code changes. Given this input, we derive the usage changes from each code modification, as described in Section 3. Note that each code change results in a set of usage changes because the old and new versions of the program may instantiate multiple objects of the same type and use them differently in the code.

4.2 Filter Uninteresting Usage Changes

The goal of this procedure is, given the large list of usage changes, to filter out the ones that are not relevant for deriving security rules. Uninteresting changes either do not affect crypto APIs, refactor crypto API calls, or introduce/delete code (as opposed to fixing an error). The input of the filtering procedure is a list of usage changes. Each usage change is a pair (F^-, F^+) of paths where F^- contains the features that are removed from the old version and F^+ contains the features added to the new version. We filter out a usage change (F^-, F^+) if one of the following conditions hold:

- **No-changes (f_{same}):** Both F^- and F^+ are empty sets. This condition indicates that, with respect to our abstraction, the API usage is identical in both the old and the new version of the program. This means that there is no actual semantic change of API usage. This filter removes the majority of the changes.
- **No-removals (f_{add}):** F^- is the empty set. This condition typically indicates that a usage of type t was added to the code. We do not use such changes since they simply say that the crypto API was introduced.
- **No-additions (f_{rem}):** F^+ is the empty set. This condition indicates that a usage of type t was removed for the code.
- **No-duplicates (f_{dup}):** There is another usage change (F'^-, F'^+) in the set such that $F^- = F'^-$ and $F^+ = F'^+$. This condition indicates that there is an identical usage change in the set of usage changes.

To see the effect of each filter, we run them in turn and report the number of remaining usage changes at stages after each filter. We remark that f_{add} and f_{rem} together subsume f_{same}, but we still consider f_{same} separately to report the number of changes that do not affect crypto APIs (shown later in Figure 6).

4.3 Cluster Usage Changes

After we obtain the filtered semantic usage changes, we cluster them together to report insights about how developers fix crypto APIs. Clustering is useful, because multiple similar changes would indicate a common misconception or a common fix regarding the API. To perform this step, we use classic clustering algorithms based on the same features used in the previous steps.

We now define a metric that captures the distance between a pair of usage changes. Our metric compares the features that the usage changes add and remove from the new and, respectively, the old version. We first define a measure of distance between two paths and then lift this measure to compare usage changes.

Distance Between Two Paths. We use the following notation. Given a path $p = l_0 \rightarrow \ldots \rightarrow l_n$ and two indices $0 \leq i \leq j \leq n$, we write $p[i]$ for l_i, $p[i : j]$ for the path $l_i \rightarrow \ldots \rightarrow l_j$. Given two paths p_1 and p_2, we denote by $commonPrefix(p_1, p_2)$ the length of the longest prefix of p_1 and p_2. That is, $commonPrefix(p_1, p_2)$ returns the index j if and only if $p_1[0 : j] = p_2[0 : j]$ and $p_1[0 : j+1] \neq p_2[0 : j+1]$. Further, we write $lev(l, l')$ to denote the Levenshtein distance [8] between the two node labels l and l', which captures the smallest number of modifications—insertions, deletions, and substitutions—required to change one label into the other. As units for modifications, we use characters for strings, while integers, bytes (which are abstracted to $\mathtt{const}_{\mathsf{byte}}$ and $\top_{\mathsf{byte}}$), and method names are treated as single units. For example, it takes 1 modification (more precisely, 1 substitution) to change any method signature to a different one. We define the Levenshtein similarity ratio between two labels l and l' as:

$$LSR(l, l') = 1 - \frac{lev(l, l')}{max(|l|, |l'|)} .$$

The distance between paths p_1 and p_2 is $pathDist(p_1, p_2) = 0$ if p_1 is identical to p_2, and otherwise it is defined as:

$$pathDist(p_1, p_2) = 1 - \frac{j + LSR(p_1[j+1], p_2[j+1])}{max(|p_1|, |p_2|)} ,$$

where $j = commonPrefix(p_1, p_2)$ is the length of the longest prefix of p_1 and p_2. In the numerator, we take the size of the common prefix and add the result of the Levenshtein similarity ratio between the remaining suffixes of p_1 and p_2. In the denominator, we have the largest length of both paths.

