To Fix or Not to Fix: A Critical Study of Crypto-misuses in the Wild

Anna-Katharina Wickert*, Lars Baumgärtner*, Michael Schlichtig†, Krishna Narasimhan*, Mira Mezini*
*Technische Universität Darmstadt
Darmstadt, Germany
Email: <wickert,baumgaertner,kri.nara,mezini>@cs.tu-darmstadt.de,
†Heinz Nixdorf Institute, Paderborn University
Paderborn, Germany
Email: michael.schlichtig@uni-paderborn.de

***Abstract*—Recent studies have revealed that 87 % to 96 % of the Android apps using cryptographic APIs have a misuse which may cause security vulnerabilities. As previous studies did not conduct a qualitative examination of the validity and severity of the findings, our objective was to understand the findings in more depth. We analyzed a set of 936 open-source Java applications for cryptographic misuses. Our study reveals that 88.10 % of the analyzed applications fail to use cryptographic APIs securely. Through our manual analysis of a random sample, we gained new insights into *effective false positives*. For example, every fourth misuse of the frequently misused JCA class `MessageDigest` is an *effective false positive* due to its occurrence in a non-security context. As we wanted to gain deeper insights into the security implications of these misuses, we created an extensive vulnerability model for cryptographic API misuses. Our model includes previously undiscussed attacks in the context of cryptographic APIs such as DoS attacks. This model reveals that nearly half of the misuses are of high severity, e.g., hard-coded credentials and potential Man-in-the-Middle attacks.**

***Index Terms*—API-misuses, cryptography, false positives**

I. Introduction

Many applications need to protect sensitive and high-value data, such as passwords or financial transactions, using cryptography (hereafter referred to as *crypto*). For this purpose, crypto APIs provide access to crypto tasks, protocols, and primitives in all major programming languages. However, several studies have revealed that developers struggle to use crypto APIs [14], [17] correctly and introduce misuses, meaning that an API usage may be used syntactically correct but problematic from a security perspective. A common crypto misuse is the use of the *Electronic Code Book (ECB)* mode for encryption. Although it has been known for a long time that *ECB* is insecure [5], it was, e.g., found in the widely used Zoom video conferencing system until May 2020[1].

To address this problem, static analyzers such as *SpotBugs*[2], *CryptoGuard* [19], and *CogniCrypt*$_{SAST}$ [13], have been proposed to support developers and security researchers in checking for such misuses. These tools have been used in various empirical studies that have produced worrying insights into the state of crypto usages in the wild [5], [13], [19], [9]. However, none of the studies performed a qualitative examination of the validity and severity of the findings. Such an examination is, however, essential for both getting a more realistic picture of the state of crypto usages in the wild and to improve future studies and analyzes. It is well-known that static analyzes may produce false positives, which are considered their *Achilles heel* [11], [24]. Moreover, in a security context, the validity of any finding should be judged from the perspective of a threat model to get a feeling for its severity.

This paper contributes to closing this gap by designing and conducting a study focusing on qualitative examination of reported crypto misuses. With regard to false positives, we are particularly interested in reports that are specific to their concrete usage. An API may be used in different – non-crypto - contexts and not the same constraints apply in all contexts. For example, the most misused JCA class `MessageDigest` from previous studies [9], [13], [19] may be used to compute hashes independent of a security context, e.g., to compute the hash of a file. However, a static analysis cannot know this and has to be conservative. Thus, it may produce false positives and draw an inaccurate picture of the security of our software. With regard to qualitatively judging the severity of findings, we formulate and use a novel, comprehensive threat model. Specifically, we designed and conducted a study to answer the following research questions:

RQ1: What are common (effective) false positives arising from misuses, e.g., due to a non-security context?
RQ2: How severe are the vulnerabilities introduced by crypto API misuses in applications?

We studied crypto misuses of two Java crypto providers – the Java Cryptography Architecture (JCA) and Bouncy Castle (BC) – in open-source Java projects from GitHub using *CogniCrypt*$_{SAST}$ [13]. We only included projects that are mainly developed and maintained by professionals to address the generalizability problem of open-source studies [21]. Altogether, we collected 936 Java projects, of which 210 use a crypto API and 88.10 % have at least one misuse.

[1]https://support.zoom.us/hc/en-us/articles/360043770412-Updating-your-Zoom-Rooms-to-version-5-0-5
[2]https://spotbugs.github.io/

```
1 private byte[] signByte(byte[] dataToSign){
2    byte[] signedBytes;
3    Signature s = Signature.getInstance("SHA1WithRSA");
4    s.initSign(getPrivateKey());
5    s.update(dataToSign);
6    // Call to signedBytes = s.sign() missing.
7    return signedBytes;
8 }
9
10 // Get a PrivateKey object with an insecure key length.
11 private PrivateKey getPrivateKey() {
12     return shortKey();
13 }
```

Listing 1: Code snippet that signs a byte array using a key.

To qualitatively analyze the misuses, we randomly picked 157 misuses for review. We observed that one fourth of the misuses of the class `MessageDigest` – the most misused class according to previous studies [13], [19], [9] and the second most in our study – occur in a non-security context. Thus, we can consider these misuses as *effective false positives* [20] that may not or cannot be fixed by developers.

To judge the severity of all findings, we defined a threat model that connects API misuses reported by *CogniCrypt$_{SAST}$* to security vulnerabilities. This model subsumes the existing vulnerability model for crypto API misuses by Rahaman et al. [19] and includes new threats, e.g., *Denial of Service* (DoS) attack and *Chosen-Ciphertext Attacks* (*CCA*). Overall, our model marks 42.78 % of the misuses as high severity.

In summary, this paper makes the following contributions:

- We studied crypto misuses found in a large, representative data set of applications [21].
- We provide empirical evidence that the reported misuses contain a significant amount of *effective false positives*. Our disclosure confirms this and reveals that many may not be fixed even though some are ranked as high severity.
- We created a novel, comprehensive threat model mapping crypto API misuses to vulnerabilities, introducing previously undiscussed threats like DoS attacks and CCAs.

II. Background

In this section, we provide background information regarding crypto API misuses in Java and introduce *CogniCrypt$_{SAST}$* [13], the crypto API misuse analyzer we used for our study. In addition, we introduce the term *effective false positives*.

A. Misuses of Java Crypto APIs

The JCA provides a set of extensible cryptographic components ranging from encryption over authentication to access control, enabling developers to secure their applications. It is implementation-independent by using a "provider" architecture; developers can plug and play their implementation of crypto primitives for use with this architecture. A commonly used provider besides the default that is shipped with the Java Development Kit (JDK) is the Bouncy Castle library.

Listing 1 illustrates a usage of the JCA to sign a byte-array `dataToSign`. For this, the `Signature` object is initialized with a signature algorithm (Line 3) and a private key, passed via the function `getPrivateKey`, is added to the `Signature` object `s` (Line 4). Next, the byte-array `dataToSign` is passed to `s` to actually compute the signature of the data (Line 5) before the function returns the signed bytes. Unfortunately, the call to `sign` that would return the signature is missing (Line 6).

A *crypto misuse*, hereafter just misuse, is a usage of a crypto API that is considered insecure by experts. A misuse may be syntactically correct, a working API usage, and may not even raise an exception. We will briefly discuss the error types defined by Krüger et al. [13] to illustrate some misuses.

1) **Constraint Errors** (Listing 1, Line 3): Crypto APIs use parameters to let developers select crypto algorithms when initializing crypto objects. These parameters are often passed as strings in a specific format.
2) **Incomplete Operation Errors** (Listing 1, Line 6): The security of crypto objects may rely on a specific protocol. For instance, `Signature` crypto objects, once initialized and filled with data via a call to `update(byte[])`, require a call to the `sign` method to complete the signing operation. Thus, the required calls to complete the use of an initialized crypto object are missing.
3) **Required Predicate Errors** (Listing 1, Line 4): Crypto objects often depend on each other. For example, `Signature` objects require a correctly generated `Key` object. In order for a composed crypto solution to be secure, it is required that its components on which it depends are secure. Thus, composing a crypto object with required but insecure objects results in a misuse.
4) **Never Type of Error:** Sensitive information, e.g., a secret key, should never be of type `java.lang.String`, as strings are considered insecure compared to mutable byte arrays. Strings are immutable and stay in memory until collected by Java's garbage collector. Thus, they are longer visible in memory for attackers than necessary and outside of the direct control of the developer [3].
5) **Forbidden Method Errors:** Certain methods of crypto objects should never be called for security reasons. For example, the `PBEKeySpec` object generates keys from passwords and requires a crypto salt while initializing the object. Therefore, constructors that are not parameterized with a salt cause a misuse.
6) **Type State Error:** Such an error occurs when an object moves into an insecure state as the result of an improper method call sequence. For example, a `Signature` object requires a call to the `initSign` method prior to any number of calls to the `update` method.
   The key difference to an *Incomplete Operation Error* is that the missing and expected call is within the call sequence, while for an *Incomplete Operation Error* the expected call sequence is not finished.

[3]https://docs.oracle.com/en/java/javase/17/docs/api/java.base/javax/crypto/spec/PBEKeySpec.html, accessed 19.09.2022

Vulnerabilities	Attack Type	Severity	Novel	C	IO	RP	NT	FM	TS
Predictable/constant crypto keys	Predictability Through Initialization	H	○	●	○	●	○	○	●
Predictable/constant passwords for PBE		H	○	○	○	○	○	●	○
Predictable/constant passwords for KeyStore		H	○	●	○	○	○	○	○
Cryptographically insecure PRNGs		M	○	○	○	●	○	○	○
Missed to finish crypto function	Predictability Through Usage	H	●	○	●	○	○	○	●
Missed to pass data		M	●	○	●	○	○	○	●
Insecure TrustManager	MitM Attacks on SSL/TLS	H	○	○	○	●	○	○	○
Insecure SSL/TLS standard		H	●	●	●	○	○	●	●
Static Salts in PBE	Chosen-Plaintext Attack (CPA)	M	○	○	○	●	○	○	○
ECB mode in symmetric cipher		M	○	●	○	○	○	○	○
Static IVs in CBC mode symmetric ciphers		M	○	○	○	●	○	○	○
Padding Oracle	Chosen-Aiphertext Attack (CCA)	M	●	●	○	○	○	○	○
Fewer than 10,000 iterations for PBE	Bruteforce Attacks	L	○	●	○	○	○	○	○
64-bit block ciphers		L	○	●	○	○	○	○	○
64-bit authentication tag GCM		L	●	●	○	○	○	○	○
Insecure cryptographic ciphers		L	○	●	○	●	○	○	○
Insecure cryptographic signature		L	●	●	○	○	○	○	○
Insecure cryptographic MAC		L	●	●	○	○	○	○	○
Insecure cryptographic hash		H	○	●	○	●	○	○	○
Usage of String	Credential Dumping	L	●	○	○	○	●	○	○
Missed to clear password		L	●	○	●	○	○	○	○
Trigger Exception	DoS Attacks	M	●	●	○	○	○	○	●

TABLE I: A model of vulnerabilities which can be detected with *CogniCrypt$_{SAST}$*. For novel ○ marks if this vulnerability is discussed by Rahaman et al. [19] and ● if not. For C: Constraint Error, IO: Incomplete Operation Error, RP: Required Predicate Error, NT: Never Type of Error, FM: Forbidden Method Error and TS: Type State Error ● marks if this vulnerability can be caused by the respective error type and ○ if not.

B. *CogniCrypt$_{SAST}$*

For our study, we used the crypto misuse detector *CogniCrypt$_{SAST}$* which follows an allowlisting approach. In contrast to denylisting approaches such as *CryptoGuard* [19] that describe vulnerabilities, allowlisting approaches describe all secure API usages. Violations of the defined rules are reported as misuses [13]. Previous studies reported a precision of 85 % to 94 % [13], [10]. Further, *CogniCrypt$_{SAST}$* supports the BC library next to the JCA.

C. *Effective False Positives*

Past research has shown that developers consider false positives as the "Achilles heel" of static analyzes [11], [2]. However, in practice, the definition of false positives varies: From a static analysis perspective, a false positive is a finding which is incorrectly identified by the analysis. For developers on the other hand, some findings can not be fixed in the application, e.g., due to a broken standard. Sadowski et al. [20] introduce the term *effective false positives* to cover reported misuses on which a user will not take further action.

For an example of an *effective false positive*, consider a usage of *MD5* that is correctly flagged by the static analysis. However, the usage of *MD5* at hand happens in a non-security context as the concrete call cannot be influenced by external factors and is not essential for the security of the software.

III. Threat-model of vulnerabilities introduced by API misuses

To reason about the potential impact of the reported misuses, we contribute a threat model extending existing models [19] with more vulnerabilities caused by API misuses. Specifically, we derive our model from a study of CrySL rules written by crypto experts, the respective APIs, and the misuses observed in our study. In combination with standard attacks and crypto misuses from previous work, our model covers a wide variety of crypto API misuses and their attack potential. We list the vulnerabilities, the attack types, the respective severity, the novelty, and the affected error types in Table I.

1) **Predictability Through Initialization:** Applications can become insecure if sensitive information like a key is predictable [5], [19], [14]. As an instance of this vulnerability, we consider predictable and constant crypto keys, e.g., a hard-coded key or predictable password used to derive a key from. Furthermore, a cryptographic secure random number in Java requires a non-predictable seed as well as the usage of a dedicated class, e.g., `java.security.SecureRandom`. If the application code misses to fullfill these requirements, the crypto operation becomes predictable. While Rahaman et al. [19] separated predictable keys and PRNGs, our model combines them, as both attack types are due to predictability.
2) **Predictability Through Usage:** In contrast to attacks of the type *Predictability Through Initialization*, the improper usage of an API can render the respective computation predictable. An example of this issue is the usage of crypto APIs which rely on data to be processed, e.g., `MessageDigest`, and fail to process the required data. Thus, essentially resulting in a predictable computation.
3) **MitM attacks on SSL/TLS:** The improper usage of

SSL/TLS can enable an attacker to launch a Man-in-the-Middle (MitM) attack to gain sensitive information [6]. This includes improper configuration of connections, e.g., incorrect verification of protocols, as well as insecure cryptographic protocols, e.g., TLS 1.1, which are vulnerable to attacks like the POODLE attack.

4) **Chosen-Plaintext-Attack (CPA):** An encryption algorithm should be provably secure against chosen-plaintext-attacks (CPA) [5]. An example is a static *Initialization Vector* (IV) for the *Cipher Block Chaining* (CBC) mode.
5) **Chosen-Ciphertext-Attack (CCA):** While an encryption scheme should be provably secure against CPA, it should also be safe against chosen-ciphertext-attacks (CCA). Concretely, we cover improper padding schemes, like PKCS5 and PKCS7 in combination with the block cipher mode CBC [23], [12], and the usage of plain RSA [3] as these are all vulnerable to CCA. Note, that a padding attack requires an interaction between the attacker and the program that responds to messages from the attacker. Thus, data at rest is not vulnerable.
6) **Bruteforce Attacks:** Some cryptographic algorithms are vulnerable to repeated, extensive computations, e.g., a feasible collision computation for MD5 and SHA-1 [22]. Further, primitives like DES [19] with a block size of 64 bit or a 64-bit authentication tag for GCM [8] are vulnerable to brute-force attacks to break the encryption.
7) **Credential dumping:** In Java, string values are immutable and therefore cannot be cleared or overwritten from memory, except when the garbage collector runs, which is out of the developer's control. Thus, the JCA enforces the usage of byte arrays, e.g., `PBEKeySpec`, to handle sensitive information. However, developers may use strings to store this information and only convert it to byte arrays before calling the crypto API. Furthermore, classes like `PBEKeySpec` provide the method `clearPassword` to clear the internal copy of the password. However, usages of this class without calling this method render an application vulnerable to credential dumping.
8) **DoS Attacks:** A Denial-of-Service (DoS) attack limits the availability of a service, e.g., by deliberately overloading the memory or consuming an extensive amount of CPU cycles. One possibility to achieve this is by triggering an unintended behavior such as uncaught exceptions [27]. Such exceptions can be raised when contradicting the API contract, e.g., by missing the initialization of a cipher object[4]. Depending on the program, such an exception can prevent further correct control flow, can cause a program crash, or if triggered in rapid succession, it can spam the log file, increase CPU usage or cause IO congestion. A recent example is CVE-2021-23372, where an exception causes the application to crash.

We categorize the severity into high, medium, and low. Our prioritization is based on factors like whether the vulnerability can be exploited remotely, how difficult an attack is to perform, and if the attack leads to a direct gain or needs the combination with other flaws to be of use. Attacks such as MitM which can be triggered remotely and provide direct benefits for an attacker [19] are ranked as high severity. We mark vulnerabilities as medium severity, which lead to compromises of secrets for active, dedicated attackers. These vulnerabilities are of great help in undermining the security of the systems [26]. Examples of this are CPA and CCA attacks. An example of a vulnerability with low severity are credentials passed as a string, as these require prior access to the system or running process to be extracted. Thus, the attacker must have exploited other vulnerabilities before to gain access to the system.

IV. Empirical Study

In this section, we describe the setup of our study and the insights gained from it. First, the data set used for our analysis is described in Section IV-A, followed by the misuses identified in Section IV-B. Our research questions are answered in Section IV-C and IV-D, respectively. We published an artifact[5] including our script for the threat model.

A. Dataset

The dataset created by Spinellis et al. [21] addresses the generalizability problem of open-source studies and consists of 17,264 GitHub projects that are mainly developed or guided by enterprise employees and span multiple languages. *CogniCrypt$_{SAST}$* works on Java binaries, hence, we filtered out the non-Java projects and attempted to compile the rest automatically for Maven/Gradle build instructions. This resulted in 936 Java enterprise-driven, open-source binaries, which serve as the basis of our study. We applied the *CogniCrypt$_{SAST}$* version 2.7.2[6] with the corresponding rule set for JCA and BC to them[7]. As the objective of our study was exploratory in nature and was designed to understand cryptographic misuses in the wild and not evaluate the tools themselves, we choose *CogniCrypt$_{SAST}$* due to its allowlisting approach and the inclusion of rules for JCA and BC. For each successfully analyzed application in our data set, *CogniCrypt$_{SAST}$* created a report including details of the crypto objects analyzed and the identified misuses.

B. Crypto (Mis)uses - an Overview

Prevalence of JCA and BC. We consider an application to use a crypto API if we found at least one usage of either JCA or BC, without judging the security of the respective usage yet. In total, we found 210 applications that use a crypto API. All of these projects use the standard Java crypto library (JCA), while 7 of these projects also use the BC API. Within these applications, we analyzed 3,294 different crypto objects (hereafter object)[8] for JCA and BC.