Distance Between Two Usage Changes. We define the distance between two sets of paths F_1 and F_2, $pathsDist(F_1, F_2)$, as the smallest distance that we obtain by first matching the paths in both sets and then summing their pair-wise path distance. Given two usage changes $C_1 = (F_1^-, F_1^+)$ and $C_2 = (F_2^-, F_2^+)$, we define the distance to be the average over the two distances between F_1^- and F_2^- and between F_1^+ and F_2^+:

$$usageDist(C_1, C_2) = \frac{pathsDist(F_1^-, F_2^-) + pathsDist(F_1^+, F_2^+)}{2} .$$

The distance metric $usageDist(C_1, C_2)$ allows us to compare how semantically similar two usage changes are.

Hierarchical Clustering. We use an agglomerative hierarchical clustering algorithm to group similar usage changes and structure them in a tree. Agglomerative clustering first introduces a leaf node in the tree for each usage change and then merges the two closest clusters according to the distance metric. As a distance metric we use the distance between two usage changes. The distance between clusters (also known as the linkage) is used to merge clusters of usage changes (higher in the tree). We use complete linkage where the distance between two clusters X and Y is given by:

$$clusterDist(X, Y) = max_{C_1 \in X, C_2 \in Y}\ usageDist(C_1, C_2) .$$

5 The DiffCode System

In this section, we first present DiffCode, a system which implements the abstraction for usage changes described in Section 3 together with the filtering and clustering procedures presented in Section 4. In the following section, we show how DiffCode can be used to infer semantic usage changes to the Java Crypto API, based on thousands of code modifications we have collected from GitHub.

5.1 System Overview

DiffCode is a new end-to-end system that implements the usage changes abstraction for Java APIs. Our implementation is in Python and spans roughly 7K lines of code.

DiffCode takes as input a set of program pairs (old and new versions of a Java program) together with a target API class, and outputs a list of semantic usage changes of the target API class together with a clustering diagram of these usage changes. The main component of DiffCode is a lightweight AST-based program analyzer, described below.

AST-based Program Analyzer. DiffCode uses a custom AST-based analyzer since many of the program versions provided as input are partial programs, such as library code without an explicit entry point and code snippets, that cannot be easily compiled. Further, we opted for an efficient and scalable analyzer that avoids heavy-weight static analysis, such as SPARK's points-to analysis [19].

Our program analyzer takes as input a GitHub username, project name, and commit ID, which together identify a Java project version. Additionally, it takes the target API class for which we want to discover usage changes. Our analyzer first finds all allocation sites of the target class (which correspond to abstract objects). For each allocation site, located in some method m, it finds the program's entry methods that can lead to executions that call method m. Note that there may be multiple such entry methods (i.e., other than `main`) if the code is a partial program or a library. For each entry method, the analyzer performs a forward execution of all relevant operations, such as object allocations and field accesses, to track the set of possible values that can be assigned to fields and variables. At each branching point (e.g., an `if` statement), the analyzer forks the execution into two executions and analyzes them independently. The result is a set of executions with derived abstract states at each method call. Each execution is used to derive a DAG as described in Section 3.4. Our analysis is inter-procedural, and currently does not support deep inheritance hierarchies and virtual functions.

6 Case Study: Java Crypto API

We now describe a case study in which we use Java projects collected from GitHub to learn semantic changes of the Java Crypto API [26]. Extracting security fixes for the Java Crypto API is relevant because: *(i)* developers often misuse this API [12, 22, 24], and *(ii)* crypto modes become obsolete over time as security experts discover attacks, and thus new, up-to-date security rules for the Java Crypto API are needed (e.g, see [30]).

API Class	Description
`Cipher`	A cryptographic cipher used for encryption and decryption
`IvParameterSpec`	An initialization vector (IV) used in ciphers that operate in feedback mode (e.g. CBC)
`MessageDigest`	An engine class designed to provide the functionality of cryptographically secure message digests such as SHA-1 or MD5
`SecretKeySpec`	A constructor for secret keys a byte array
`SecureRandom`	An engine class that provides the functionality of a Random Number Generator (RNG)
`PBEKeySpec`	A user-chosen password that can be used with password-based encryption

Figure 5. Target classes for learning semantic usage changes.

More concretely, in this section we address the following research questions. First, we investigate the effectiveness of our abstraction and filters on distilling semantic code changes, and whether these are security fixes or buggy changes. Second, we report on our experience in clustering code changes and eliciting security rules. Finally, we investigate the relevance of the elicited security rules by applying these on Java projects collected from GitHub.

6.1 Experimental Setup

Data Set. To obtain our training dataset, we scanned over 30,000 popular GitHub repositories. We selected the master branches of projects that use the Java Crypto API and have at least 30 commits. We duplicated projects in case the commit history has a common prefix. We remark that our selection method helps DiffCode ignore toy projects that are unlikely to contain interesting code changes. Indeed, our method selected some of the most starred Java projects excluding forks. For training, our selection led to 461 Java projects from 397 distinct users.

We cloned these repositories and traversed the master branch commits of each repository. We consider 6 target API classes in our case study; see Figure 5. For each commit that changes at least one target class, we fetched the versions before and after the commit. Using this procedure, we collected a total of 11,551 code changes (i.e., pairs of programs) for all of the target classes.

Target API Class	Usage Changes	After filtering stage f_{same}	f_{add}	f_{rem}	f_{dup}
`Cipher`	15829	419	204	116	75
`IVParameterSpec`	4967	58	24	12	11
`MessageDigest`	8277	116	78	27	17
`SecretKeySpec`	15543	226	120	55	45
`SecureRandom`	26008	309	131	26	21
`PBEKeySpec`	1549	29	21	17	17

Figure 6. Usage changes per target API class after abstraction and filtering. The actual commits are available at http://diffcode.ethz.ch

6.2 Effectiveness of Abstraction and Filtering

In Figure 6, we give the number of usage changes for each target API class and then show the effectiveness of our abstraction and filters. The second column gives the total number of usage changes, and the four columns to the right show the number of usage changes that remain after each filtering stage (see Section 4.2 for the list of stages). For example, the filter f_{same}, which removes changes that do not affect the target class, reduces the number of changes by more than an order of magnitude (e.g. 419 down from 15,829 for class `Cipher`). Filtering out changes that add or remove API calls of the target class, as well as removing duplicates further reduces the number of changes by another order of magnitude. At the end, the number of remaining changes makes the follow-up manual inspection feasible (e.g. only 75 changes for the `Cipher` class).

Security Fixes vs Buggy Changes. Next, we address two questions: *(i)* whether the collected code changes represent security fixes or buggy changes and *(ii)* whether the filters keep these while removing the non-semantic code changes. To distinguish between security fixes and buggy changes, we encoded five security rules supported in CryptoLint [12], a security checker for crypto APIs. We denote these rules CL1-CL5. For instance, the first rule CL1 states: "Do not use ECB mode for encryption".

For each change, we check whether a rule triggers in the old version (before applying the change) and whether it triggers in the new version (after applying the change). Based on the result, we classify each code change as a: *(i) security fix*, if a rule triggers in the old version but not in the new version, *(ii) buggy change*, if a rule triggers in the new version but not in the old version, and *(iii) non-semantic change*, if the rule triggers identically in both versions.

In Figure 7, we give the number of usage changes classified into security fixes, buggy changes, and non-semantic changes with respect to rules CL1-CL5. Note that the total number of changes varies across the rules as we count only changes that are applicable to the rule. For example, CL1 refers to the class `Cipher`, for which we have collected

Rule	Change Type	Total Changes	Filtered changes f_{same}	f_{add}	f_{rem}	f_{dup}	Remain. changes
CL1	fix	8	0	0	0	1	7
	bug	1	0	0	0	0	1
	none	15820	15410	215	88	40	67
CL2	fix	1	0	0	0	0	1
	bugs	0	0	0	0	0	0
	none	4966	4909	34	12	1	10
CL3	fix	4	0	0	0	0	4
	bug	1	0	0	0	0	1
	none	15538	15317	106	65	10	40
CL4	fix	1	0	0	0	0	1
	bug	0	0	0	0	0	0
	none	1548	1520	8	4	0	16
CL5	fix	1	0	0	0	0	1
	bug	1	0	0	0	0	1
	none	1547	1520	8	4	0	15

Figure 7. Filtered security fixes (fix), buggy changes (bug), and non-semantic changes (none) using the four filters. The right-most column shows the type of changes that remain after applying all filters.

15,829 usage changes in total. In the figure, we also show the number of usage changes removed by each filter.

The data shows two important findings. First, most code changes have no-semantic meaning with respect to these rules. The filters, however, effectively eliminate the non-semantic changes, where the most effective filter is f_{same} which can detect and eliminate code refactorings. Second, the semantic changes are not filtered out. Only 1 semantic change is eliminated by the f_{dup} filter to remove a duplicate security fix (see CL1).