[4]https://docs.oracle.com/en/java/javase/17/docs/api/java.base/javax/crypto/Cipher.html, accessed 19.09.2022

[5]https://doi.org/10.6084/m9.figshare.21178243

[6]https://github.com/CROSSINGTUD/CryptoAnalysis/releases/tag/2.7.2

[7]Previous studies used older *CogniCrypt$_{SAST}$* versions [13], [9], [19].

[8]*CogniCrypt$_{SAST}$* uses the term OBJECT to refer to any initialized Java object which interacts with a class covered by a CrySL rule [13]

JCA Class	Misuses observed	Projects affected
`Cipher`	563	36
`MessageDigest`	472	113
`SSLContext`	414	68
`SecretKeySpec`	218	53
`Mac`	161	34
`Signature`	143	18
`KeyStore`	120	44

TABLE II: An overview of the observed misuses and affected projects for all JCA classes with more than 100 misuses.

Misuses of JCA and BC. In the 210 projects, we spotted crypto misuses in 185 and out of these 5 projects have misuses of both the BC and JCA library. The remaining 180 projects only use the JCA library for crypto. While previous studies reported 95 % [13] and 96 % [9] of applications with misuses for the JCA, we observe 88.10 % projects with at least one misuse. Thus, the selection criteria of so-called enterprise-driven applications [21] may have a slight positive impact on the number of misuses and thus security of the applications.

Leading causes of misuses (error types): We identified 2,695 misuses and those are mainly distributed over *Required Predicate*, *Incomplete Operation*, and *Constraint* errors. Most of the misuse reports, in total 960, are due to a *Required Predicate*, which means that composing multiple crypto objects seems to be challenging to get right. An example of this misuse is in line 4 of Listing 1 where the passed key is generated insecurely. Thus, the required predicate of the function call `initSign`, namely a secure key, is not fulfilled and causes a misuse. *Incomplete operation* errors arising out of a missing method call contributed to 815 misuses. For example, crypto primitives may require multiple method calls for completion, e.g., initializing, updating, and finally retrieving a hash code. While this is the second most frequent misuse, only 37.80 % of the projects are impacted by such misuses. Most common in applications are misuses due to insecure parameters to functions, e.g., *SHA-1* as a parameter to an initialization of a `MessageDigest` object. Overall, 80 % of the applications contain at least one such *Constraint* error, causing in total 566 misuses. This prevalence can be explained by the fact that the security of algorithms evolves over time, algorithms may have not been updated, and may be used for tasks beside the crypto domain. We observed both cases during the disclosure of our manual analysis (Sec IV-E).

Most frequently misused crypto classes: The error types discussed above describe misuses from the API perspective. Another interesting perspective is investigating which crypto primitives were often misused based on the classes that pertain to their implementation, as certain classes may be more prone to *effective false positives*, e.g., `MessageDigest`, or to high-severity misuses, e.g., `SSLContext`. In the following, we will discuss all classes with more than 100 misuses. Table II presents the total number of misuses and affected projects for these classes. While in total most of the misuses occur due to the class `Cipher`, 80.54 % of the projects, have no misuse of this class at all. Thus, affecting only 36 projects. We observed the second-most misuses due to the class, `MessageDigest` with 472 misuses in total. Thus, more projects (113) have at least one misuse of the class `MessageDigest` than no misuse of this class. This indicates that misuses of the class `MessageDigest` are widespread among many applications. The class `SSLContext` contributes to 414 misuses in total within 68 projects. We observe fewer misuses for the class `SecretKeySpec` with 218 misuses in total distributed among 53 projects. These results reveal that only a few JCA classes are frequently misused in many of the applications. For our data set, we can avoid 61.86 % of the misuses by fixing misuses of these four classes.

C. Manual Analysis of Reports (RQ1)

To gain a more in-depth understanding of common (effective) false positives, we randomly sampled 157 (Confidence: 99 %, margin of error: 10 %) misuses. The first four authors of this paper independently analyzed the sampled misuses, with at least two reviews per misuse. In case of disagreement, the reviewers in question resolved them with further code review until they reached a conclusion. The focus of our analysis is the precision of the report as well as the context of the misuse to answer our first research question.

Previous studies [13] concentrated on measuring precision under the assumption that the rules of the static analysis are correct and based on error types that can be easily verified manually, namely *Constraint* and *Type State* errors. In contrast, we inspected all error types and show that some misuses can occur due to an incomplete or outdated rule-set or occur in a non-security context. Thus, resulting in *effective false positives*. We also consider such issues as a contributing factor to the overall lower precision discussed in this paper.

Undecided Reports. We explicitly marked misuses as *undecided* if it was impossible to judge their validity due to cryptic or confusing error reports. A common reason was that the reported insecure usage could not be identified at the location of the report, nor in the respective callers. In total, we observed 31 misuses which fall into this category. All of these misuses occur for the more complex error types, namely *Incomplete Operation*, *Required Predicate*, and *Type State*. We assume that these error reports will be of little help to the user attempting to make decisions based on them. For the remainder of the discussion, we will focus on the remaining 126 misuses.

True Positives. Our qualitative analysis revealed 93 (roughly 74 %) true positives distributed among all previously discussed JCA classes and error types. By using regex, e.g., with grep, to detect usages such as MD5, 25 true positives can be detected. Analyst or developers may consider such simple and fast tools as a prefilter of more sophisticated analyses such as *CogniCrypt*$_{SAST}$ [13] and *CryptoGuard* [19]. The remaining true positives are due to an incorrect order of method calls (21), which requires a typestate analysis, storing secrets in Java strings (22), using a default value of the JCA (15), or hard-coded credentials (2). The remaining 8 instances that we classified as true positives from the static analysis depends on information that can not be dissolved statically. True

```
14 final MessageDigest md = getMessageDigest(algo);
15 final byte[] buffer = new byte[4096];
16 int count = 0;
17 while ((count = inputStream.read(buffer)) > 0) {
18     md.update(buffer, 0, count);
19 }
20 return md.digest();
```

Listing 2: A `MessageDigest` object which only calls `update` when it has something to read from an `InputStream`.

positives due to incorrect call orders and relying on defaults are to the best of our knowledge not covered in existing denylist approaches. Thus, showcasing the importance of a sophisticated static analysis beyond simple pattern matching.

We observed 8 misuses because a required call is only present in a loop body where the respective loop may not be executed (Listing 2, Line 18). Thus, a misuse may be present and causing a vulnerability. While these cases should be analyzed to identify if a vulnerability can be triggered causing a true positive, most often these cases results into effective false positives indicating a coding smell. Note, that this is a known limitation of analyses such as *CogniCrypt$_{SAST}$* [13].

False Positives. Our manual analysis resulted in 35 false positives caused by several reasons. One reason are incorrect API specifications. *CogniCrypt$_{SAST}$* relies on the correctness of the API specifications, and in case of incorrect or outdated specifications, the respective report is erroneous and leads to a false positive. In total, we observed 17 misuses due to this reason. For example, the *Optimal Asymmetric Encryption Padding (OAEP)* was encoded with a typo as *OEAP* in the CrySL rule, causing a misuse to be reported, when the correct padding scheme *OAEP* is passed. We fixed this error in a pull request which is already merged into the main branch of the CrySL rules repository. Previous studies have assumed the correctness of the CrySL rules [13] and thus not considered these usages as false positives. With this assumption, our analysis reveals a true positive rate of 84 % that is similar to previous studies [13], [10]. In another example, the call order of functions was modeled incorrectly. After a discussion about this issue, a pull request fixing this problem was opened.

Besides the correctness of the specification, the underlying static analysis can cause false positives due to imprecise modeling of the program flow. Our reviews revealed 11 false positives due imprecise modeling, e.g., wrapping of crypto objects. The remaining false positives are due to further challenges with respect to modeling the program statically.

Effective False Positives. During our analysis, we observed that some misuses discussed previously are *effective false positives* (cf. Section II-C). Thus, the user of the analysis would not take any action, as, while they may acknowledge that it is theoretically a correct finding, it is invalid or irrelevant for the specific application. We provide an overview of the common patterns that we identified in the wild in Figure 1.

Concretely, we identified 9 out of the 126 misuses from a non-security context. One example is the usage of *MD5* in

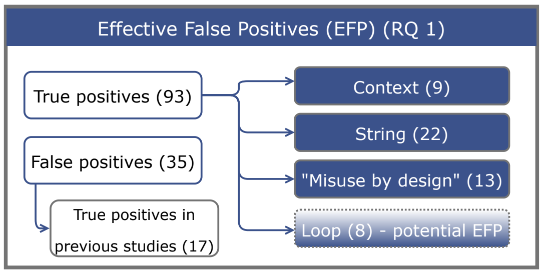


Fig. 1: Common *effective false positives* patterns that we identified during our manual analysis.

deeplearning4j[9]. Here, the *MD5*-digest is used to verify that the input array was not modified during execution. Another example is in the project UAVStack[10] where *MD5*-hashes of a file are used to provide content-aware detection changes.

The number of *effective false positive* misuses due to a non-security context may appear low at first sight. However, one has to consider this number in relation to the overall number of reports for the specific kind, e.g., class `MessageDigest`. It turns out that 22.86 % of the `MessageDigest` misuses in our randomly selected sample fall into the category of *effective false positive* due to non-security context. If we consider (a) that roughly 25% of the reported `MessageDigest` misuses are potentially *effective false positives* and (b) that this class causes by far the most of the misuses reported by previous studies [9], [13], [19], respectively the second most frequent in our study, it becomes clear that previous reports may contain a significant number of *effective false positives*. Further, this may significantly decrease the acceptance of these tools by developers [11], [20].

Besides the context, we observed that the 22 true positives, which use a string instead of a byte-array, may result in *effective false positives*. While it is possible to read user input as a byte-array, some credentials are passed by API design as a string instead of the byte-array. Thus, the developer has no effective and easy way to fix the misuse by hand.

Our analysis revealed another potential source for *effective false positives* not considered by previous studies. Some misuses may be intentionally introduced in the projects contained in our data set. For example, the static analyzer SonarSource[11] includes tests, not following the typical naming scheme for test, for correct API usages and insecure crypto misuses. These misuses, in total 13 in our sample, within the tests are intended.

Research Question 1

A non-security context, the usage of string, and "intentional misuses" are common *effective false positives*.

[9]An open-source deep learning library for JVM-based languages (Stars: 12.2k, Forks: 4.9k), https://github.com/deeplearning4j/deeplearning4j

[10]Graphical tool to monitor and analyze programs running JVM (Stars: 667, Forks: 278), https://github.com/uavorg/uavstack

[11]https://github.com/SonarSource/sonar-java (Stars: 755, Forks: 501)

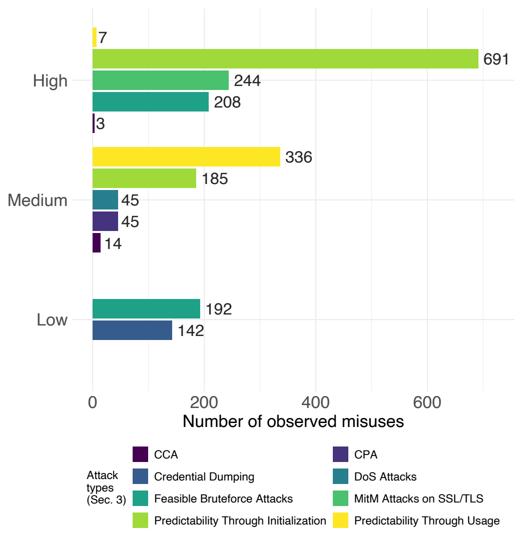


Fig. 2: Number of misuses by severity for the attack types.

D. *Vulnerabilities (RQ2)*

Based upon our vulnerability model introduced in Section III, we will discuss the most common vulnerabilities and their severity in this section. Most of the misuses are due to the attack types *Predictability Through Initialization* (876), *Bruteforce Attacks* (400), and *Predictability Through Usage* (343). In total, we identified 1,153 high-severity, 643 medium-severity, and 334 low-severity misuses. We present the number of misuses, their severity, and their relationship to the respective attack types in Figure 2.

High-severity vulnerabilities. Vulnerabilities belonging to the attack type *Predictability Through Initialization* are caused by 691 misuses spread over 108 projects. They are due to an insecurely generated key caused by a *Required Predicate* error. Another source of insecurely generated keys are insecure key generation parameters, e.g., *HmacSHA1* or *DES*. These were reported for 84 misuses within 29 projects.

The usage of SSL, TLS 1.0, and TLS 1.1 is insecure and should be avoided. We found 141 such misuses in 61 individual projects. For example, 29 misuses within 20 projects are caused by using SSL.

Medium-severity vulnerabilities. Our analysis detected 336 misuses which can result in *Predictability Through Usage* by observing the crypto function calls, their call order, and inputs. For example, the analysis identifies at least one path which consumes the crypto object without adding any data, e.g., instantiating a message digest without providing an input. Beside the predictable computations, the most relevant medium-severity vulnerabilities are due to insecure PRNGs (total: 185) and DoS attacks caused by exceptions (total: 45).

Low-severity vulnerabilities. Most (192) of the low-severity vulnerabilities are of attack type *brute-force* for ciphers, e.g., using an insecure signature algorithm or a 64-bit block cipher. For the attack type *credential dumping*, we identified 130 low-severity misuses within 47 projects due to the use of a string instead of a byte-array for passing on secrets. If an attacker has control over the system, they might retrieve the secret handled as string from memory as it is immutable and cannot be cleared from memory. Furthermore, we found 12 misuses which use a byte array for their secrets and misses to clear the memory explicitly. Thus, the secret may stay in memory as for the previously discussed secrets handled as strings.

Research Question 2

Nearly half the misuses (42.78 %) are of high-severity and should be prioritized while fixing misuses.

E. *Responsible Disclosure.*

We informed - when possible - all projects for which we could confirm a true positive from the analysis perspective. To avoid influencing the maintainers, we did not provide information about our judgment w.r.t *effective false positives*. So far, we received feedback for 22 misuses from 55 reported misuses. One project fixed the misuse that we classified as high severity and 15 projects considered the misuse that we reported as an *effective false positive*, with responses ranging over "`MD5` is used to generate test data", "the affected program component is not shipped to the customers", "the usage is secure in our setting", or "misuses in dependencies are ignored". One of these projects added documentation to the identified insecure usage, and initiated an internal analysis of their code for crypto misuses. Overall, the disclosure confirms our observation that a more fine-grained assessment of misuses is important, e.g., considering a context. For five misuses, we received no final decision from the maintainers, e.g., they only thanked us for the report, and for one misuse we were asked to provide a fix that requires to change their API. Further, some discussions raised the questions which entity, e.g., the maintainers or the user of an application, is responsible for fixing a misuse.

V. Related Work

Previous studies on crypto misuses have primarily focused on Android applications and identified that 88 % to 99 % of the applications using crypto have at least one misuse [5], [16], [13], [19], [18]. Beside Android applications, Krüger et al. [13] inspected maven artifacts, mostly Java libraries. To demonstrate the scalability of their tool *CryptoGuard*, Rahaman et al. [19] analyzed 46 Apache projects in addition to their set of Android applications. All studies focused on precision from the tool's perspective, rather than understanding false positives from the developer's or security analyzer's perspective. An exception is an exploratory study [10] that opened issues for some identified misuses. The results obtained mostly agree with the feedback we received from the disclosure as well as with our manual analysis. In contrast, we not only rely on the judgement of developers by manually analyzing the misuses.

Beside Java, IoT firmware, iOS applications and Python projects have been analyzed, revealing that nearly one fourth to 82 % of the software is vulnerable [28], [15], [7], [25].

Previous research about misuses in code available from Q&A platforms such as Stack Overflow reveals that 71 % to 90 % of the posts have at least one misuse [4]. Thus, 98 % of the Android Apps with code copied from Stack Overflow snippets have at least one misuse [1].

VI. Conclusion

In this paper, we presented a study of Java crypto misuses that focuses on understanding (effective) false positives and the severity of misuses. To understand common *effective false positives* that arise from misuses, we manually reviewed a random sample of misuses and identified several *effective false positives* such as the ones happening in a non-security context. We provide an extensive threat model covering previously undiscussed vulnerabilities such as DoS attacks in the context of crypto API misuses. Our threat model marks nearly half of the misuses as high-severity. For example, 29.05 % of the applications still use SSL, TLS 1.0, or TLS 1.1. and may be exploitable remotely.

Overall, our study reveals several directions for future work. While we got first insights into the differences of misuses between the JCA and BC, future research can answer the question if the BC API supports developers to write more secure code. Further, our manual analysis reveals several sources of *effective false positives* that can be addressed in existing static analysis tools.

Acknowledgments

We want to thank the anonymous reviewers and everyone who supported us on our journey to publish this work.

This research work has been founded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1119 – 236615297 and – SFB 1053 – 210487104 – and by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE, by the LOEWE initiative (Hesse, Germany) within the emergenCITY center.