The data in Figure 7 also shows that most of the changes are indeed security fixes, not buggy changes. Namely, over 80% of the code changes correspond to actual security fixes.

6.3 Clustering Security Fixes and Eliciting Rules

Next, we report on our experience in eliciting rules from the security fixes. We inspected each fix, together with any other modifications that are similar (in terms of distance), on GitHub. In more detail, we inspected the concrete code patch, the commit message, and any additional comments that describe the commit.

Clustering Security Fixes. We constructed a dendrogram for each target API class, using the hierarchical clustering algorithm described in Section 4.3. We depict a (partial) dendrogram derived for the `Cipher` API class in Figure 8. This dendrogram shows three usage changes. These changes show that developers are switching from the insecure ECB mode of

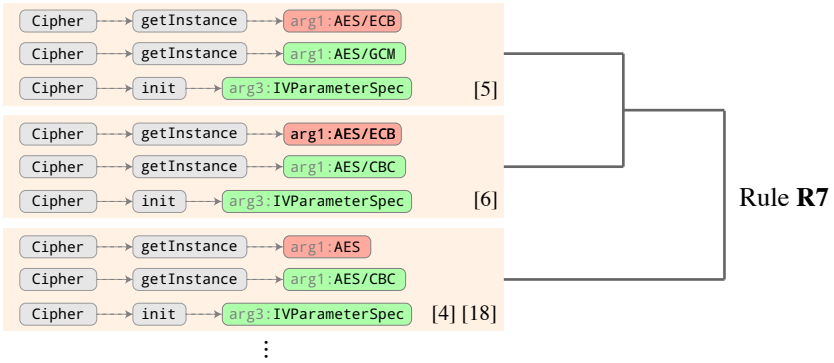


Figure 8. Partial hierarchical clustering for the Cipher API class. All three usage changes show a switch from the insecure ECB mode of AES to the more secure CBC and GCM modes. The merge of the three usage changes identifies the security rule **R7** stating that the cipher should not be used in ECB mode. We give links to concrete GitHub revisions in the references.

AES to the more secure CBC and GCM modes. We also give the concrete commits (with links in the references) that are identified with these usage changes. The top-most two usage changes are joined together to form a cluster, and then this cluster is joined with the third usage change. We found such information provided by the clustering helpful to navigate through and inspect the security fixes.

Rules. We consider rules of the form $t\colon \varphi$ where $t \in \mathit{Types}$ is a type and φ is a logical formula interpreted over a set $S \subseteq \mathcal{P}(\mathit{Methods} \times \mathit{AStates})$ of method and abstract state pairs. For example, the logical formula

$$\varphi \equiv \exists (m, \sigma^a) \in S.\ m = \texttt{getInstance}(X) \wedge \Delta^a(X) = \texttt{SHA-1}$$

is satisfied, denoted $S \models \varphi$, if the set S contains a pair (m, σ^a) such that $m = \texttt{getInstance}(X)$ and $\sigma^a(X) = \texttt{SHA-1}$. A rule $t\colon \varphi$ *matches* an abstract object a of a given program with abstract uses AUses if $\mathit{type}(a) = t$ and $\mathit{AUses}(a) \models \varphi$. We say that a rule $t\colon \varphi$ is applicable to an abstract object a if $\mathit{type}(a) = t$. For brevity, we write

$$\texttt{getInstance}(X) \wedge X = \texttt{SHA-1}$$

as a shorthand for the logical formula given above. We sometimes conjoin multiple rules to express more complex ones. For example, the composite rule $(t_1\colon \varphi_1) \wedge (t_2\colon \varphi_2)$ matches a program P if both $t_1\colon \varphi_1$ and $t_2\colon \varphi_2$ match some, possibly different, abstract objects in P.

Elicited Rules. Using DiffCode on our data set of Java projects, we managed to discover a number of different security rules, which we also validated by checking security papers, blogs, and bulletins. In Figure 9 we list all security rules. Out of these, **R2**, **R7**, **R9**, **R10**, **R11**, **R12** are known and have been documented. For details on these we refer the reader to [12]. We next describe some of the other rules.

Rule **R1** states that SHA-256 should be used instead of SHA-1. Indeed, security researchers have recently announced the first practical technique for generating a collision for SHA-1 [30], and they have warned developers that it is recommended to switch to the more secure SHA-256.