References

[1] Acar, Y., Backes, M., Fahl, S., Garfinkel, S., Kim, D., Mazurek, M.L., Stransky, C.: Comparing the Usability of Cryptographic APIs. In: IEEE Symposium on Security and Privacy. SP, USENIX Association (2017)

[2] Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y.: Evaluating static analysis defect warnings on production software. In: ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. PASTE, ACM (2007)

[3] Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In: Annual International Cryptology Conference. CRYPTO, Springer (1998)

[4] Braga, A., Dahab, R.: Mining Cryptography Misuse in Online Forums. In: IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE (2016)

[5] Egele, M., Brumley, D., Fratantonio, Y., Kruegel, C.: An Empirical Study of Cryptographic Misuse in Android Applications. In: ACM Conference on Computer & Communications Security. CCS, ACM (2013)

[6] Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., Smith, M.: Why eve and mallory love android: An analysis of android ssl (in)security. In: ACM Conference on Computer and Communications Security. CCS, ACM (2012)

[7] Feichtner, J., Missmann, D., Spreitzer, R.: Automated Binary Analysis on iOS: A Case Study on Cryptographic Misuse in iOS Applications. In: ACM Conference on Security & Privacy in Wireless and Mobile Networks. WiSec, ACM (2018)

[8] Ferguson, N.: (May 2005), https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/comments/cwc-gcm/ferguson2.pdf

[9] Gao, J., Kong, P., Li, L., Bissyandé, T.F., Klein, J.: Negative results on mining crypto-api usage rules in android apps. In: International Conference on Mining Software Repositories. MSR, IEEE/ACM (2019)

[10] Hazhirpasand, M., Ghafari, M., Nierstrasz, O.: Java cryptography uses in the wild. In: 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM (2020)

[11] Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R.: Why don't software developers use static analysis tools to find bugs? In: International Conference on Software Engineering. ICSE, IEEE (2013)

[12] Klima, V., Rosa, T.: Side channel attacks on cbc encrypted messages in the pkcs# 7 format. IACR Cryptol. ePrint Arch. (2003)

[13] Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M.: Crysl: An extensible approach to validating the correct usage of cryptographic apis. In: IEEE Transactions on Software Engineering. TSE, IEEE (2019)

[14] Lazar, D., Chen, H., Wang, X., Zeldovich, N.: Why Does Cryptographic Software Fail?: A Case Study and Open Problems. In: Asia-Pacific Workshop on Systems. APSys, ACM (2014)

[15] Li, Y., Zhang, Y., Li, J., Gu, D.: icryptotracer: Dynamic analysis on misuse of cryptography functions in ios applications. In: International Conference on Network and System Security. NSS, Springer (2015)

[16] Muslukhov, I., Boshmaf, Y., Beznosov, K.: Source Attribution of Cryptographic API Misuse in Android Applications. In: Asia Conference on Computer and Communications Security. ASIACCS, ACM (2018)

[17] Nadi, S., Krüger, S., Mezini, M., Bodden, E.: Jumping through hoops: why do Java developers struggle with cryptography APIs? In: International Conference on Software Engineering. ICSE, ACM (2016)

[18] Piccolboni, L., Guglielmo, G.D., Carloni, L.P., Sethumadhavan, S.: CRYLOGGER: Detecting Crypto Misuses Dynamically. In: Symposium on Security and Privacy (SP). IEEE (May 2021). https://doi.org/10.1109/SP40001.2021.00010

[19] Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., Yao, D.: CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. In: Conference on Computer and Communications Security. CCS, ACM (2019)

[20] Sadowski, C., Van Gogh, J., Jaspan, C., Soderberg, E., Winter, C.: Tricorder: Building a program analysis ecosystem. In: International Conference on Software Engineering. ICSE, IEEE (2015)

[21] Spinellis, D., Kotti, Z., Kravvaritis, K., Theodorou, G., Louridas, P.: A Dataset of Enterprise-Driven Open Source Software. In: International Conference on Mining Software Repositories. MSR, ACM (2020)

[22] Stevens, M., Bursztein, E., Karpman, P., Albertini, A., Markov, Y.: The first collision for full sha-1. In: Annual International Cryptology Conference. CRYPTO, Springer (2017)

[23] Vaudenay, S.: Security flaws induced by cbc padding—applications to ssl, ipsec, wtls... In: Advances in Cryptology. EUROCRYPT, Springer (2002)

[24] Wagner, S., Jürjens, J., Koller, C., Trischberger, P.: Comparing bug finding tools with reviews and tests. In: Testing of Communicating Systems. TestCom, Springer (2005)

[25] Wickert, A.K., Baumgärtner, L., Breitfelder, F., Mezini, M.: Python crypto misuses in the wild. In: International Symposium on Empirical Software Engineering and Measurement. ESEM, ACM (2021)

[26] Wijayarathna, C., Arachchilage, N.A.G.: Using cognitive dimensions to evaluate the usability of security APIs: An empirical investigation. In: Information and Software Technology. IST, Butterworth-Heinemann (2019)

[27] Wu, J., Liu, S., Ji, S., Yang, M., Luo, T., Wu, Y., Wang, Y.: Exception beyond exception: Crashing android system by trapping in "uncaught exception". In: International Conference on Software Engineering: Software Engineering in Practice Track. ICSE-SEIP, IEEE (2017)

[28] Zhang, L., Chen, J., Diao, W., Guo, S., Weng, J., Zhang, K.: Cryptorex: Large-scale analysis of cryptographic misuse in iot devices. In: International Symposium on Research in Attacks, Intrusions and Defenses. RAID, USENIX Association (2019)

jGuard: Programming Misuse-Resilient APIs

Simon Binder
Technical University of Darmstadt
Darmstadt, Germany
simonuwe.binder@stud.tu-darmstadt.de

Krishna Narasimhan
Technical University of Darmstadt
Darmstadt, Germany
kri.nara@st.informatik.tu-darmstadt.de

Svenja Kernig
Technical University of Darmstadt
Darmstadt, Germany
svenja.kernig@gmx.de

Mira Mezini
Technical University of Darmstadt
Darmstadt, Germany
mezini@informatik.tu-darmstadt.de

Abstract

APIs provide access to valuable features, but studies have shown that they are hard to use correctly. Misuses of these APIs can be quite costly. Even though documentations and usage manuals exist, developers find it hard to integrate these in practice. Several static and dynamic analysis tools exist to detect and mitigate API misuses. But it is natural to wonder if APIs can be made more difficult to misuse by capturing the knowledge of domain experts (i.e, API designers). Approaches like CogniCrypt have made inroads into this direction by offering API specification languages like CrySL which are then consumed by static analysis tools. But studies have shown that developers do not enjoy installing new tools into their pipeline. In this paper, we present jGuard, an extension to Java that allows API designers to directly encode their specifications while implementing their APIs. Code written in jGuard is then compiled to regular Java with the checks encoded as exceptions, thereby making sure the API user does not need to install any new tooling. Our evaluation shows that jGuard can be used to express the most commonly occuring misuses in practice, matches the accuracy of state of the art in API misuse detection tools, and introduces negligible performance overhead.

CCS Concepts: • **Software and its engineering** → *API languages*.

Keywords: API, DSL, Java

ACM Reference Format:

Simon Binder, Krishna Narasimhan, Svenja Kernig, and Mira Mezini. 2022. jGuard: Programming Misuse-Resilient APIs. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22), December 06–07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3567512.3567526

SLE '22, December 06–07, 2022, Auckland, New Zealand
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9919-7/22/12...$15.00
https://doi.org/10.1145/3567512.3567526

1 Introduction

Software developers typically rely on a wide range of libraries written by others and made available via application programming interfaces (APIs). Using APIs, developers can focus on their application instead of reimplementing common functionality over and over again [29]. Moreover, as libraries are typically developed by experts in their domain, their implementation are likely to be more secure and reliable than what developers without domain-specific expertise would write. But for libraries to actually make developers more productive and applications more reliable, they must be used *correctly* – which, in practice, turns out to be challenging [33]. Commonly, an *API misuse* is defined as application code failing to comply with usage constraints of the API [31]. For example, consider the `Cipher` class from Java's standard cryptographic library, JCA [15] and its allowed API call sequencing: An encrypting cipher can be initialized with a call to `init`. Then, chunks of plaintext byte data can be encrypted with `doUpdate` before the encryption is finalized with `doFinal`. Further, it is required that the cipher may not be initialized more than once. Unfortunately, usage constraints like those sketched above for `Cipher` are typically not explicitly defined, let alone automatically enforced. Such *API misuses* can lead to runtime crashes, security vulnerabilities, or other defects [19, 23, 30]. As the importance of using APIs increases [29], misuses tend to become more costly. For example, the video-conferencing software Zoom used an non-secure configuration of the AES encryption algorithm [24]; among other issues, this led to an US$ 85M class-action lawsuit[1].

One approach to reduce misuses is to improve the documentation of the API. But past research has shown that the documentation [17] of a library alone is not typically helpful to developers [11]. Developers may also consult public forums to discuss correct API usages. However, research has shown that answers on forums such as StackOverflow typically contain API misuses too [39]. Static analysis approaches have been proposed to detect API misuses in application code. Traditionally, they employ *deny-listing*, i.e, the program is scanned for code patterns that represent library misuses. CryptoLint [13] is an example of this approach. The problem with deny-listing analyzers is that they can only detect a fixed hardcoded subset of all API misuses [8]. To mitigate this problem, *allow-listing* approaches have been proposed. Static analysis take a customizable set of usage rules specified by domain experts as a parameter and search for deviations from these patterns in application code. An example of an allow-listing approach for API misuse detection is CogniCrypt [19], which is fed with rules written in the API usage specification language CrySL [20]. Allow-listing approaches have their own problems, too. While rule sets make an analyzer more versatile, they need to be updated continuously to keep up with the API's evolution and must be very accurate to minimize false-positive reports. Allow-listing approaches also have the problem that the specification language is decoupled from the API's implementation which may be a deterrent for API designers (Problem **P1**). A solution to (**P1**) could be to use a design-by-contract approach like JML [21] (an allow-listing variant) that allow API designers to specify constraints using annotations. But such constraints, among other things, target functional requirements instead of API usage constraints and do not have first-class support for important API usage patterns like type-states. A problem with both allow-listing and deny-listing approaches is that application programmers may be reluctant to add additional tooling into their workflows [16] (**P2**). Both kinds of static analysis for API misuse detection perform poorly on code with complex control flow, or on misuses resulting out of methods called in incorrect order [4] **(P3)**.

In this paper, we propose an alternative approach to mitigating API misuses. Instead of relying on external detectors to statically analyze application code, we provide library developers with programming support for APIs that are resilient to misuses by-design by allowing them to embed usage contracts into the API's implementation (addressing **P1**). Compliance to these contracts is checked as part of standard application testing (addressing **P2**). Our approach is catered specifically to the main categories of API uage constraints and has first-class support for constraints like type-states that express the order in which method sequences must be invoked **(P3)**. More specifically, we introduce jGuard, an extended version of Java that allows API developers to express predicates on parameters and guards that dictate the expected flow of method calls, among other things. jGuard enables library authors to specify very precise usage constraints in the API implementation. A compiler translates jGuard annotations into runtime checks in Java source code and library developers ship guarded versions of their libraries. jGuard is fully backward-compatible with Java and allows incrementally annotating a library. We implemented jGuard as an extension to Java 11 using the Jetbrains Meta-Programming System [37].

Our proposal has several benefits. First, it can detect misuses more accurately than static analyzers, as it can reason about the runtime state of API objects. Second, since the API implementation and contracts are developed together, deviations from intentions of the API designer and the encoded usage contracts is reduced. We evaluated jGuard along three axes: expressiveness, accuracy, and performance impact. We show that jGuard can express the most frequently occuring misuses according to the state of the art API misuse benchmark. jGuard has a higher recall than a state-of-the-art API misuse detector without reporting any false positives (in comparison to 67 FPs). Our performance evaluation on real world projects and micro-benchmarks simulating worst case scenarios indicate that introducing additional checks on APIs has negligible impact on performance. In summary, the contributions of the paper are:

1. We design and implement a extension to Java capable of expressing the most important API usage constraint categories in Section 2.
2. We compare the expressiveness of jGuard against the state of the art in API usage specification languages in Section 3
3. We evaluate jGuard based on its ability to express the top-10 misuses occuring in the wild from all domains as per the state of the art API misuse benchmark, establish that checks introduced by jGuard does not introduce major performance overhead and compare the results with CogniCrypt, a state of the art API misuse detector in the security domain in Section 4

Section 5 discusses threats to validity. Section 6 presents related work, and we conclude our results in Section 7.

Data availability: A virtual machine image running jGuard is made available.[2] A video explaining how to reproduce the

[1]https://threatpost.com/zoom-settlement-85m-security-investment/168445/

[2]https://doi.org/10.5281/zenodo.5767812

language features of jGuard is also pulished.[3] For readers interested in trying out jGuard, we recommend setting up jGuard from sources. [4]

2 Language Extensions

This section motivates and presents the categories of API usage patterns followed by a high-level overview of jGuard's main concepts and how they map to the API usage patterns. This is then followed by a fairly detailed look into the individual language extensions and a discussion on how jGuard can be used in practice by API designers.

2.1 API Misuse Categories

In this section, we curate categories of API misuse based on categories established by *MuC*, a misuse categorization considered state of the art [4], and Robillard et al. [32] who conducted a survey of 60 techniques over a decade and consolidated the API usage properties. (***M** indicates the category comes from MuC and **R** indicates the property comes from Robillard et al.'s work.*):

1. *Unordered usage pattern **(R)*** are high-level patterns that describe what kind of API elements (classes, methods etc.) co-occur.
2. *Method calls **(M)*** are usage patterns that describe the order in which method calls need to occur and can be further sub-categorized into *Sequential usage patterns **(R)*** that specify in what order method calls of a single API object must be called in, and its variant *multi-object properties **(R)*** that involves multiple API objects.
3. Missing *conditions **(M)*** occur when the usage does not satisfy mandatory constraints. This category is analogous to the *Behavioral specifications **(R)***.
4. Usage patterns can be also be categorized based on their dependence on *iterations **(M)*** and *exception handling **(M)***.
5. Finally, *Migration mappings **(R)*** do not explicitly talk about usage constraints, but about potential mappings required of client code to support a different API version.

2.2 High-level Design of jGuard Concepts

Like traditional APIs, jGuard-based APIs provide reusable domain-specific functionality. But in addition, internally they also keep track of and reason on how they are being used to self-guard their execution and to enforce usage constraints. For illustration, consider how an API that provides the functionality of JCA's `Cipher` class would be modeled with jGuard in Figure 1. The developer of a jGuard-annotated API declares classes and methods, whose usage needs to be constrained, as verified (syntactically expressed with the `verified` annotation).

```
1   public verified class Cipher {
2     private guard isInitialized;
3     private guard encryptionStarted finally false;
4
5     public verified void init(@NonNull Key key)
6        require !encryptionStarted, !isInitialized
7        when returns sets isInitialized,
8        when returns sets encryptionStarted {}
9     public verified byte[] doUpdate(byte[] data)
10       require encryptionStarted {}
11    public verified byte[] doFinal(byte[] data)
12       require encryptionStarted
13       when returns resets encryptionStarted {}
14  }
```

Figure 1. jGuard-annotated variant of the `Cipher` class

A **verified class** is the top-level unit for all checks against misuses. That is, only verified classes may contain jGuard language extensions like **verified methods** and **guard** declarations in their definition. A verified class declaration without jGuard-specific members behaves identical to a regular Java class with the same properties and members. Verified classes can declare **guard** variables to be used as gatekeepers for enforcing correct method call orders. As an example, the implementation of `Cipher` in Figure 1 introduces the `isInitialized` **guard** (Line 2), which tracks whether a `Cipher` object has been initialized. A **verified method** is a Java method with usage constraints that are checked at runtime. As already mentioned, a verified method declaration is only allowed in a verified class. Expanding the syntactic definition of regular Java methods, verified methods may define **requirements** and **consequences** introduced by jGuard as part of their headers (in the vein of `throws` declarations in standard Java method headers). **Requirements** and **consequences** refer to guards using **guard references**. At high level, the combination of using requirements to query for the state of guards and of consequences to update the state of guards is used to express API usage constraints. **Requirements** serve as gatekeepers for invocations of individual methods. As an example, the `init` method requires that the guard `isInitialized` is not set (Lines 5-6). Requirements in jGuards can contain any Java boolean expression and can refer to guards like they refer to any other variable. This is possible because jGuard is embedded into Java rather than being a new language from scratch[5]. Verified methods can also declaratively state the effect of their execution on the usage state of the receiver object. This is what the **consequences** declaration in their

[3] https://youtu.be/tnQGMyMjZRA
[4] https://github.com/jguardnm86/sle22

[5] The implementation of jGuard relies on the composable language features offered by the Jetbrains MPS workbench. [36]

header is for. The consequences are expressed as a reaction to triggers (relevant events regarding usage history, such as returning from a method execution with a particular value). For instance, the `init` method declares that on returning from its execution, the receiver `Cipher` object has been initialized (`when returns sets isInitialized` in Line 7). Methods may also reset guards in their consequences, e.g., `when returns resets encryptionStarted` (Line 13). A verified method declaration without requirements or consequences has the same semantics as a Java method declaration with the same type parameters, annotations, visibility, return type, name, arguments and throws declaration.

Concepts of jGuard are designed to be able to express the API usage patterns described in Section 2.1. *Unordered usage pattern* can be described by **verified** annotations on top of classes and methods. *Method calls* can be checked by expressing valid method sequences through **Guards** (Section 2.3). Missing *conditions* can be found through preconditions expressed as **Requirements** (Section 2.4). Misuses related to *iterations* can be detected by validating repeated method invocations with guards. It is possible to specify usage patterns on *exception handling* by changing a guard in response to a method completing exceptionally using **Consequences** (Section 2.5). Usage constraints change during API evolution, and such variable specifications can be specified and modularly maintained using **Meta Variables** (Section 2.6) thereby expressing *Migration mappings*.

jGuard use in practice: The jGuard compiler generates runtime checks from verified declarations while translating to Java and produces two versions of the API – with or without the checks enabled. Application developers may then use the version of a library with checks enabled during development and testing to detect and avoid potential misuses of the API in their code. They can subsequently integrate the version of the API with checks disabled for deployments to avoid runtime performance costs and increased code size.

2.3 Guards

Certain types of objects require that methods are invoked in a specific order. For instance, it is invalid to invoke further methods on an `InputStream` after calling `close`. The `Cipher` class from Figure 1 also requires its methods to be invoked in a specific order. For instance, `doUpdate` and `doFinal` must be called after `init`. Valid invocation patterns are typically expressed through *method sequences* [20], which describe the order in which methods are allowed to be called. Valid method sequences can be seen as an implicit state of an object with the current state describing which method calls are legal. Invoking a method may change an object's state. jGuard offers guards to make this implicit state explicit. Like other members of a class, guards can be declared as `static` or `instance`.

In Figure 1, two guards (`isInitialized`, `encryptionStarted`) are declared inside a verified `Cipher`. Their names must be unique inside classes. Similar to fields in Java, instance guards are inherited by subclasses. Guards have a default value of `false` to prevent issues arising out of uninitialized use. Finalizers on guards may be used to declare that guards must be in a specific state at the end of an object's lifecycle. The `encryptionStarted` guard has a finalize clause to express that encryption operations must always be completed. A guard can be referenced just like any field. Guard references may only be used inside as part of a requirement or consequence. In the example from Figure 1, guard references are used as requirements to verify if a method call is valid. The method `doUpdate` has a requirement that refers to the guard `encryptionStarted` to ensure updates do not happen before encryption has commenced.

2.4 Requirements

Requirements are a jGuard language construct to express preconditions that need to be verified before a method is invoked. As already mentioned, only boolean expressions can be used in a **requirement**. Any **requirement** expression that evaluates to `false` signals to the jGuard-compiler that a contract on the method has been violated. Requirements are evaluated in order and an exception is thrown on the first requirement that does not evaluate to `true`. jGuard allows annotating constructors with **requirement** as well.