Rule **R3** states that the preferred mode for using the `SecureRandom` class is `SHA-1PRNG`, which is initially seeded via a combination of system attributes and the `java.security` entropy gathering device. Later, we have found the motivation to be described in [2].

Rule **R4** states that `SecureRandom.getInstanceStrong()` should be avoided on server-side code running on Solaris/Linux/MacOS where availability is important, as documented in [28]. This is because `SecureRandom.getInstanceStrong()` returns the `NativePRNGBlocking` mode of `SecureRandom`, which may block and thus developers suggest to avoid it [28].

Rule **R5** states that the `BouncyCastle` provider should be used instead of the default Java Crypto API provider because BouncyCastle does not have the 128 bit secret key restriction [3].

Rule **R6** detects that `SecureRandom` is vulnerable on Android SDK versions 16, 17, and 18 if the Linux PRNG module is not installed [1]. The implementation of the check `HAS_LPRNG` is described in [1].

Rule **R8** states that `Cipher` should not be used in `DES` mode because this mode is no longer considered secure [23].

Rule **R13** states that developers should add integrity after having exchanged a symmetric key, which is frequently done with an asynchronous cipher such as `RSA`. A common fix is to switch to the `AES` cipher with or in combination with `HMAC` [6]. Note that, to match vulnerable projects, rule **R13** is expressed as a composite rule that refers to three distinct objects – two objects of type `Cipher` and one object of type `Mac`. We remark that the rule matches any projects that have the two Cipher objects but lacks the required Mac object. In particular, the rule does not explicitly define in which order these objects are declared and used.

Overall, while some of these rules may be known to some security researchers, with the DiffCode system we were

ID	Description	Rule
R1	Use SHA-256 instead of SHA-1 [30]	MessageDigest: getInstance(X) ∧ X=SHA-1
R2	Do not use password-based encryption with iterations count less than 1000 [7]	PBEKeySpec: <init>(_,_,X,_) ∧ X<1000
R3	SecureRandom should be used with SHA-1PRNG [2]	SecureRandom: <init>(X) ∧ X≠SHA-1PRNG
R4	SecureRandom with getInstanceStrong should be avoided	SecureRandom: ¬getInstanceStrong
R5	Use the BouncyCastle provider for Cipher	Cipher: getInstance(_,X) ∧ X≠BC
R6	The underlying PRNG is vulnerable on Android v16-18 [17]	SecureRandom: <init>(_) ∧ ¬LPRNG ∧ MIN_SDK_VERSION≥16
R7	Do not use Cipher in AES/ECB mode [9]	Cipher: getInstance(X)∧ (X=AES ∨ X=AES/ECB)
R8	Do not use Cipher with DES mode [23]	Cipher: getInstance(X) ∧ X=DES
R9	IvParameterSpec should not be initialized with a static byte array [9]	IvParameterSpec: <init>(X) ∧ $X \neq \top_{byte[]}$
R10	SecretKeySpec should not be static	SecretKeySpec: <init>(X) ∧ $X \neq \top_{byte[]}$
R11	Do not use password-based encryption with static salt	PBEKeySpec: <init>(_,X,_,_) ∧ $X \neq \top_{byte[]}$
R12	Do not use SecureRandom static seed	SecureRandom: setSeed(X) ∧ $X \neq \top_{byte[]}$
R13	Missing integrity check after symmetric key exchange [6]	(Cipher: getInstance(X) ∧ *startsWith*(X,AES/CBC)) ∧ (Cipher: getInstance(Y) ∧ Y=RSA) ∧ ¬(Mac: getInstance(Z) ∧ *startsWith*(Z,Hmac)

Figure 9. Security rules derived from security fixes applied to the Java Crypto API.

able to systematically derive all of them. Further, DiffCode enabled us to create a single checker for all of these rules.