2.5 Consequences

Guards can only be set or reset using method **consequences** . While requirements are checked prior to method invocations, consequences allow reasoning about the result of method executions. Consequences are declared in method headers and execute **Actions** as reactions to **Triggers**. jGuard supports different kinds of triggers that dictate when a consequence's actions are executed.

- A ***Condition Trigger*** with an input parameter *e* matches a method's invocation, if it completed normally without throwing an exception and if *e* evaluated to `true`.
- A ***Throws Trigger*** matches method invocations that threw an exception either because a requirement failed or because evaluating the declared method body threw a specific exception.
- A ***Returns Trigger*** matches method invocations that return a specified value.

jGuard performs type checks to prevent incorrect triggers. For instance, the exception type of a throws trigger must be a subtype of `java.lang.RuntimeException`, `java.lang.Error` or any type declared in the `Throws` section of the enclosing

```
public verified class SafeIterator<T> {
  private guard canCallNext;
  public verified boolean hasNext()
    when returns true sets canCallNext {// ...}
  public verified T next() require canCallNext
    when returns resets canCallNext {// ...}}
```

Figure 2. Code snippet showing guards against Iterator misuse

```
state IsSecureKey for Key {
  boolean isSecure = false;}
public verified void init(@NonNull Key key)
  requires IsSecureKey(key).isSecure, ...
```

Figure 3. Example for an external state definition in cryptographic contexts

verified method. After a trigger has been matched, its corresponding actions are executed. Actions can be either be setting or resetting guards. Actions are executed in the order in which they have been declared.

Relationship to type-state analysis: Guards and using them in runtime requirements can be seen as a dynamic implementation of **type-state analysis** and explicitly encodes an object's state describing whether certain operations are valid. In Figure 2, a guard describes whether a call to `Iterator.next()` is safe and consequences describe how the object's state changes with method invocations. By allowing arbitrary Java expressions in requirements and consequence filters, complex access patterns can be described in addition to type-state requirements.

External object states: In exceptional cases, it may be desirable to express requirements for a non-verified class. In particular, classes defined in external libraries or the JDK cannot be modified easily. The `Cipher` class from Figure 1 may want to enforce that a key passed to `init` has been generated from a secure key generator [13]. The `Key` class is provided by the JDK and its jar in the client application cannot be modified. Figure 3 illustrates how an external state definition of the type `Key` can be introduced by defining the external `isSecure` parameter which defaults to `false` for new instances of `Key`. In the `init` method of the cipher, a requirement then asserts that the external state has been set for the key thereby confirming that it is secure.

Compiling Guards, Requirements and Consequences: Now, we describe how guards, requirements and consequences are compiled from jGuard into regular Java. As guards can either be set or reset, they naturally compile to boolean fields. Each guard declaration generates a new field with the same annotations and modifiers as the guard. If the guard has

```
public void init(@NonNull Key key) {
  if (key == null) reportMisuse();
  if (encryptionStarted) reportMisuse();
  if (isInitialized) reportMisuse();
   // <original body of the method>}
```

Figure 4. Statements generated for requirements

```
public verified class Consq {
 verified boolean isValid (String key) throws T1 when
      returns true sets g1 when throws T1 resets g2  {
   <body>}}
//Code below generated by jGuard
private boolean isValid_$mangled(String key) throws T1
  {Old body}
boolean isValid(String key) throws T1 { try {
    var retVal = isValid_$mangled(key);
    if(Objects.equal(retVal, true)) {set g1}
    return retVal;}
    catch (Error|RuntimeException|T1 e) {
      if(e instanceof T1){ reset g2}
      throw e;}}
```

Figure 5. Statements generated for consequences

been declared with an **initial value**, a matching initializer is added to the field. Requirements are compiled to Java by inserting `if` statements at the beginning of a method. Violated requirements are then reported as misuses. Figure 4 highlights the statements jGuard would generate for the requirements of `init` from Figure 1.

As one would expect, the semantics of verified constructors are similar to verified methods except that the requirements for constructors are checked *after* executing the body of a verified constructor in order to comply with Java specifications on constructors.

To illustrate how consequences are compiled, consider the example in Figure 5. The class `Consq` has one method `isValid` which sets `g1` when the method returns `true` and resets `g2` when the exception `T1` is thrown. In the compiled version, the original method `isValid` now serves as a wrapper to a newly generated method `isValid_$mangled`, which now contains the body of the original method. The wrapper `isValid` executes the `isValid_$mangled` and evaluates the guards. Depending on the **Trigger** used in a consequence, the generated code is placed in the `try` section (for triggers matching return values) or in the `catch` section (for triggers matching exceptions). A **Returns Trigger** is transformed into an equivalent **Condition Trigger** as follows. A **Returns Trigger** without an additional expression is compiled to a **Condition Trigger** with a literal `true` as an expression. A **Returns Trigger**

```
public verified class Cipher {
  meta Set<String> allowedAlgorithms;
  public static verified Cipher getInstance(String algorithm)
    requires allowedAlgorithms.contains(algorithm) {}}
instantiation FIPS {
  HashProvider.allowedAlgorithms.add("TrippleDES");}
instantiation BSI {
  HashProvider.allowedAlgorithms.add("AES");}
```

Figure 6. Example for meta-variables used to express variability of acceptable algorithms in BSI and FIPS

with an additional expression *e* is compiled to a conditional trigger with the expression `Objects.equal(return value, e)`. This condition is then used to create an `if` statement executing the matching `Action`. For multiple consequences in the same block, an `if/else if` chain is generated to ensure only the first consequence is matched. A **Throws Trigger** compiles into an `instanceof` check of the exception caught in the `catch` block. A guard `reset` is transformed into an assignment of the field representing the guard.

2.6 Meta Variables

API usage constraints may vary depending on the usage contexts. For instance, code adhering to standards set by the German *BSI* is not allowed to use the *SHA-512/224* configuration of the *SHA2* algorithm, while this is allowed in code following the *FIPS* standard set by the American *NIST*. Other factors may influence the API usage constraints as well, such as versions (e.g., whether the application is running on Android 10 or 11). When annotating an API with jGuard, a library designer is able to write rules flexible enough to express these constraints. In Figure 6, the `algorithm` parameter of the `getInstance` method is bound with the meta-variable `allowedAlgorithms` which is then refined to specific values using `instantiations`.

2.7 Relationship to Design by Contract

Even though jGuard is inspired by design-by-contract approaches like JML, there are a few key differences. jGuard's grammar and specification are much smaller than JML, potentially making it easier to use. The code that jGuard is generated into is easy to reason about (one field per guard, requirements compiled into checks at the start of the method). jGuard is aligned specifically to API design whereas JML leans more towards formal verification. JML specifications target functional correctness only (e.g., preconditions of the values of parameters to a method). jGuard supports expressing such constraints, but also others that are specific to usage correctness as opposed to functional correctness, e.g., type state like constraints (guards). Although, JML can simulate guards by re-using variables inside annotations, their values cannot be modified as reactions to events like returns or exceptions thereby making type-state specifications infeasible.

JML is a modelling language on top of an implementation language from which checks for testing and verification tools are generated. jGuard is an implementation language embedded directly into the language in which the API is also implemented (Java) and it is used by the API implementer alongside the latter to implement the library so that it is hardened against misuses by design. Within jGuard, concepts like guards, requirements and consequences are first-class citizens enabling language engineering tooling support (e.g., detecting use of undefined guards). Finally, JML requires tooling for every developer trying to benefit from it, jGuard just requires the library developer to adopt it.

3 JGuard's Expressiveness Compared to CrySL

To measure the effectiveness of JGuard, we compare it to a state of the art API usage specification language, CrySL. To the best of our knowledge, CrySL is the most expressive language when it comes to specifying usage patterns described by MuC and Robillard et al. We first describe the sections of CrySL using the example in Figure 7, map them to the API usage patterns and finally compare the expressiveness of jGuard with that of CrySL.

On a high level, every section in a rule corresponds to a particular type of API misuse violation. **OBJECTS** and **EVENTS** gather parameters and methods that can be violated for a particular type. These allow expressing **unordered usage patterns**. Each pattern in the **EVENTS** section is of the form `<label: method(parameters)>`. These labels can further be grouped using aggregates of the form `<aggregate := labels>`. The events section further defines named method parameters whose values can be constrained later, in the **CONSTRAINTS** section . The **ORDER** section describes the order in which methods of a particular type must be invoked. It is defined as a regular expression of method-event patterns forming a *usage pattern*. The methods used in this section are labels/ aggregates defined in the **EVENTS** section. In our example, a correct use of a **KeyGenerator** object should contain one call to the methods aggregated as **Gets** followed optionally by a call to **Inits** and finally a call to methods aggregated by **gk**. This allows one to express **Sequential usage patterns**. The **CONSTRAINTS** section defines all parameter constraints for parameters defined in the **EVENTS** section. This allows one to express **Behavioral specifications**. **ENSURES** and **REQUIRES** clauses describe how objects of different types must be correctly composed. By rely/guarantee-reasoning, they allow one to

```
SPEC javax.crypto.KeyGenerator
OBJECTS
int secretKeySize; java.security.spec.
     AlgorithmParameterSpec params; javax.crypto.SecretKey
     key; java.lang.String secretKeyAlgorithm; java.
     security.SecureRandom ranGen;
EVENTS
g1: getInstance(secretKeyAlgorithm); g2: getInstance(
     secretKeyAlgorithm, _); Gets := g1 | g2;
i1: init(secretKeySize);
i2: init(secretKeySize, ranGen); i3: init(params);
i4: init(params, ranGen);
i5: init(ranGen);
Inits := i1 | i2 | i3 | i4 | i5;
gk: key = generateKey();
ORDER
Gets, Inits?, gk
CONSTRAINTS
secretKeyAlgorithm in {"AES", "HmacSHA224", "HmacSHA256", "
     HmacSHA384", "HmacSHA512"}; secretKeyAlgorithm in {"
     AES"} => secretKeySize in {128, 192, 256};
REQUIRES
randomized[ranGen];
ENSURES
generatedKey[key, secretKeyAlgorithm];
```

Figure 7. CrySL specification for the `KeyGenerator` class

express the **multi-object properties** variant of **Sequential usage patterns**, which span multiple API objects. A detailed documentation of the CrySL language can be found on its official website [18]. Now, we show that for every section of a CrySL spec, we can construct equivalent jGuard annotated code.

Expressing CrySL's ORDER in JGuard. The `ORDER` section of a CrySL rule corresponds to a deterministic finate automaton (DFA) $\mathcal{A}^o = (Q, \mathbb{M}, \delta, q_0, F)$ with $Q = \{q_0, \dots, q_n\}$ defining a language $L(\mathcal{A}^o) \subseteq \mathbb{M}^*$ of allowed method sequences [20]. We construct a verified implementation of a class for every such DFA and complete the construction by establishing that every misuse reported by the DFA will also be reported by the constructed verified class (by showing semantic equivalence between the constructed verified class and the corresponding CrySL DFA). Each state in the CrySL DFA can be represented by a private guard added to the class. Depending on the state q_i, the guard is declared as follows: For each state q_i, we introduce a private guard named g_i. If $i = 0$, the guard g_0 is declared to be `initially set`. If $q_i \notin F$, the guard g_i is declared to be `finally reset`.
Next, we define changes to each verified method $m \in \mathbb{M}$ necessary to implement state transitions defined in the DFA of the CrySL rule $\mathcal{A}$. The method m is replaced with a verified method, copying its original visibility, return type, type parameters, parameters, thrown types and body. For each pair $(q_i, q_j) \in Q^2$ with $\delta(q_i, m) = q_j$, a **Consequence** is added to m. The consequence has a **ConditionTrigger** matching q_i and changing the currently active guard from g_i to g_j. In jGuard, the syntax for such consequence is `when` g_i `resets` g_i `,sets` g_j.

Next, to show that the semantics of the constructed verified class matches the runtime semantics of the CrySL DFA, let m^o be a sequence of method invocations on an instance of the annotated class. Further, let $(s_0, \dots, s_{n-1}) \in Q^+$ be the trace of states taken in $\mathcal{A}$ when running over m^o. That is, $s_0 = q_0$ and $\delta(s_i, m_i) = s_{i+1}$ for all $0 \le i < |m^o| = n$. Per induction, we show that for each such i, $s_i = q_j$ iff the guard g_j is set. When the mapped class is instantiated, the DFA is in its initial state q_0. By construction, all guards are not set except for g_0 corresponding to the initial state. For $i > 0$, the automata is in the state $s_i = q_j$ if it has been in the state $s_{i-1} = q_k$ before the method invocation m_i. As per the induction hypothesis, this implies that g_k was the only guard active before the method invocation m_i. When the method returns, its consequences are evaluated. As g_k is the only guard set, only the consequence introduced for $\delta(q_k, m_i) = q_j$ will match. This consequence then resets g_k and sets g_j, satisfying the induction hypothesis. Having established that the guards in a method sequence match states taken by $\mathcal{A}$, we must show that a misuse is reported iff $s_{n-1} \notin F$ (the final states). If $s_{n-1} = q_f$ is in F, the guard g_f is the only guard active at the end of the method invocation trace. As g_f has been constructed without a `finally` clause, no further checks are performed by jGuard and no misuse is reported. On the contrary, if $s_{n-1} = q_e \notin F$, the single set guard g_e has been declared to be `finally reset`. The semantics of jGuard mandate a misuse being reported in this case. As the construct shown here reports a misuse iff $m^o \notin L(\mathcal{A})$ (method sequence is not valid in the language defined by the CrySL's DFA), it matches the runtime semantics of CrySL.

Expressing CrySL's Constraints in JGuard. Here, we will define a mapping from every CrySL constraint to a corresponding jGuard expression which is then used in an appropriate requirement.

In CrySL, a constraint is a boolean expression on specified objects. Specified objects are declared as variables. Variables are then bound in response to an event. Each set of bound variables must adhere to all constraints specified in a rule. One does not need to evaluate all constraints for each event to adhere to CrySL's semantics. An equivalent check is to evaluate only those constraints referencing variables that might have changed. We now construct a mapping τ mapping CrySL constraint c to jGuard expressions. Using this expression in a **Requires** section for the method m will then yield an equivalent check. For each method m mentioned as an event in a CrySL rule, let C_m be the set of constraints. For each $c \in C_m$, we now define $\tau(c)$ as follows[6].

- If c is of the form `c₁ => c₂`, then $\tau(c) = !\tau(c_2)|\tau(c_1)$. This is equivalent, as $a \implies b \iff \neg a \lor b$.
- If c is of the form `c₁<OP>c₂` with <OP> being `||`, `&&`, `+`, `-`, `%`, `*`, `/`, `<`, `<=`, `>`, `>=`, `!=` or `==` then $\tau(c) = \tau(c_1)\langle OP\rangle\tau(c_2)$.
- If c is of the form `c₁.ID` for an identifier *ID*, then $\tau(c) = \tau(c_1).ID$

Next, each method m with $C_m \neq \emptyset$ is turned into a verified method. All requirements $\tau(c) \mid c \in C_m$ are added to the verified method. This yields a construction semantically equivalent to the CrySL rule.

CrySL supports a predicate `notHardCoded(e)`, which evaluates to `true` iff the expression e is not hard coded in the program's source, e.g. through a string literal. As this cannot be detected at runtime reliably, jGuard does not have a comparable mechanism and this predicate cannot be translated. However, it should be noted that the `notHardCoded` predicate in CrySL cannot be fully accurate either. It is designed to express guards against static cryptographic keys embedded in the application. Still, it would fail to detect constants being loaded dynamically, for instance by reading a file contained in the application's JAR through a class loader. These values are still conceptually hard coded, without being detectable by a CrySL rule.

Expressing CrySL's Ensures and Requires in JGuard. As defined in Section 3, the `ENSURES` section sets or resets a predicate, identified by the class annotated by a CrySL rule and a name for the predicate. Additionally, a predicate may hold arguments, which will be bound to variables in CrySL. Each predicate is defined on an instance o of the class C annotated by the CrySL rule. Further, each predicate has a name and a set of additional arguments $(a_0, \dots, a_i)$. To express CrySL predicates in jGuard, a new class and an external state declaration is introduced. The class contains a field for each parameter a_i in the predicate. The external state declaration is defined on this class. By default, predicates declared in an `ENSURES` section are added or removed in the last event of valid method sequences. In addition, CrySL allows specifying an `after` clause to perform the change after an event. As all events can be mapped to a state q in the automaton $\mathcal{A}$ as defined in Section 3, we must add or remove valid predicates in response to $\mathcal{A}$ entering such a state q. To set a predicate with argument variables $v_0, \dots, v_i$ in state q, consider all tuples $(q_p, m, q) \in Q \times \mathbb{M} \times Q$ with $\delta(q_p, m) = q$. As constructed in Section 3, the equivalent verified method will have a **Consequence** set for this state transition. To represent a predicate being fulfilled, we add an action that sets the state variable to this consequence. This consequence shall construct a new instance of the predicate class with the bound variables. Note that all CrySL variables available in this construct refer to method parameters or its return value, which means that they can be transformed to jGuard expressions available in a consequence. The state variable is then changed to a new set containing previous predicates and the new instance. To remove an active predicate, the corresponding consequence simply sets the introduced state variable to a new set not including the predicate to be removed. With ensures clauses being translated to external state declarations that are set iff the corresponding CrySL predicate is set, a `REQUIRES` section can be translated to a jGuard requirement. The requirement simply queries the external state declaration on the object to verify whether a matching predicate exists or not. An absence of this predicate will cause the requirement to evaluate to `false`, causing a misuse to be reported.

jGuard's expressive superiority over CrySL: . In addition to the requirements mapped from a CrySL constraint, jGuard **allows using every valid boolean Java expression in a requirement**, making it a lot more expressive in this regard. An obvious benefit of this is the ability to express complex preconditions based on the values of parameters (e.g., expressing the strict rules on connection string URLs in the JDBC API [2]). jGuard **enables tracking the dynamic state of an object at runtime**, which is important for frequently occurring misuses, where the value of one object is dependent on runtime values of another object. This is elaborated further with an example in Section 4.