On Automating Rule Elicitation. We remark that DiffCode can also automatically suggest a rule by constructing a predicate that matches any use that has the features present in the old versions and does not have those added to the new versions. Note that this predicate would match any usage that is not fixed according to the code changes. As a simple example, consider the removed and added features depicted in Figure 2(d). The generated security rule would be

$$\begin{aligned}&\texttt{Cipher}\colon (\texttt{getInstance}(X) \wedge X = \texttt{AES})\\ &\quad \wedge(\texttt{getInstance}(Y) \Rightarrow Y \neq \texttt{AES/CBC/PKCS5Padding})\\ &\quad \wedge(\texttt{<init>}(X', \_, \_, Y') \Rightarrow Y' \neq \texttt{IVParameterSpec})\end{aligned}$$

This rule captures that AES Ciphers which use the default AES mode, and neither use the AES/CBC/PKCS5Padding mode nor pass an object of type IVParameterSpec to the constructor, must be fixed. While using the above method one can completely automate the generation of rules, identifying whether a rule is security-relevant in a purely automated manner is challenging and goes beyond the scope of this work.

6.4 Relevance of The Elicited Security Rules

To evaluate the relevance of the discovered security rules, we developed a security checker, called CryptoChecker, that supports all rules in Figure 9. We ran CryptoChecker on 519 Java projects. These include all 463 Java projects we used for training as well as additional 56 projects which we downloaded after eliciting the rules. We report the number of discovered rule violations in Figure 10. For each security rule, we give *(i)* the total number of projects that have at least one usage *applicable* to the security rules, and *(ii)* the number of projects that have at least one *insecure* usage according to our rules. For instance, rule **R1** is only applicable to usages of the API class MessageDigest as it stipulates how classes of type MessageDigest should be instantiated. The number 257 in the first row thus indicates that there are 257 projects (49.5% of the 519 projects) that have at least one usage of type MessageDigest. The *matching* column indicates that 89 out of the 257 projects (34.6%) have at least one usage that matches rule **R1** .

Rule	Applicable	(% of total)	Matching	(% of appl.)
R1	257	(49.5%)	89	(34.6%)
R2	64	(12.3%)	15	(23.4%)
R3	305	(58.8%)	289	(94.8%)
R4	305	(58.8%)	3	(1%)
R5	211	(40.7%)	206	(97.6%)
R6	59	(11.4%)	48	(81.4%)
R7	211	(40.7%)	60	(28.4%)
R8	211	(40.7%)	20	(9.5%)
R9	124	(23.9%)	7	(5.6%)
R10	232	(44.7%)	12	(5.2%)
R11	64	(12.3%)	7	(11%)
R12	305	(58.8%)	1	(0.3%)
R13	8	(1.5%)	4	(50%)

Figure 10. Rule violations for the analyzed projects.

Overall, the data in Figure 10 confirms recent findings that developers struggle to use the Java Crypto API correctly [24]. In > 57% of the projects CryptoChecker discovers at least one security rule that is matched. We remark that CryptoLint [12], a similar checker to CryptoChecker, can be used to check some (but not all) of CryptoChecker's rules. However, since CryptoLint is not publicly available, we were unable to compare our results.

Finally, we used some of the reports of CryptoChecker to report 15 security violations, 3 of which were confirmed. The reported issues are listed at http://diffcode.ethz.ch.

7 Related Work

In this section, we survey several recent works that are most closely related to ours.

Misuse of Crypto APIs. The authors of [22] describe a set of security guidelines related to the use crypto APIs, e.g. that password-based encryption must be used with a random seed, encryption keys must not be hard-coded, and so forth. Applications that do not follow these guidelines are considered vulnerable. They examine 49 Android applications and show that 87.8% of them suffer from at least one vulnerability.

The authors of [11] present a manual analysis of different crypto APIs (OpenSSL, Java, PyCrypto, and others) and discuss seven problems related to API misuses, such as reuse of initialization vectors, lack of code samples in the API documentation, safe API defaults, and so forth. The scope of this report is on crypto APIs, not on the applications that use these APIs. In [24], the authors report on a survey to discover why developers often fail to use crypto APIs correctly. They also present suggestions to API developers that may mitigate the problem of API misuse.

Detecting Misuse of Crypto APIs. Several works consider the problem of automatically detecting misuse of crypto API in Java applications. OWASP provides a list of static analysis tools [5] that target security issues, however these tools have a very limited set of crypto checks (e.g. see FindSecBugs [4]). CryptoLint [12] is a specialized system that checks Android applications for crypto API misuses with six fixed rules, such as "Do not use ECB mode for encryption". To check these properties, CryptoLint statically computes a program slice immediately before invoking a crypto API and checks properties on the arguments passed to that API. They evaluate >11K Android applications and show that 88% of the applications violate at least one rule. Compared to CryptoLint, CryptoChecker supports a more comprehensive set of security rules.