Guards can express more valid method sequences than regular expressions supported by CrySL. A prime example for such sequences is the `Iterator` class from the JDK, where `next` may only be called once if the previous call to `hasNext` returned true. Figure 2 shows how an iterator would be guarded with jGuard. The guard `canCallNext` is set upon successful completion of the method `hasNext`, indicating the method `next` is a valid upcoming call in the sequence. A less obvious benefit (that we have not yet validated empirically) is the ability to access sophisticated solvers and analysis libraries inside requirements.

jGuard also **supports the ability to re-use complex constraints across methods**. In Figure 8, an empty method `verifyEngineParameters` models a re-usable requirement that is used across the `CipherSpi` class. Also, such formulations that reference global variables require hard-coding the built-in predicates using CrySL.

Take Away: Except for outliers like detecting hard coded constants, jGuard matches and exceeds the expressiveness of the state of the art in API usage constraint specification.

[6]We have listed a construction for a few CrySL constraints and omitted the rest for lack of space

```
private void initFromSpec(OAEPParameterSpec pSpec) throws
      NoSuchPaddingException {
  MethodBody; verifyEngineParameters();  }
private verified void verifyEngineParameters() require
     allowNoPadding || !cipher instanceof
     RSABlindedEngine, allowPKCS1Padding || cipher.
     getClass().getSimpleName() :ne: "PKCS1Encoding"{  }
```

Figure 8. Re-usable constraints using jGuard

4 Evaluation

We evaluate jGuard along the following research questions:
(Accuracy) RQ1-a: How accurate is jGuard when dealing with most frequently occurring misuses?
(Accuracy) RQ1-b: How accurately do checks generated from jGuard protect against misuses in the cryptographic domain (in comparison to CogniCrypt)?.
(Performance) R2: What is the performance impact of runtime checks generated from jGuard annotations?

(RQ1) Most frequently occurring misuses: To validate jGuard's ability to express and accurately detect common misuses, we use the MuBench dataset of API misuses. The MuBench project collects a large number of misuses found in publicly-available source code [3].

Justification for MuBench: MuBench can be considered as a valuable benchmark to base our evaluation for the following reasons. First, MuBench is to the best of our knowledge the largest automated benchmark for API-misuse detectors and is publicly open-sourced [7]. Second, MuBench is actively maintained and is the subject of many top-tier publications [8]. Finally, each reported misuse in MuBench is annotated with a description and its location in the source code.

Experiment and results: We grouped misuses from MuBench by their set of misused classes and investigated the most commonly misused classes. For this part of the evaluation, we pick the top 10 frequently occurring misuses not related to cryptography. We discuss misuses related to cryptography in RQ1-b where we also compare our results with CogniCrypt. For each of the classes belonging to a misuse category, we wrote a jGuard-annotated version of the class and manually verified whether the reported misuse matches what is expected based on the MuBench report. Consider the most frequently occuring misuse that concerns the `ByteArrayOutputStream` class and states that its objects need to be properly flushed (e.g., using the `toByteArray()` method) before they can be consumed elsewhere; this is to ensure that there are no pending writes into the stream. Figure 9 illustrates how jGuard can be used to express this constraint, we introduce a guard `hasPendingWrites`, which is set inside the ByteArrayOutputStream object's constructor (as part of the `attach` method). The `toByteArray` call expects that the guard is reset, which happens only if the `flush` method is called.

[7]https://github.com/stg-tud/MUBench
[8]https://github.com/stg-tud/MUBench#publications

```
public verified class VerifiedByteArrayOutputStream extends
      OutputStream {
  private ByteArrayOutputStream output;
  public guard hasPendingWrites;
  public VerifiedByteArrayOutputStream() { output = new
       ByteArrayOutputStream();}
  public verified void attach() when returns then sets
       hasPendingWrites {}
  public void write(int i) throws IOException { output.
       write(i);}
  public verified byte[] toByteArray() require !
       hasPendingWrites {
    return output.toByteArray();}}
  public verified class VerifiedObjectOutputStream extends
       ObjectOutputStream {
  private VerifiedByteArrayOutputStream out;
  public VerifiedObjectOutputStream(
       VerifiedByteArrayOutputStream out) throws
       IOException {
    super(out); this.out = out; out.attach(); }
  public verified void flush() throws IOException when
       returns then resets out.hasPendingWrites {
    super.flush();} }
```

Figure 9. jGuard annotated `ByteArrayInputStream`

RQ1-a: jGuard can accurately guard against most prevalent misuses happening in practice according to MuBench. Out of the total 139 misuses, it was possible to accurately express guards for 124 (89.2 %) misuses.

(RQ2) Comparison to CogniCrypt: To answer **RQ1-b**, we compare the recall and precision of jGuard and CogniCrypt on JCA.

Justification for CogniCrypt and JCA: CogniCrypt is a valuable API misuse detector to position jGuard against for multiple reasons. First, it uses CrySL as the backbone, which is a state of the art in API misuse specification languages (cf. Section 3). Second, it follows an allow-listing approach to static API misuse detection, which has been shown to be superior to deny-listing approaches [11]. Finally, in the domain of cryptographic misuses, CogniCrypt achieves a higher accuracy than other static analyzers [20], making it a good baseline for the evaluation. The rationale for performing our evaluation with the JCA crypto library shipped with JDK is as follows. First, CogniCrypt is also evaluated on JCA.

Second, it is representative for modern APIs that support mechanisms that complicate usage constraint specifications, such as plug-and-play providers among other things [8].

Experiment and results: Specifically, we created a jGuard-annotated version of BouncyCastle, an implementation of JCA [38]. As a source of misused applications to perform the comparison, we extracted a misuse dataset from MuBench containing all misuses related to `javax.crypto`-API. Overall, we found a total of 38 misuses, out of which, 8 were not further considered – three of them were reported on Dalvik (not original JDK), one was from a non-cryptographic class, and four point to unavailable sources. All remaining 31 misuses were categorized based on their description as reported in MuBench. While static analysis tools can run on source code (or, in case of CogniCrypt, on compiled Java bytecode), evaluating jGuard requires running the code to detect misuses. We replaced the standard JCA-provider with the verified BouncyCastle implementation and ran the misused applications from MuBench (now utilizing a verified API). We measure recall by comparing detected misuses (reported via standard error output) to the dataset extracted from MuBench. Given these 31 misuses to consider, we were able to express guards for 26 of them: Two misuses were related to missing exception handling which cannot be detected at runtime. Two additional misuses are formed by developers using insecure, hardcoded keys for secure operations which is also impossible to detect effectively. Finally, one described misuse was using an encrypting method as a form of decryption which is a high-level usage violation that can't be detected by guarding the JCA implementation. Being able to express 26 out of 31 misuses (83.9 %) roughly matches our findings from **RQ1-a**.

Some methods contained multiple distinct misuses, e.g., one used the DES encryption algorithm with the ECB chaining mode. MuBench treats them as separate misuses. A guarded library implementation throws an exception on the first misuse, thus the second misuse does not occur and is not reported. To work around this peculiarity, we isolated the separate misuses by creating variants of the application to contain only the first misuse (then only the second and so on). To quantify false positives, we manually analyzed the source code and the MuBench report and used our domain knowledge to determine whether a reported misuse hints towards a valid issue or not. Out of 31 cryptographic misuses, jGuard checks detected 26 (83.87 %) and CogniCrypt 25 (80.65 %). Notably, jGuard's false-negatives match the misuses for which we were unable to design runtime checks. In other words, the introduced checks did not miss a misuse they were designed to detect. jGuard did not report any false positives, while CogniCrypt reported 67 false positives, yielding a precision of 27.17 %. For an example of false positives, consider one related to the usage rule *"do not use a non-random IV for CBC encryption"* [13]. In the Cipher Block

Misuse	**jGuard** TPs, FPs	**CogniCrypt** TPs, FPs
Using DES	🟢 (9/9), 0	🟢 (9/9),15
Using ECB	🟢 (9/9), 0	🟢 (9/9), 23
Initializing Cipher multiple times	🟢 (3/3),0	🟠 (2/3), 10
Initializing MAC multiple times	🟢 (1/1), 0	🔴 (0/1), 2
Unsafe RSA configurations	🟢 (1/1), 0	🟢 (1/1), 4
Unsafe PBE	🟢 (1/1), 0	🟢 (1/1), 4
Non-random IV for CBC	🟢 (1/1), 0	🟢 (1/1), 1*
Zero-length array in `doFinal`	🟢 (1/1), 0	🔴 (0/1), 2
Missing exception handling	🔴 (0/2), 0	🔴 (0/2), 3
Using static keys	🔴 (0/2), 0	🟢 (2/2), 3
Encryption Cipher for decryption	🔴 (0/1), 0	🔴 (0/1), 0
Summary	26/31 , 0	25/31, 67

* TPs = True positives. FPs = False positives

Figure 10. Overview of running jGuard and CogniCrypt on cryptographic misuses in MuBench

Chaining mode (CBC), the common attack against ECB is adverted by using the output of a previous block when encrypting subsequent blocks. The initial block is combined with an *Initialization Vector* (IV) instead [25]. This mode is only secure when IVs for encryptions are random [6, 28]. The rules in CogniCrypt enforce random IVs for both encryption and decryption. For decryption, however, the IV must only match the one used for encryption, i.e., the receiver should not compute its own random IV. The example is representative of a broader class of false positives related to cases where values of some objects are dependent on the runtime values of other objects. Such checks are not easy to express using static analysis tools. CogniCrypt also reported a lot of unrelated warnings that do not represent any misuse. As an example, consider the implementation of the `Cipher.getCipher(int)` method where CogniCrypt reports that calls to `update`, `wrap`, or `doFinal` are expected, even though this method is concerned only with the creation of a `Cipher` object. For those warnings, we checked if they might indicate *another* valid misuse. If we determined that the warning was invalid, we considered it to be a false positive. An overview of the comparison with CogniCrypt is provided in Figure 10.

RQ1-b: With regard to misuses of cryptographic libraries reported in MuBench, jGuard's recall (26/31) is comparable to that of CogniCrypt(25/31), but its precision is much higher – no false-positives were reported by jGuard while CogniCrypt reported as many as 67.

(RQ2) Performance impact: jGuard generates dynamic checks for annotated classes and inserts additional code to perform these checks into the generated files. Hence, to understand how feasible it is to use jGuard-annotated libraries in practice, we need to measure a potential runtime impact. To answer **RQ2**, we create a hybrid collection of 20 benchmarks (11 real-world, 9 synthetic but common cryptographic tasks). For benchmarking, we use JMH (the Java Microbenchmark Harness) [10]. Each benchmark has 10 warmup iterations of 2 seconds each and 6 execution iterations of 10

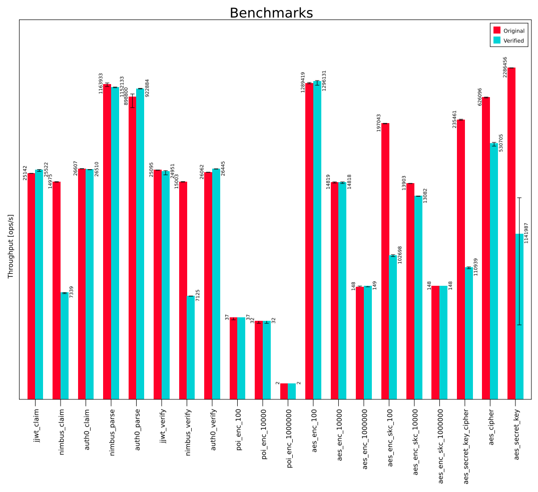


Figure 11. Benchmark results

seconds each. The benchmarks are performed with OpenJDK 17.0.2 on an Intel i5-1135G7, 32GB memory running on Manjaro with kernel version 5.16.14-1. The real-world benchmarks were selected from Maven [9] based on the popularity and feasibility of writing benchmarks: We chose multiple **JWT** libraries including JJWT, Nimbus-JOSE-JWT and Java-JWT by auth0 and **Apache PO** , a library for reading and writing Microsoft Office binary and `ooxml` files. For Apache POI, we use our own benchmark. An in-memory xlsx file is encrypted and written to memory. It is performed each with a file containing 100, 10 000 or 1 000 000 rows, where each of the row's first cell contains a constant string. We also curated several synthetic benchmarks simulating popular cryptographic tasks offered using BouncyCastle. These are **AES cipher creation, AES secret key creation, combined AES secret key and cipher creation, AES encryption of an array, AES secret key and cipher creation** and **encryption of an array**. We measure the throughput (op/s). The AES encryption benchmarks are performed with an array of length 100, 10 000 and 1 000 000 bytes each. The results of our benchmarks are summarized in Figure 11. Apart from **nimbus_verify** and **nimbus_claim** where the verified implementations are about twice as slow than the non-verified(original) counterparts, all the other real world benchmarks show similar performance for both the original and the verified implementations. Infact, the **Apache POI** benchmarks show no real difference between the normal and verified version at all Our synthetic benchmarks indicate that for pure encryption, there is no difference in performance. Creation of ciphers and secret keys are slower (verified aes_cipher and verified aes_secret_key are 15.2% and 50% slower than their non-verified counterparts). But, performance difference for encryption with creation of a secret key and cipher (aes_enc_skc) depends on the payload size (the higher the payload size, the smaller the performance impact).

RQ2: In almost all of the realworld benchmarks, jGuard introduces little to no performance overhead. Where there has been a considerable impact, these operations do not seem to occur often, especially when there is a large payload size or other tasks (e.g., Apache POI). Even for scenarios where the performance overhead is high, API designers can request jGuard to generate jar files with checks disabled for deployment after using the checks enabled for testing.

5 Threats to Validity

jGuard-generated checks are a form of dynamic analysis, requiring users to execute code to find misuses. To confidently claim the absence of API misuse, an application developer would need to write tests thoroughly covering interactions with jGuard-annotated libraries. Still, our analysis of MuBench misuses in real-world projects found that most misuses occur unconditionally, hinting that they may be detected by simple tests or during development. Due to informal descriptions of misuse categories in existing work [4, 23], a formally sound argument on jGuard's capabilities in guarding against them was not possible. The informal arguments might miss instances of misuses that jGuard is unable to describe. However, the practical evaluation did indicate that jGuard can express guards against most misuse categories reported in MuBench. Further, a comparison against CrySL shows that jGuard has an expressiveness exceeding current API specification languages. We used our domain expertise to validate if certain misuse reports are false positives which could involve a potential bias. We believe this bias is mitigated greatly because our definition of false positives came out of a thorough read of the MuBench misuse descriptions. jGuard only partially supports misuses caused by forgotten `try`-statements in code invoking methods that may potentially throw an exception. Some guard against this misuse are included in the Java programming language itself, which forces callers to handle a range of possible exceptions with an explicit `throws` clause in the method declaration or a surrounding `try/catch` mechanism. Apart from that, it is not possible to detect the presence of a try statement at runtime without modifications to the JVM.The limits of guards in the presence of inheritance/ polymorphism is not known. Similarly, no special consideration was put into thread-safety when changing guards. Although guards are compiled into regular fields, it is possible that the current implementation might require engineering adaptations to fully support inherited classes or synchronized modifications. That being said, none of the misuses encountered during our experiments with RQ1-a or RQ1-b required such extensive support for

[9] https://mvnrepository.com/artifact/org.bouncycastle/bcprov-jdk15on/usages

inherited guards or language constructs guarding against concurrency misuse. The difficulty of writing jGuard specifications is not understood clearly yet. But, we have provided a 1-1 mapping from CrySL to jGuard in Section 3 which can be used as a compiler for API designers used to writing specifications in CrySL which has shown to be easily usable. We have laid the first steps in this direction by implementing a version of CrySL within the MPS ecosystem making this conversion feasible [10]. Further, we did not encounter any examples where the jGuard annotations contained much more lines of specifications than a corresponding CrySL specifications.

6 Related Work

Specialized concepts present in some programming languages can be used to harden APIs against misuse without requiring further tools. Kotlin is a JVM language compatible to Java and most well-known for *null safety* [1], meaning that misuses related to the handling of `null` values are prevented by the language. *Refinement types* are used to express semantic properties that can efficiently be checked statically [35]. For instance, a type `{x: Int | x > 127}` would express all integers greater than 127. Trivially, this mechanism can be used to express usage constraints on arguments by introducing matching types. Additionally, *Implicit Refinement Types* can be used to efficiently express constraints on the state of the program [34]. This allows formalizing misuses related to method sequences. Bengtson et al. introduced refinement types for an F# typechecker to express cryptographic properties of implementations [7] Aspect-oriented programming is a programming technique aiming to improve the composability of software [14]. While aspect-oriented programming is not generally used as a toolkit to guard APIs against misuse, it may be used to derive constructs similar to our language-level work (see e.g.[5]). However, aspect-oriented programming is a fundamental approach to building software, whereas jGuard can be used to incrementally annotate an implementation without substantially changing its structure. With *design by contract*, individual modules or methods specify their requirements and behavior through formal contracts [27]. The Eiffel programming language uses this approach to increase the reliability of software [26]. This is different to jGuard, which primarily focuses on API misuse, does not require changes to client code and compiles to Java. A lot of existing work has focused on using static analysis to find API misuses. Simple analysis tools may work on a known set of rules. *CryptoLint* is a static-analysis tool finding common violations of cryptographic principles in Android applications [13]. CogniCrypt ([19]) is an eclipse project aiding developers in the correct use of cryptographic APIs. Instead of using hard-coded rules, it finds misuses based on configurable rules. The *CrySL* specification language is introduced to express allowed usage patterns. As the evaluation has shown, jGuard is more expressive than CrySL. BlockHound is a tool to detect misuses related to calling blocking methods in a context where this harms the application's reactiveness [12]. It works by installing a custom class loader intercepting invocations to forbidden APIs, which is a form of dynamic analysis that does not require implementations to be changed. Li et al. introduced *residual investigation*, a technique combining static and dynamic analysis to improve accuracy [22]. Based on potential findings reported by FindBugs, a generated dynamic check verifies whether the static analyzer was correct. Java MOP is a MOP implementation for Java code [9]. The overall goal of MOP is to increase an application's reliability, whereas jGuard puts a focus on API misuse specifically. Further, jGuard does not require end users to install additional tooling.

[10]https://github.com/CROSSINGTUD/Crysl-MPS

7 Conclusion

API misuses can be potentially expensive mistakes. Several approaches have been proposed to detect such misuses statically and dynamically. In this paper, we presented JGuard, a Java extension that can produce defensive APIs that are misuse-resistant by design. We designed our language to be able to express the common misuse categories in the wild. We evaluated JGuard in comparison with a state of the art, API misuse specification language CrySL and found it to match the capabilities in finding misuses. In addition, dynamic checks allowed more accurate analysis without the high degree of false-positives reported by CogniCrypt. JGuard was also able to exceed the expressiveness of critical cryptographic violations listed by CryptoLint. We also evaluated the expressivity of JGuard with regards to the most frequently occurring misuse categories in the wild. A formal comparison against CrySL and misuse categories revealed that jGuard exceeds the expressiveness of CogniCrypt, especially outside of the domain of cryptographic misuses.

Acknowledgements

This work has been co-funded by the Crossing SFB 119 and through the support of the National Research Center for Applied Cyberse- curity ATHENE.

References

[1] Marat Akhin and Mikhail Belyaev. 2021. *Kotlin language specification.* https://kotlinlang.org/spec/pdf/kotlin-spec.pdf

[2] Alvin Alexander. 2019. Java JDBC connection string examples. https://alvinalexander.com/java/jdbc-connection-string-mysql-postgresql-sqlserver/.

[3] Sven Amann, Sarah Nadi, Hoan Anh Nguyen, Tien N. Nguyen, and Mira Mezini. 2016. MUBench: A Benchmark for API-Misuse Detectors.

In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR 2016)*. https://doi.org/10.1145/2901739.2903506

[4] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Transactions on Software Engineering* 45, 12 (2019), 1170–1188. https://doi.org/10.1109/TSE.2018.2827384

[5] Pavel Avgustinov, Eric Bodden, Elnar Hajiyev, Laurie J. Hendren, Ondrej Lhoták, Oege de Moor, Neil Ongkingco, Damien Sereni, Ganesh Sittampalam, Julian Tibble, and Mathieu Verbaere. 2006. Aspects for Trace Monitoring. In *Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4262)*, Klaus Havelund, Manuel Núñez, Grigore Rosu, and Burkhart Wolff (Eds.). Springer, 20–39. https://doi.org/10.1007/11940197_2

[6] Mihir Bellare. 2018. *Symmetric Encryption.* http://cseweb.ucsd.edu/~mihir/cse107/slides/s-se.pdf

[7] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement Types for Secure Implementations. *ACM Trans. Program. Lang. Syst.* 33, 2 (2011), 8:1–8:45. https://doi.org/10.1145/1890028.1890031

[8] Rodrigo Bonifacio, Stefan Krüger, Krishna Narasimhan, Eric Bodden, and Mira Mezini. 2021. Dealing with Variability in API Misuse Specification. arXiv:2105.04950 [cs.CR]

[9] Feng Chen and Grigore Roşu. 2005. Java-MOP: A Monitoring Oriented Programming Environment for Java. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nicolas Halbwachs and Lenore D. Zuck (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 546–550.

[10] Diego Costa, Cor-Paul Bezemer, Philipp Leitner, and Artur Andrzejak. 2019. What's wrong with my benchmark results? studying bad practices in JMH benchmarks. *IEEE Transactions on Software Engineering* 47, 7 (2019), 1452–1467.

[11] Uri Dekel and James D Herbsleb. 2009. Improving API documentation usability with knowledge pushing. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 320–330.

[12] Felix Dobslaw, Morgan Vallin, and Robin Sundström. 2020. Free the Bugs: Disclosing Blocking Violations in Reactive Programming. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 177–186. https://doi.org/10.1109/SCAM51674.2020.00025

[13] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. ACM, 73–84. https://doi.org/10.1145/2508859.2516693

[14] Tzilla Elrad, Robert E. Filman, and Atef Bader. 2001. Aspect-Oriented Programming: Introduction. *Commun. ACM* 44, 10 (Oct. 2001), 29–32. https://doi.org/10.1145/383845.383853

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

[16] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[17] Douglas Kramer. 1999. API documentation from source code comments: a case study of Javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*. 147–153.

[18] Stefan Krüger. 2022. The CrySL Language, CogniCrypt. https://www.eclipse.org/cognicrypt/documentation/crysl/.

[19] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. 2017. CogniCrypt: supporting developers in using cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 931–936. https://doi.org/10.1109/ASE.2017.8115707

[20] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2018. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands (LIPIcs, Vol. 109)*, Todd D. Millstein (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:27. https://doi.org/10.4230/LIPIcs.ECOOP.2018.10

[21] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 2006. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes* 31, 3 (May 2006), 1–38. https://doi.org/10.1145/1127878.1127884

[22] Kaituo Li, Christoph Reichenbach, Christoph Csallner, and Yannis Smaragdakis. 2014. Residual Investigation: Predictive and Precise Bug Detection. *ACM Trans. Softw. Eng. Methodol.* 24, 2, Article 7 (Dec. 2014), 32 pages. https://doi.org/10.1145/2656201

[23] Xia Li. 2020. *An Integrated Approach for Automated Software Debugging via Machine Learning and Big Code Mining.* The University of Texas at Dallas.

[24] Bill Marczak and John Scott-Railton. 2020. *Move Fast and Roll Your Own Crypto: A Quick Look at the Confidentiality of Zoom Meetings.* Citizen Lab. https://citizenlab.ca/2020/04/move-fast-roll-your-own-crypto-a-quick-look-at-the-confidentiality-of-zoom-meetings/

[25] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. 2018. *Handbook of Applied Cryptography.* CRC Press. https://books.google.de/books?id=YyCyDwAAQBAJ

[26] Bertrand Meyer. 1988. Eiffel: A language and environment for software engineering. *Journal of Systems and Software* 8, 3 (1988), 199–246. https://doi.org/10.1016/0164-1212(88)90022-2

[27] Bertrand Meyer. 2002. *Design by contract.* Prentice Hall Upper Saddle River.

[28] Bodo Möller. 2004. *Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures.* https://www.openssl.org/~bodo/tls-cbc.txt

[29] Dong Qiu, Bixin Li, and Hareton Leung. 2016. Understanding the API usage in Java. *Information and Software Technology* 73 (2016), 81–100. https://doi.org/10.1016/j.infsof.2016.01.011

[30] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. 2020. API-Misuse Detection Driven by Fine-Grained API-Constraint Knowledge Graph. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 461–472.

[31] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2012. Automated API property inference techniques. *IEEE Transactions on Software Engineering* 39, 5 (2012), 613–637.

[32] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. 2013. Automated API Property Inference Techniques. *IEEE Transactions on Software Engineering* 39, 5 (2013), 613–637. https://doi.org/10.1109/TSE.2012.63

[33] Martin P Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6 (2011), 703–732.

[34] Anish Tondwalkar, Matthew Kolosick, and Ranjit Jhala. 2021. Refinements of Futures Past: Higher-Order Specification with Implicit Refinement Types. In *35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 18:1–18:29. https://doi.org/10.4230/LIPIcs.ECOOP.2021.18

[35] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 209–228.

[36] Markus Voelter. 2011. Language and IDE Modularization and Composition with MPS. In *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers (Lecture Notes in Computer Science, Vol. 7680)*, Ralf Lämmel, João Saraiva, and Joost Visser (Eds.). Springer, 383–430. https://doi.org/10.1007/978-3-642-35992-7_11

[37] Markus Voelter. 2013. *Language and IDE Modularization and Composition with MPS*. Springer Berlin Heidelberg, Berlin, Heidelberg, 383–430. https://doi.org/10.1007/978-3-642-35992-7_11

[38] Chamila Wijayarathna and Nalin A. G. Arachchilage. 2018. Why Johnny Can't Store Passwords Securely? A Usability Evaluation of Bouncycastle Password Hashing. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018* (Christchurch, New Zealand) *(EASE'18)*. Association for Computing Machinery, New York, NY, USA, 205–210. https://doi.org/10.1145/3210459.3210483

[39] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are Code Examples on an Online Q&A Forum Reliable?: A Study of API Misuse on Stack Overflow. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 886–896. https://doi.org/10.1145/3180155.3180260

NerdBug: Automated Bug Detection in Neural Networks

Foad Jafarinejad
Department of Computer Science, Technical University of Darmstadt
Darmstadt, Hessen, Germany
foad.jafarinejad@stud.tu-darmstadt.de

Krishna Narasimhan
Department of Computer Science, Technical University of Darmstadt
Darmstadt, Hessen, Germany
kri.nara@st.informatik.tu-darmstadt.de

Mira Mezini
Department of Computer Science, Technical University of Darmstadt
Darmstadt, Hessen, Germany
mezini@st.informatik.tu-darmstadt.de

ABSTRACT

Despite the exponential growth of deep learning software during the last decade, there is a lack of tools to test and debug issues in deep learning programs. Current static analysis tools do not address challenges specific to deep learning as observed by past research on bugs specific to this area. Existing deep learning bug detection tools focus on specific issues like shape mismatches. In this paper, we present a vision for an abstraction-based approach to detect deep learning bugs and the plan to evaluate our approach. The motivation behind the abstraction-based approach is to be able to build an intermediate version of the neural network that can be analyzed in development time to provide live feedback programmers are used to with other kind of bugs.

CCS CONCEPTS

• **Software and its engineering** → **Software defect analysis**; **Maintaining software**; **Model-driven software engineering**; **Correctness**; **Context specific languages**; **System modeling languages**; **Software testing and debugging**; *Scripting languages*; *Software development techniques*.

KEYWORDS

Bug Detection, Debugging, Machine Learning, Deep Learning

ACM Reference Format:

Foad Jafarinejad, Krishna Narasimhan, and Mira Mezini. 2021. NerdBug: Automated Bug Detection in Neural Networks. In *Proceedings of the 1st ACM International Workshop on AI and Software Testing/Analysis (AISTA '21), July 12, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3464968.3468409

1 INTRODUCTION

Bugs in software contribute to trillions of dollars in organizational costs, and past studies have shown that up to 150 of every 1000 line of software is error-prone [1]. To mitigate this issue, static analysis tools like FindBugs [3] and SpotBugs [18] have been proposed that aid in detecting bugs. These tools have a pre-defined set of popular bug patterns they detect and primarily operate on Java. Today a lot of software, especially recommendation systems like speech-to-text services and suggestions for purchases, use deep learning. The deep learning market is estimated to grow by 47% between 2018 and 2023 [2].

Deep learning programs - most of which are written in Python - also suffer from bugs; thus, rendering the existing Java-based tools useless. Tools like Pychecker [12] detect bugs in Python. However, the existing bug patterns do not cover the novel bugs that are produced by deep learning code and the unconventional programming paradigms involved in building deep learning software. Amimla [7] is an approach that explores the direction of detecting API misuses in deep learning programs but fails to detect other kinds of bugs like dimension mismatches. There are frameworks [9, 14] that use deep learning to detect bugs; however, they do not apply to bugs in deep learning programs. Approaches like CRADLE [13] apply to finding and localizing bugs in deep learning software libraries like TensorFlow and Theano. CRADLE detects bugs in the backend by detecting inconsist behavior of the same algorithm across different implementations: like the different padding of image sizes for the same scheme. CRADLE does not detect bugs introduced by improper use of these libraries. Pythia [10] is an approach that is closest to accomplishing this, in that it is a static analysis framework that detects shape mismatches in deep learning programs. Although this is a step in the right direction, as shown by previous studies bugs in deep learning programs encompass a wide variety of categories than just shape mismatches.

In this paper, we present a vision for a novel approach to detect bugs in programs that perform deep learning. One of the challenges of detecting deep learning bugs is that the issues are not evident from the code but rather only evident in the underlying neural network that needs to be built. Building networks from code takes time and therefore delays the live feedback developers are used to when detecting normal bugs using static analysis tools and IDEs. Since we build our abstraction with bare minimum information required to detect each bug, we are confident this abstraction can be built much quicker than the complete neural network graph. Neural network visualization approaches [3] have been proposed to aid developers to gain a better understanding of the underlying networks, but they do not locate or isolate potential problems. We propose "NerdBug," an approach that relies on the idea of an abstraction layer that serves as a toned-down intermediate representation for detecting bugs in deep learning programs. Our approach's core is to use classical program analysis tools, e.g., static program analysis

[1] https://www.softwaretestingnews.co.uk/the-real-cost-of-software-bugs/

[2] https://www.marketsandmarkets.com/Market-Reports/deep-learning-market-107369271.html

[3] https://playground.tensorflow.org/

[19], to identify the minimal information from the code required to detect deep learning bugs. Afterward, we build an abstraction based on the collected information, and analyze the abstraction to detect and isolate bugs. Also, our approach is capable of handling complex architectures by using interprocedural analysis [16] and function inlining. The rest of the paper is organized as follows. In Section 2, we review some preliminaries about deep learning and bugs in neural networks. In Section 3, we present the envisioned approach of NerdBug and provide examples to illustrate how in NerdBug we plan to build abstractions from code dedicated to isolating bugs. In Section 4, we describe how we plan to evaluate NerdBug, especially the dataset and the criteria we plan to use.

2 PRELIMINARIES

2.1 Neural Networks

Deep Neural Network (DNN)[15] is a computational model inspired by the structure of brain neural networks. In a simplified view, the brain consists of a set of computation units called inter-connected neurons. The Artificial Neural Network (ANN), a very broad term that encompasses any form of Deep Learning model (shallow and deep), is made of a set of artificial nodes that are connected to each other based on a complex network. Usually, for simplicity, we call ANNs Neural Networks as NN. An NN is a directed graph in which each node is a neuron and each edge is representing the connection between neurons. In NNs, each neuron will receive a set of inputs, which are a weighted sum of the output of neurons that are connected to its income edges.

2.2 Bugs in Neural Networks

A comprehensive study of bugs characteristic developers face during implementation of NNs was conducted previously [6]. The authors investigated and categorized bugs in software that makes use of deep learning libraries as below.[4]

i) API bug ii) Coding bug iii) Data bug iv) Structural Bug (SB) v) Non-Model Structural Bug (NMSB)

The SB has 5 subtypes 1) Control and sequence bug 2) Data flow bug 3) Initialization bug 4) Logic bug 5) Processing bug

This NMSB has 4 subtypes, which correspond to the subtypes of the SB bugs, except for the data flow bug category.

3 METHODOLOGY

At the core of our proposal is an abstraction layer that captures the bare minimum information required to detect each kind of bug, some sort of an intermediate representation of the NN designed with the specific debugging needs in mind. To guide the building of bug-specific abstractions, we have sub-categorized the bugs outlined in the previous section based on the programming context they occur in, whether in the context of control or data flow, or during initialization, processing etc. as follows:

(1) Class A [Data bug, Coding Bug, Structural Bug (control and data-flow, Non Model Structural Bug (control and data-flow)]
(2) Class B [Structural and Non Model Structural Bug - Initialization]
(3) Class C [Structural and Non Model Structural Bug - Processing]
(4) Class D [Structural and Non Model Structural bug - Logic]

[4]For detailed explanation of these categories, we refer the readers to the original work.

Based on Islam et al. [6] more than 39% of bugs belongs class A, which is the highest presence for any type of bugs, so in this work we only focus on class A. We will design individual abstractions for each of the remaining classes as future work after evaluating our approach with class A's bugs.

3.1 Abstraction Algorithm

The pseudocode of the algorithm that we envision to create our abstraction graph is presented in Algorithm 1. As a first step, the algorithm attempts to isolate the sections of code that alter deep learning model objects, and their dependencies using static analysis. Once the search space of the code is narrowed, our algorithm constructs a graph representing the underlying neural network. For each input data or NN layer, a node is added to the graph and populated with information extracted by code analyzes. The type `node` is inherited by `layer_node` and `data_node` types. In a `data_node` object, we only persist information like *type* and *#node*. However, in the `layer_node` we store information such as *type*, *#node*, *filter_size*, *#filter*, and *activation*. Following is a description of each of the attributes of the `node` class: I) **type:** represents the type of the node, whether it is an input data, or a specific kind of layer, e.g., input layer, dense layer, etc. II) **#nodes:** represents the number of nodes or the dimension of the input data as a tuple (number of data points, dimension of each data point). III) **#filter and filter_size:** this represents the number of filters and the size of the convolution filter. IV) **activation:** this represents the activation function corresponding to each layer.

Algorithm 1: Abstraction Algorithm

```
Input  :Python code
Output:A graph $\mathcal{G}$ which representing the abstraction of the input python code
1  $\mathcal{G}$ = an empty graph
2  node = {type : ∅, #node : ∅, filter_size : ∅, #filters : ∅, activation : ∅}
3  processed_code = interprocedural_inlining(input_code) // see text
4  nn_chain = get_nn_chain(processed_code) // see text
5  for vertex v in nn_chain do
6      if line of v represents input data then
7          node[type] = input_data
8          node[#node] = dimension of input data
9          Add node to $\mathcal{G}$
10         Clear node
11     else if line of v represents a layer then
12         node[type] = type of layer
13         node[#node] = number of neurons in layer
14         node[filter_size] = filter size in the line
15         node[#filters] = num. of filters in the line
16         node[activation] = activation of layer
17         Add node to $\mathcal{G}$
18         Clear node
19     end if
20 end for
21 return $\mathcal{G}$
```

Pre-processing program analysis. To simplify the construction of our abstraction, our algorithm performs static analysis on the input code, results of which are used for later steps.

Our manual analysis of deep learning code in the wild revealed, model definitions and initialization were delegated to wrapper functions distant from their use. Therefore, we will perform an interprocedural call-graph analysis and inline function calls that modify model objects. This step also helps in overcoming the architecture complexity of the code constructing the neural network.

In fairly large projects, code sections that deal with model objects and their dependencies may be minimal, and these are the sections of the code that are of interest to identify bugs in neural network. Our algorithm flags all definitions of model objects based on an extensible list of code patterns through which model objects can be initialized. One such pattern could be all calls to the function `keras.Sequential`. Then, we identify their definition-use chains (def-use chains). In python, objects can be re-assigned to other values at will. For example, a model object initialized with `keras.Sequential()` could later be re-defined to a numeric value, meaning the rest of the chain after this re-definition is no longer of interest to us. To mitigate this, our algorithm performs a reaching definition analysis to identify parts of the use-def chain that only relate to actual model initializations. As a future work, we also imagine tweaking existing type systems to incorporate these information to perform bug detection as a form of type checking. We envision that the program analysis our algorithm performs before building the abstraction will evolve.

We explain our algorithm 1 assuming the input is the example code presented in Figure 1[5], which builds a neural network to classify digits of MNIST dataset [11]. The code contains two lines numbered 18, 20, and 22 colored with blue and red, also distinguished by underline and strike out, respectively. The blue variant with underline is bug free, while the red one with strike out is buggy. In the first walk through the code, we will consider the bug-free variant of the code.