The CMA analyzer, presented in [29] is very similar to [12] and also targets finding misuses of crypto APIs in Android. Compared to [12], CMA considers more security rules.

AmanDroid [31] is a system for precise static analysis of Android applications. AmanDroid computes a dependency graph that captures control- and data-flow dependencies for all objects. The security analysis is then phrased as graph queries over the dependency graph. For example, we can check an application for absence of data leaks by checking that there is no path from a source to a sink. AmanDroid can be also used to check whether applications misuse crypto APIs by encoding the rules of [11, 13, 24] in terms of graph queries. In [16], the authors explain the concepts behind AmanDroid in a more general, cleaner setting.

Misuse of crypto APIs is not an Android or Java-specific problem: the evaluation of a dynamic analysis tool for iOS [20] found that over 65% of the tested iOS applications suffer from vulnerabilities due to API misuse.

Repairing Misuse of Crypto APIs. The CDRep system presented in [22] can be used to detect and repair misuses of Android's crypto API. For the detection step, CDRep performs an analysis similar to the one presented in [12]. Given an Android APK, CDRep detects the instructions responsible for a particular misuse of the crypto API. The responsible instructions include the call to a particular crypto API, called the indicator instruction, and instructions on which the indicator instruction depends. For example, one security rule states that applications must not use encryption in ECB mode. The responsible instructions that violate this rule would include: the call to the encrypt method, the instruction that constructs the encryption object, and the instruction that initializes the encryption-scheme string (e.g. "AES/ECB") passed to the encryption constructor. After identifying the instructions responsible for a given cryptographic misuse, CDRep uses manually pre-defined patch templates to suggest candidate repairs.

Learning from Code. Several prior works check for API errors by first learning a likely specification of a program and its API calls [13, 18]. The recent APISan system [32] automatically infers the correct usage of APIs by observing the contexts of the calls from multiple projects. For example, APISan can learn that the return value of a method is typically checked for null and then it can report outliers to this learned specification. This allows APISan to check large codebases in a precise and scalable way for given predefined types of issues such as null dereferences and overflows. In contrast to these works, we focus on crypto APIs for which (i) we do not know the kind of issues that may be present and (ii) the majority of the projects misuse the APIs. The work of Long and Rinard [21] considers the reverse task of ours and learns from correct code to guide automatic generation of bug fixes.

Learning from Code Changes. Several works propose to learn from previous code changes to help developers complete a new change. A system by Zimmermann et al. [33]

warns developers if a newly developed change only does a subset of what other changes did. A more recent work by Nguyen et al. [25] developed a code completion engine that precisely predicts code for new changes based on code in previous code changes. In contrast to our approach, however, these works cannot find issues in existing code and make predictions only for code modifications.

8 Conclusion

We presented a new data driven approach for extracting semantically meaningful API usage changes from concrete code fixes collected by processing public repositories. The approach is based on an abstraction for code changes which captures the implications of a change to objects of the Crypto API. These implications are represented as semantic features that are removed from the old and added to the new version of the program. Our abstraction enables us to distill relevant semantic changes using filters that eliminate purely syntactic modifications. As a final step we (hierarchically) cluster the remaining, semantically meaningful security fixes, enabling us to derive new security rules.

We also presented DiffCode, a system that implements our data driven approach. We applied DiffCode to Java code changes collected from GitHub and extracted security fixes for the Java Crypto API. Based on these results, we identified 13 relevant security rules which we implemented in a new security checker called CryptoChecker. We evaluated CryptoChecker on a number of public Java projects, discovering misuses of the Java Crypto API in $> 57\%$ of the analyzed projects.

The data driven approach presented in this work allowed us to systematically derive relevant security rules some of which are missing from existing checkers. We believe that this work is an important step towards solving the general problem of automatically deriving API misuse checks.

References

[1] 2013. Some SecureRandom Thoughts. https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html

[2] 2015. The Right Way to Use SecureRandom. https://tersesystems.com/2015/12/17/the-right-way-to-use-securerandom/

[3] 2016. Which security implementation should I use: Bouncy Castle or JCA? https://blog.idrsolutions.com/2016/08/which-security-implementation-should-i-use-bouncy-castle-or-jca/

[4] 2017. FindSecBugs Bugs Patterns. https://find-sec-bugs.github.io/bugs.htm

[5] 2017. OWASP Source Code Analysis Tools. https://www.owasp.org/index.php/Source_Code_Analysis_Tools.