Algorithm 1 initializes an empty graph $\mathcal{G}$, and an empty data structure *node* in lines 1 and 2. Next, it applies inter-procedural call-graph tracing in this example, the output is the same as input. After that, all chains of the code will be extracted and the ones that have their roots in definitions of neural network models will be extracted and iterated through. Lines 6 to 10 of our algorithm describe how a node in our abstraction will be constructed for a source line that corresponds to an input data. The first node of chain corresponding to line 4 of the code represents an input data (**x_train** represents the training data, and **y_train** represents the training labels. **x_test** represents the test data, and **y_text** represents the test labels.). Since the training data consists of 60000 data points each a matrix of dimension 28×28, we initialize the node with the attribute **#node** set to $[60000, (28, 28)]$ and add it to the empty graph $\mathcal{G}$ in line 9.

Lines 11 - 18 of the Algorithm create nodes in $\mathcal{G}$, if the corresponding line in the source code represents a layer's creation. Consider a node in the chain that is corresponding to line 17 of the input code. This will result in a node representing an input layer and the **#node** attribute set to $(28, 28)$ in our graph $\mathcal{G}$ as shown in Figure 2. Next nodes in the chain are corresponding to lines 18 - 23 of the input code (blue variant with underline) represent conv2D, maxPooling2D, conv2D, maxPooling2D, flatten, and dense layers with 1, 1, 1, 1, 1, and 10 nodes, and activated with "relu," $\emptyset$, "relu," $\emptyset$, $\emptyset$, and "softmax," function, escorted by filter_size of $(3, 3)$, $(2, 2)$, $(3, 3)$, $(2, 2)$, $\emptyset$, and $\emptyset$, such that #filter is 32, $\emptyset$, 64, $\emptyset$, $\emptyset$, and $\emptyset$, respectively. Respective nodes are added to $\mathcal{G}$. The final result is shown in Figure 2 - the variant with the blue rectangle for the third, fifth, and seventh nodes.

[5]This is the solution of keras.io to classify MNIST digits.

```
1   from tensorflow import keras
2   from tensorflow.keras import layers
3
4   (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
5   # Pre−processing data  ...
13  y_train = keras.utils.to_categorical(y_train, 10)
14  y_test = keras.utils.to_categorical(y_test, 10)
15
16  model = keras.Sequential()
17  model.add(keras.Input(shape=(28, 28,  1)))
18  model.add(layers.Conv2D(32, kernel_size=(3, 3), activation="relu"))
18  ~~model.add(layers.Conv2D(32, kernel_size=(3, 3), activation="rlu"))~~
19  model.add(layers.MaxPooling2D(pool_size=(2, 2)))
20  model.add(layers.Conv2D(64, kernel_size=(3, 3), activation="relu"))
20  ~~model.add(layers.Conv2D(64, kernel_size=(3, ), activation="relu"))~~
21  model.add(layers.MaxPooling2D(pool_size=(2, 2)))
22  model.add(layers.Flatten())
22  ~~# An empty line~~
23  model.add(layers.Dense(10, activation="softmax"))
24  # Compile, train , and evaluate the model
```

Figure 1: Implementation of simple convolutional neural network to classify digits of MNIST dataset keras.io website.

3.2 Detecting Bugs from Abstraction

After creating the abstraction based on Algorithm 1, we will look for anomalies in the abstracted graph. An anomaly is considered to have taken place when the output of a node is inconsistent with the input of the node to its right. An inconsistency could arise due to any number of issues like type or dimension mismatch. For example, in the blue-variant of Figure 2 output and type of each node is consistent with the input of its next nodes; therefore, there is no anomaly in the abstracted graph, which allows us to infer that there is no bug in the code.

To understand how our algorithm detects anomalies, consider another implementation that was manifested in Figure 1 with the bug represented as the red line and struck out. The red variant of line 18 sets the the activation function to *rlu* instead of *relu*. The abstracted graph of this is presented in Figure 2 with the red rectangle above the third node.

In the abstraction shown in Figure 2 with red rectangles, the second node specifies that the dimension of each input data 28×28. The third node specifies that 32 filters of 3×3 will be convoluted to the input data and the activated by *rlu* function. This function is neither defined in the code, nor in the deep learning libraries. This anomaly shows that there is a bug in this layer of the neural network. The forth node describes that 2-dimension polling operating with filter size 2×2 will be applied to its input. The fifth node indicates that 64 filters of $3\times$ will be convoluted to its inputs and activated by relu. This dimension is not valid. This problem indicates that there is a bug in this layer of the neural network. Since in the fist example and second example the activation function and kernel size was miscoded, respectively; these are code bugs. Even though, these are rare bugs, our algorithm can detect them. At this point, the developer will be presented with readable information of the graph and should take a look into the code of this layer and determine the correct size.

In the presented abstraction in Figure 2 with red rectangles, consider that the developer forgot to write a line of code that is corresponding to the seventh node (imagine the abstracted graph with out node number 7). In this case, the next node for the node

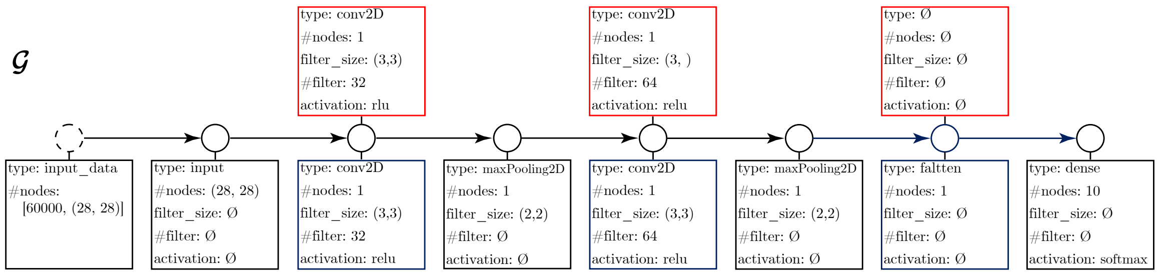


Figure 2: Abstracted graph for code shown in Figure 1

number 6 will be node number 8. Now, the sixth node performs a 2-dimension max pooling, and the eighth node presents a dense layer with 10 node. This connection is not feasible, because it is not possible to connect 2-dimension tensors to a dense layer. Thus, this is an anomaly and it indicates that there is a bug in the structure of the neural network. These examples indicate that our algorithm is capable of detecting different bugs in the code.

3.3 Implementation-POC

A prototype of *NerdBug* had been implemented in `Python` language. In this implementation we extended and utilized beniget[6], a static analysis library for python code. In order to test if our algorithm generalizes to developer-written code with large neural networks, we evaluated our prototype to detect a dimension mismatch bug in a code performing prediction task written by a researcher for their topology optimization project. The neural network in the project comprised 363881 parameters and 363865 trainable parameters.

4 EXPERIMENTAL EVALUATION

For preliminary experiment, *NerdBug* had been evaluated on a real world project which is written by using Keras and Tensorflow APIs. The result showed nerdBug could detect bugs during development time. We plan to experimentally evaluate *NerdBug* using the dataset described by Islam et al. [6], in which 2716 posts and 500 commits were gathered from Stack Overflow and GitHub respectively. Using these, a set of bugs for *Caffe* [8], *Keras* [4], *Tensorflow* [1], *Theano* [2], and *Torch* [5] libraries were constructed. We will evaluate NerdBug by detecting bugs in this dataset in terms of *false positive*, *false negative*, *precision*, *recall*, and F_1 score. Although our work is targeting a broader category of bugs, we also plan to compare our results with other works that detect API misuses in deep learning code [7] and bugs in machine learning algorithm implementations [17]. We plan to open-source NerdBug and any modifications to the dataset we perform.

ACKNOWLEDGMENTS

This work has been co-funded by the Crossing SFB 119 and through the support of the National Research Center for Applied Cybersecurity ATHENE.

[6]https://git.io/JGK5O

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.
[2] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv* (2016), arXiv–1605.
[3] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Softw.* 25, 5 (Sept. 2008), 22–29. https://doi.org/10.1109/MS.2008.130
[4] François Chollet et al. 2015. keras.
[5] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. 2002. *Torch: a modular machine learning software library.* Technical Report. Idiap.
[6] Md Johirul Islam. 2020. Towards understanding the challenges faced by machine learning software developers and enabling automated solutions. (2020).
[7] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 510–520.
[8] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. 675–678.
[9] Ashima Kukkar, Rajni Mohana, Yugal Kumar, Anand Nayyar, Muhammad Bilal, and Kyung-Sup Kwak. 2020. Duplicate Bug Report Detection and Classification System Based on Deep Learning Technique. *IEEE Access* 8 (2020), 200749–200763.
[10] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.15
[11] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/. (2010). http://yann.lecun.com/exdb/mnist/
[12] Neal Norwitz. [n.d.]. PyChecker. *SourceForge project http://pychecker. sourceforge. net* ([n. d.]).
[13] H. V. Pham, T. Lutellier, W. Qi, and L. Tan. 2019. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1027–1038. https://doi.org/10.1109/ICSE.2019.00107
[14] Michael Pradel and Koushik Sen. 2017. Deep learning to find bugs. *TU Darmstadt, Department of Computer Science* (2017).
[15] Shai Shalev-Shwartz and Shai Ben-David. 2014. *Understanding machine learning: From theory to algorithms.* Cambridge university press.
[16] Thomas C Spillman. 1971. Exposing Side-Effects in a PL/I Optimizing Compiler.. In *IFIP Congress (1)*. 376–381.
[17] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 271–280.
[18] David A. Tomassi. 2018. Bugs in the Wild: Examining the Effectiveness of Static Analyzers at Finding Real-World Bugs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 980–982. https://doi.org/10.1145/3236024.3275439
[19] Brian A Wichmann, AA Canning, DL Clutterbuck, LA Winsborrow, NJ Ward, and DWR Marsh. 1995. Industrial perspective on static analysis. *Software Engineering Journal* 10, 2 (1995), 69–75.

Impact of Programming Languages on Machine Learning Bugs

Sebastian Sztwiertnia[*]
Maximilian Grübel[*]
Amine Chouchane[*]
Technical University of Darmstadt
Germany
firstname.lastname@stud.tu-darmstadt.de

Daniel Sokolowski
Krishna Narasimhan
Mira Mezini
Technical University of Darmstadt
Germany
{sokolowski,kri.nara,mezini}@cs.tu-darmstadt.de

ABSTRACT

Machine learning (ML) is on the rise to be ubiquitous in modern software. Still, its use is challenging for software developers. So far, research has focused on the ML libraries to find and mitigate these challenges. However, there is initial evidence that programming languages also add to the challenges, identifiable in different distributions of bugs in ML programs. To fill this research gap, we propose the first empirical study on the impact of programming languages on bugs in ML programs. We plan to analyze software from GitHub and related discussions in GitHub issues and Stack Overflow for bug distributions in ML programs, aiming to identify correlations with the chosen programming language, its features and the application domain. This study's results enable better-targeted use of available programming language technology in ML programs, preventing bugs, reducing errors and speeding up development.

CCS CONCEPTS

• Computing methodologies → Machine learning; **• Software and its engineering → General programming languages**; **• General and reference → Empirical studies**.

KEYWORDS

machine learning, programming languages, empirical study

ACM Reference Format:
Sebastian Sztwiertnia, Maximilian Grübel, Amine Chouchane, Daniel Sokolowski, Krishna Narasimhan, and Mira Mezini. 2021. Impact of Programming Languages on Machine Learning Bugs. In *Proceedings of the 1st ACM International Workshop on AI and Software Testing/Analysis (AISTA '21), July 12, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3464968.3468408

1 INTRODUCTION AND BACKGROUND

The popularity and relevance of ML steadily increases. This ubiquity makes code quality and bug prevention in ML programs more relevant than ever. Bugs increase development costs tremendously and can lead to severe accidents. Between 40% to 80% of the project cost can be attributed to software testing [7]. Fixing ML bugs takes a lot of time; *timing and optimization* bugs on average 72 days and *algorithm and method* bugs—most common in ML—require on average 92 days [29]. Unidentified bugs may lead to reduced model performance [13], and they can cause real-world accidents, e.g., the Soyuz TMA-1 spaceship missed its supposed landing spot [23]. Also, 60% of the bugs analyzed by Islam et al. [13] lead to an application crash, enforcing the importance of preventing them. ML also becomes popular in safety-critical applications, e.g., autonomous driving [31], making its correctness more relevant than ever.

Existing research on bugs in ML programs focuses on ML libraries and investigates the developers' usage and problems [13, 28, 29, 32]. However, not only libraries but also the used programming language can impact code quality and encountered bug characteristics. Islam et al. [13] provide first evidence by analyzed bug distributions in ML programs. They found similar bug characteristics across the investigated ML libraries, hinting that there are common problems that might be caused by the language. Yet, so far, there is no focused investigation of the impact of the programming language nor its features on bugs in ML programs.

Today Python is central to ML development, especially for model development and integration. According to a survey on the popular ML platform Kaggle [15], Python is the most often used and the most recommended language to aspiring data scientists; presumably due to its perceived user-friendliness. However, Python lacks in speed compared to other languages, has higher memory consumption, and is more error-prone due to its dynamic typing, which can be frustrating [21]. It may well be that a considerable number of bugs are due to the prevalence of Python. But, we simply do not know.

For general purpose software on GitHub repositories, studies investigated the impact of programming languages on code quality [3, 24]. While these studies come to different findings regarding a programming language's impact, they underline the complexity of this question, where already finding suitable metrics is challenging [10]. Further, ML programs are different, making it hard to apply engineering know-how from general software programs or utilizing standard static analyzers to find bugs. The ML development process is data-driven in contrast to traditional code-driven software development [1] and requires closer collaboration between separate professions, including domain experts, data scientists, software engineers, operators and more [19].

In this paper, we aim to fill this gap by designing and conducting a systematic analysis of the effect of the chosen different programming language on bugs in ML programs. Gaining insight into whether and how programming languages influence bugs in ML programs will steer language development towards preventing ML bugs. Further, it can improve the language selection, which reduces the probability to face bugs, speeding up development and saving

[*]The first three authors contributed equally to this research.

cost. Also, the ML community can benefit as a whole by receiving tools and features better tailored to their needs.

Therefore, in this paper:

- We propose an empirical study on bugs in ML programs on GitHub and Stack Overflow (§2), correlating bug distributions with (1) the chosen programming language, (2) the application domain, and (3) the programming language's features.
- We present initial evidence for our research objectives (§3).
- We analyze the potential impact of our study, enabling better application of programming languages to ML programs (§4).

2 INVESTIGATING THE PROGRAMMING LANGUAGE IMPACT ON ML BUGS

To shed light on the impact of programming languages on bug characteristics in ML programs, we propose to study existing ML programs on GitHub and related problem discussions in GitHub issues and Stack Overflow questions and answers. We first ask:

RQ1: Do the bug characteristics depend on the chosen programming language? We assess whether the problems and peculiarities in the data can be attributed to the chosen programming language. We expect that the bug characteristics of ML programs can be grouped into significantly different clusters, which are mapped to their programming languages. Then we ask:

RQ2: Does the application domain influence the bug characteristics within a chosen programming language? To achieve guidance in choosing the right language, we aim to identify whether the application domain, e.g., computer vision or email filtering, correlates with the bug characteristics. In case, we provide evidence that future research on languages for ML must consider the application domain. Otherwise, likely, language improvements for one domain generalize well to other domains. To further detail, we ask:

RQ3: Are differences in the bug distribution explainable by the features of the chosen programming language? We assess bugs present in one language but unlikely in another and analyze correlations with the languages' features. This includes finding misuses of features and identifying the lacking features to prevent bugs. As Python is the most common ML language, we especially expect to find bugs that could be prevented by features available in other languages that are absent in Python.

2.1 Impact of the Language Choice

To answer RQ1, we first collect a sufficiently big dataset from Stack Overflow and GitHub, common sources for such studies [3, 11, 13, 24, 33]. Stack Overflow is a questions and answers platform for developers seeking help with programming problems. Its tagging system assigns a programming language to each question, making it easy to extract data. GitHub is a hosting platform for code repositories. Like Stack Overflow, GitHub has a tagging system for programming languages and a public API to download the data.

First, we download all GitHub repositories by using the same methodology as Gonzalez et al. [11] as it resulted in a sufficient amount of repositories (4.524) and excludes irrelevant repositories such as tutorials, homework assignments, etc. Stack Overflow questions and answers related to ML projects are gathered using the methodology of Islam et al. [13]. In this study, we focus on Python, C++, JavaScript, Java, R and Go as these are the main programming languages represented in the data of Gonzalez et al. [11]. We are aware of programming languages offering ML capabilities inherently, e.g., Julia and Swift. Due to these special capabilities, we suspect an impact on bug distributions that is too unique to compare to the prior mentioned general-purpose languages. Therefore, we do not include Swift and Julia in our proposed study.

In the next step, we will apply the pre-processing procedure proposed by Islam et al. [13], filtering and labeling the data. We reduce the data to bug-related content by selecting only the GitHub commits whose message includes "fix". For GitHub issues and Stack Overflow questions and answers, we will filter by keywords, too, extracting all posts containing "bug", "error" or "fail". For our study, the bug type, bug effect, root cause and programming language are the necessary attributes that we will extract. To categorize bugs, we will use the taxonomy of Beizer [2], also used by Islam et al. [13].

In the analysis phase, we identify the distributions of bug characteristics for each programming language. To compare the discrete distributions pairwise, we will use the Kolmogorov-Smirnov test (KS test) [18]. It prevents the unjustified assumption of normal distribution and comparing means, in contrast to the t-test statistic used by Islam et al. [13]. We chose the significance level $\alpha = 5\%$.

For instance, for the characteristic *bug type*, we will find the language pairs that have *significantly* different occurrence of these bugs. Finding these differences would support our hypothesis that the choice of the programming language impacts the correctness of ML programs. Also, the findings can indicate which languages are better suited for the development of ML-related applications as their choice is likely to reduce the number of bugs.

2.2 Impact of the Application Domain

To answer RQ2, we reuse the dataset collected in §2.1. However, for the analysis of the application domain's impact, we add the application domain attribute by tagging the data accordingly. Based on Shinde and Shah [27] we will categorize into the five domains *Computer Vision*, *Prediction*, *Semantic Analysis*, *Natural Language Processing* and *Information Retrieval*.

The analysis will follow the same procedure as in §2.1, however, we now test the impact of the application domain on the observed bug characteristics distribution instead of the programming language. We apply the KS test to each pair of domains across all programming languages and within each language. This obtains the domain pairs with *significantly* different bug characteristics across all languages and separately for each programming language.