[6] 2017. Top 10 developer Crypto mistakes. https://littlemaninmyhead.wordpress.com/2017/04/22/top-10-developer-crypto-mistakes/

[7] Martín Abadi and Bogdan Warinschi. 2005. *Password-Based Encryption Analyzed.* Springer Berlin Heidelberg, Berlin, Heidelberg, 664–676. https://doi.org/10.1007/11523468_54

[8] Mikhail J. Atallah and Susan Fox (Eds.). 1998. *Algorithms and Theory of Computation Handbook* (1st ed.). CRC Press, Inc., Boca Raton, FL, USA.

[9] M. Bellare and P. Rogaway. 2017. Course notes for introduction to modern cryptography. cseweb.ucsd.edu/users/mihir/cse207/classnotes.html

[10] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '78).* ACM, New York, NY, USA, 84–96. https://doi.org/10.1145/512760.512770

[11] Somak Das, Vineet Gopal, Kevin King, and Amruth Venkatraman. [n. d.]. *IV = 0 Security Cryptographic Misuse of Libraries.* Technical Report. MIT.

[12] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13).* ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/2508859.2516693

[13] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code. *SIGOPS Oper. Syst. Rev.* 35, 5 (Oct. 2001), 57–72. https://doi.org/10.1145/502059.502041

[14] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. 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 (CCS '12).* ACM, New York, NY, USA, 50–61. https://doi.org/10.1145/2382196.2382205

[15] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12).* ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/2382196.2382204

[16] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. 2015. Exploring and Enforcing Security Guarantees via Program Dependence Graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15).* ACM, New York, NY, USA, 291–302. https://doi.org/10.1145/2737924.2737957

[17] David Kaplan, Sagi Kedmi, Roee Hay, and Avi Dayan. 2014. Attacking the Linux PRNG On Android: Weaknesses in Seeding of Entropic Pools and Low Boot-Time Entropy. In *8th USENIX Workshop on Offensive Technologies (WOOT '14).* USENIX Association, San Diego, CA. https://www.usenix.org/conference/woot14/workshop-program/presentation/kaplan

[18] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. 2006. From Uncertainty to Belief: Inferring the Specification Within *(OSDI '06).* USENIX Association, Berkeley, CA, USA, 161–176. http://dl.acm.org/citation.cfm?id=1298455.1298471

[19] Ondrej LhotÃąk. 2002. Spark: a flexible points-to analysis framework for Java.

[20] Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2014. iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications. In *Network and System Security: 8th International Conference.* 349–362. https://doi.org/10.1007/978-3-319-11698-3_27

[21] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16).* ACM, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[22] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. 2016. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16).* ACM, New York, NY, USA, 711–722. https://doi.org/10.1145/2897845.2897896

[23] Dhruv Mohindra. 2016. Do not use insecure or weak cryptographic algorithms. https://www.securecoding.cert.org/confluence/display/java/MSC61-J.+Do+not+use+insecure+or+weak+cryptographic+algorithms

[24] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 935–946. https://doi.org/10.1145/2884781.2884790

[25] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. 2016. API Code Recommendation Using Statistical Learning from Fine-grained Changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 511–522. https://doi.org/10.1145/2950290.2950333

[26] Oracle. 2017. Java Cryptography Architecture (JCA) Reference Guide. http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto

[27] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 419–428. https://doi.org/10.1145/2594291.2594321

[28] Amit Sethi. 2016. Proper use of Java SecureRandom. https://www.synopsys.com/blogs/software-security/proper-use-of-javas-securerandom/

[29] Shao Shuai, Dong Guowei, Guo Tao, Yang Tianchang, and Shi Chenjie. 2014. Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications. In *Proceedings of the 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (DASC '14)*. IEEE Computer Society, Washington, DC, USA, 75–80. https://doi.org/10.1109/DASC.2014.22

[30] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, Yarik Markov, Alex Petit Bianco, and Clement Baisse. 2017. Announcing the first SHA1 collision.

[31] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 1329–1341. https://doi.org/10.1145/2660267.2660357

[32] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: Sanitizing API Usages through Semantic Cross-Checking. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 363–378. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun

[33] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. 2004. Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 563–572. http://dl.acm.org/citation.cfm?id=998675.999460