If we observe that bug distributions are significantly different between domains across languages, we show that programming language research for ML must take the application domains into account. If the insights differ for the programming languages, we find the languages that are least and most error-prone for each investigated domain. This would illustrate that an ML project's language selection should take the application domain into account and provides qualified guidance during this process.

2.3 Impact of Programming Language Features

To answer RQ3, we reuse the dataset from §2.1 and combine it with a dataset of programming language features available in each of the investigated languages. Based on [8], we propose using the taxonomy introduced by Jordan et al. [14] with the addition of the class of programming language. This taxonomy includes *type checking*, *state cell assignment*, *state cell deletion*, *high order types*, *single assignment*, *modularity unit*, *functions* and *interactive input/output*. These features are complemented, as described, by the differentiation of programming language classes: *scripting language*, *object-oriented language* and *functional language* based on Scott [25]. To distinguish between language classes is reasonable as it was found that certain language classes are less prone to error than others [24].

In the analysis, we reuse the method used in §2.1, but this time, we assess for each language feature separately the impact of its availability on the bug characteristics distribution. We will find the language features whose availability correlates with the bugs in ML programs. For these, we repeat the KS test for each bug category individually and apply further manual analysis, finding which bugs are prevented by the feature and explanations why.

With the knowledge gained through this study, a set of language features could be found that potentially minimizes bugs in ML programs. This can be used to guide language selection and to shape future language and ML library development, suggesting to support language features that are likely to prevent bugs.

3 INITIAL EVIDENCE

ML Programs Are Different. The development of ML-driven applications does not follow the traditional software development workflow and strongly focuses on data [1]. Islam et al. [13] show that data bugs are most common in ML programs, further supporting that ML faces new problems that do not exist in traditional software development. Additionally, current debugging practices do not support identifying bugs in ML code well [32], indicating differences in the bugs and their distribution between ML and non-ML programs. Humbatova et al. [12] analyzed projects using popular deep learning frameworks, derive a bug taxonomy and validate it with a user study. While they did not investigate the impact of the used programming language, they give evidence to the relevance and difference of ML bugs. Also, the development of ML programs is different. Traditionally, the software design is secured from unsolicited change, hindering ML data scientists in their exploration [26]. Data scientists today also rely on informal versioning, consisting of commenting out parts of the code to keep it for later reference [16]. This approach of exploratory programming leads to new code quality trade-offs [4]. These insights show that ML programs are different from programs discussed in software engineering so far and, thus, provide evidence that this ML-focused study is needed.

Programming Languages Impact Bug Characteristics. Various studies analyzed code quality based on the programming language choice for general programs by, e.g., relating the language choice to the programs' bug distribution. Ray et al. [24] found a correlation between the chosen language and software bug distributions. However, Berger et al. [3] could only partly reproduce these results. Programming languages also impact bugs in projects utilizing multiple languages. Kochhar et al. [17] showed that the degree of error-proneness changes when specific programming languages are added to a project. In particular, the error-proneness increased when adding C++, Objective-C, Java, TypeScript, Clojure or Scala. This indicates that the languages may have different bug characteristics, providing initial evidence to RQ1. However, detailed insight for ML-related programs is not available yet.

Application Domains Impact Bug Characteristics. Linares-Vásquez et al. [20] analyzed the code quality of Java projects for different application domains. They used object-oriented metrics [6] as a proxy for code quality and found a negative correlation between anti-patterns and object-oriented metrics. Linares-Vásquez et al. [20] found that anti-patterns varied across application domains. Further, Islam et al. [13] identified that bug patterns in ML programs are different compared to other software domains. This provides initial evidence to RQ2, showing for general programs that their bug characteristics depend on the application domain and that ML programs have different characteristics than non-ML ones. However, a detailed analysis for application domains within ML is missing yet.

Programming Language Features Prevent Bugs. Programming languages differ in their set of supported features. Some of these can reduce the amount of bugs as shown, e.g., for type systems [22]. In contrast, dynamically typed languages like Python detect these errors at run time, giving less guarantee that a program will work. E.g., Islam et al. [13] confirm by identifying that Python is prone to data bugs and that type and shape mismatches are common problems. Programming language research is focused on developing language features preventing the introduction of bugs. For instance, gradual type systems have been explored recently for dynamically typed languages, including Python [30] and JavaScript [5]. E.g., TypeScript is a superset of JavaScript with static type system features and Gao et al. [9] observed that it could have prevented 15% of bugs in the investigated code on GitHub. These efforts to prevent common errors by adding a feature to the language give initial evidence to RQ3. However, the relationship between present errors in ML programs and language features is yet to be discovered.

4 IMPACT ANALYSIS

The presented study potentially impacts all future ML programs and, due to ML's ubiquity, a huge share of future systems. The improved understanding of the languages' impact and flaws for ML programs can help to select better suiting ones for the faced problem as well as improving the languages themselves. Both potentially improve ML program correctness and eases development, which can speed up ML development and further propel its ubiquity.

In detail, the insights through RQ1, RQ2 and RQ3 can help already today to identify preferable languages for ML problems from a bug prevention perspective. RQ2 guides programming language researchers for ML languages on how central the application domain is for their contributions, i.e., how probable it is that their evaluated improvements generalize to other domains. Further, RQ2 and especially RQ3 are valuable sources to guide the development of programming languages and libraries for future ML programs. ML library authors can leverage the identified shortcomings and apply language-level solutions as their library APIs themselves usually can be seen as a language within the programming language (an embedded DSL). Even without altering their APIs, library authors can improve their documentation and training through the improved awareness of mishaps not prevented by the language.

Reducing the number of introduced bugs also reduces friction and frustration during the development of ML programs. Thus, based on RQ2 and RQ3, easier-to-learn languages may be selected for teaching. This helps the community to grow fast and shapes the shared knowledge and standards within the community.

5 CONCLUSION

In this paper, we propose an empirical study on programming languages and PL bugs based on data from Stack Overflow and GitHub. First, we assess the impact of the programming language choice on bug distributions in ML programs. Second, we investigate the effect application domains have on language-specific and general bug distributions in ML programs. Lastly, we examine the programming language features' impact. We outline the method and research plan and identify initial evidence. The results will impact practitioners and scientists, revealing the code quality effects of programming languages in ML, potentially guiding language choice and feature design, preventing bugs in future ML programs.

ACKNOWLEDGMENTS

This work has been co-funded by the German Research Foundation (SFB 1119), by the Hessian LOEWE initiative (Software-Factory 4.0), and by the German Federal Ministry of Education and Research (BMBF) and the Hessian Ministry of Higher Education, Research and the Arts (HMKW) within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

REFERENCES

[1] Saleema Amershi, Andrew Begel, Christian Bird, et al. 2019. Software Engineering for Machine Learning: A Case Study. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice* (Montreal, Quebec, Canada) *(ICSE-SEIP '19)*. IEEE Press, 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[2] Boris Beizer. 1984. *Software System Testing and Quality Assurance.* Van Nostrand Reinhold Co., USA.

[3] Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. 2019. On the Impact of Programming Languages on Code Quality: A Reproduction Study. *ACM Trans. Program. Lang. Syst.* 41, 4, Article 21 (Oct. 2019), 24 pages. https://doi.org/10.1145/3340571

[4] Mary Beth Kery and Brad A. Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 25–29. https://doi.org/10.1109/VLHCC.2017.8103446

[5] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. Springer-Verlag, Berlin, Heidelberg, 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

[6] S.R. Chidamber and C.F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493. https://doi.org/10.1109/32.295895

[7] S. Eldh, H. Hansson, S. Punnekkat, et al. 2006. A Framework for Comparing Efficiency, Effectiveness and Applicability of Software Testing Techniques. In *Testing: Academic Industrial Conference - Practice And Research Techniques (TAIC PART'06)*. 159–170. https://doi.org/10.1109/TAIC-PART.2006.1

[8] Onyeka Ezenwoye. 2018. What Language? - The Choice of an Introductory Programming Language. In *2018 IEEE Frontiers in Education Conference (FIE)*. 1–8. https://doi.org/10.1109/FIE.2018.8658592

[9] Zheng Gao, Christian Bird, and Earl T. Barr. 2017. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 758–769. https://doi.org/10.1109/ICSE.2017.75

[10] M. Garkavtsev, N. Lamonova, and A. Gostev. 2018. Chosing a Programming Language for a New Project from a Code Quality Perspective. In *2018 IEEE Second International Conference on Data Stream Mining Processing (DSMP)*. 75–78. https://doi.org/10.1109/DSMP.2018.8478454

[11] Danielle Gonzalez, Thomas Zimmermann, and Nachiappan Nagappan. 2020. The State of the ML-Universe: 10 Years of Artificial Intelligence & Machine Learning Software Development on GitHub. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 431–442. https://doi.org/10.1145/3379597.3387473

[12] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1110–1121. https://doi.org/10.1145/3377811.3380395

[13] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[14] Howell Jordan, Goetz Botterweck, John Noll, et al. 2015. A feature model of actor, agent, functional, object, and procedural programming languages. *Science of Computer Programming* 98 (2015), 120–139. https://doi.org/10.1016/j.scico.2014.02.009

[15] Kaggle. 2019. State of Data Science and Machine Learning 2019. https://www.kaggle.com/c/kaggle-survey-2019/data, last accessed on 2021-06-03.

[16] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 1265–1276. https://doi.org/10.1145/3025453.3025626

[17] P. S. Kochhar, D. Wijedasa, and D. Lo. 2016. A Large Scale Study of Multiple Programming Languages and Code Quality. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 563–573. https://doi.org/10.1109/SANER.2016.112

[18] Andrey Kolmogorov. 1933. Sulla determinazione empirica di una lgge di distribuzione. *Inst. Ital. Attuari, Giorn.* 4 (1933), 83–91.

[19] Grace A. Lewis, Stephany Bellomo, and Ipek Ozkaya. 2021. Characterizing and Detecting Mismatch in Machine-Learning-Enabled Systems. arXiv:2103.14101 https://arxiv.org/abs/2103.14101

[20] Mario Linares-Vásquez, Sam Klock, Collin McMillan, et al. 2014. Domain Matters: Bringing Further Evidence of the Relationships among Anti-Patterns, Application Domains, and Quality-Related Metrics in Java Mobile Apps. In *Proceedings of the 22nd International Conference on Program Comprehension* (Hyderabad, India) *(ICPC 2014)*. Association for Computing Machinery, New York, NY, USA, 232–243. https://doi.org/10.1145/2597008.2597144

[21] Abhinav Nagpal and Goldie Gabrani. 2019. Python for Data Analytics, Scientific and Technical Applications. In *2019 Amity International Conference on Artificial Intelligence (AICAI)*. 140–145. https://doi.org/10.1109/AICAI.2019.8701341

[22] S. Nanz and C. A. Furia. 2015. A Comparative Study of Programming Languages in Rosetta Code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 778–788. https://doi.org/10.1109/ICSE.2015.90

[23] D.L. Parnas and M. Lawford. 2003. The role of inspection in software quality assurance. *IEEE Transactions on Software Engineering* 29, 8 (2003), 674–676. https://doi.org/10.1109/TSE.2003.1223642

[24] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A Large Scale Study of Programming Languages and Code Quality in Github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 155–165. https://doi.org/10.1145/2635868.2635922

[25] Michael L. Scott. 2009. 1 - Introduction. In *Programming Language Pragmatics (Third Edition)* (third edition ed.), Michael L. Scott (Ed.). Morgan Kaufmann, Boston, 5–39. https://doi.org/10.1016/B978-0-12-374514-9.00010-0

[26] Beau Sheil. 1986. Datamation®: Power Tools for Programmers. In *Readings in Artificial Intelligence and Software Engineering*, Charles Rich and Richard C. Waters (Eds.). Morgan Kaufmann, 573–580. https://doi.org/10.1016/B978-0-934613-12-5.50048-3

[27] Pramila P. Shinde and Seema Shah. 2018. A Review of Machine Learning and Deep Learning Applications. In *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. 1–6. https://doi.org/10.1109/ICCUBEA.2018.8697857

[28] X. Sun, T. Zhou, G. Li, J. Hu, H. Yang, and B. Li. 2017. An Empirical Study on Real Bugs for Machine Learning Programs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 348–357. https://doi.org/10.1109/APSEC.2017.41

[29] F. Thung, S. Wang, D. Lo, and L. Jiang. 2012. An Empirical Study of Bugs in Machine Learning Systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. 271–280. https://doi.org/10.1109/ISSRE.2012.22

[30] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. *SIGPLAN Not.* 50, 2 (Oct. 2014), 45–56. https://doi.org/10.1145/2775052.2661101

[31] Jianxiong Xiao. 2017. Learning Affordance for Autonomous Driving. In *Proceedings of the 2nd ACM International Workshop on Smart, Autonomous, and Connected Vehicular Systems and Services* (Snowbird, Utah, USA). Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/3131944.3133941

[32] Ru Zhang, Wencong Xiao, Hongyu Zhang, et al. 2020. An Empirical Study on Program Failures of Deep Learning Jobs. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1159–1170. https://doi.org/10.1145/3377811.3380362

[33] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, et al. 2018. Are Code Examples on an Online Q A Forum Reliable?: A Study of API Misuse on Stack Overflow. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 886–896. https://doi.org/10.1145/3180155.3180260

Towards Code Generation from BDD Test Case Specifications: A Vision

Leon Chemnitz
Software Technology Group
Technische Universität Darmstadt
Darmstadt, Germany
leon.chemnitz@stud.tu-darmstadt.de

David Reichenbach
Software Technology Group
Technische Universität Darmstadt
Darmstadt, Germany
david.reichenbach@stud.tu-darmstadt.de

Hani Aldebes
Software Technology Group
Technische Universität Darmstadt
Darmstadt, Germany
hani.aldebes@stud.tu-darmstadt.de

Mariam Naveed
Software Technology Group
Technische Universität Darmstadt
Darmstadt, Germany
mariam.naveed@stud.tu-darmstadt.de

Krishna Narasimhan
Software Technology Group
Technische Universität Darmstadt
Darmstadt, Germany
kri.nara@st.informatik.tu-darmstadt.de

Mira Mezini
Software Technology Group
Technische Universität Darmstadt
Darmstadt, Germany
mezini@informatik.tu-darmstadt.de

***Abstract*—Automatic code generation has recently attracted large attention and is becoming more significant to the software development process. Solutions based on Machine Learning and Artificial Intelligence are being used to increase human and software efficiency in potent and innovative ways. In this paper, we aim to leverage these developments and introduce a novel approach to generating frontend component code for the popular Angular framework. We propose to do this using behavior-driven development test specifications as input to a transformer-based machine learning model; however, we do not provide any proof-of-concept solution in this work. Our approach aims to drastically reduce the development time needed for web applications while potentially increasing software quality and introducing new research ideas toward automatic code generation.**

***Index Terms*—machine learning, code generation, behavior driven development, software testing, transformer, artificial intelligence, software engineering, frontend**

I. Introduction

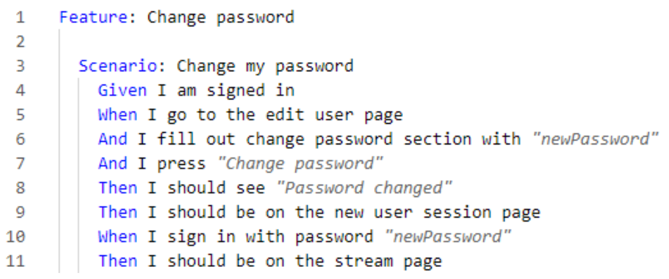
A rational software developer wants to make the most out of their time and work as efficiently as possible. This is true in virtually every profession since no professional likes to waste their time or that of the person or company that hires them. A developer that gets more work done in the same amount of time simply generates more value, which is in the best interest of all parties involved.
This inherent interest in optimizing work efficiency that permeates all fields of work has been one of the key motivations for many former and current research efforts that aim to assist developers in their day-to-day work and streamline the process of writing code. [1]–[7]
A major aspect of software development is dedicated to quality assurance (QA) and therein most notably automated software testing. Software testing plays a significant role in ensuring that the functional and non-functional requirements of a system are met. In other words, software testing makes sure a system behaves as expected. Automation and code generation in the area of software testing reduce the manual effort involved with QA which increases developer efficiency and enables systems to scale better. [8], [9]
There exist many methods and techniques in the realm of software testing, but in this work, we will focus on behavior-driven development (BDD) tests. BDD is a practice that integrates natural language (NL) into test specifications to make them comprehensible for people with little or no technical background.
Initially developed for the processing of NL, recent progress in Artificial Intelligence (AI) and Machine Learning (ML) enabled the application of novel techniques to coding assistance [3]–[6]. The transformer model has marked a milestone in natural language processing (NLP) [10] and is the technique we will be focusing on here but the ideas we present are not confined to any particular ML model.
In this work, we propose an approach that further leverages the NLP capabilities of transformers for code generation by using BDD test specifications as input for the generation task. We aim to provide developers with a tool to optimize their efficiency while also enforcing established software engineering practices i.e. automated software testing. While we explore and describe a possible implementation of this, we do not provide any proof-of-concept solution as part of this work.

The rest of the paper is structured as follows: Section II describes the research and technologies our work is based upon and our reasoning behind choosing them. In Section III we will describe our approach in detail. Section IV discusses the potential impact of our vision and Section V gives a conclusion.

II. Background

As stated in [11], reducing the manual effort involved with software testing is one of the key motivations driving current testing-related research. The 2020 Literature Review by Trudova et al. has shown that most approaches focus on the generation of test cases, while none attempt to generate

code from test specifications. This suggests that our described techniques are indeed novel. We propose a method to generate application code from test case specifications to further reduce the developer time needed in application development. Specifically, we aim to extract the relevant information needed for the generation of code from the specification of test cases adhering to behavior-driven development (BDD) standards.
The practice of BDD is a technique that evolved from test-driven development (TDD) [12]–[14]. It is an agile software development process that, at its core, focuses on the specification of test cases at the beginning of a development cycle together with domain experts and stakeholders. For the test cases to be interpretable by these domain experts and stakeholders, the test case specification is usually realized with a domain-specific language (DSL) that integrates natural language and is executed by a specialized tool or framework (e.g. cucumber [15]). An example of a BDD test case for the cucumber framework can be found in Figure 1.



```
Feature: Change password

  Scenario: Change my password
    Given I am signed in
    When I go to the edit user page
    And I fill out change password section with "newPassword"
    And I press "Change password"
    Then I should see "Password changed"
    Then I should be on the new user session page
    When I sign in with password "newPassword"
    Then I should be on the stream page
```

Fig. 1. Cucumber Test specifying a change password feature, based on real-world example [16].

In 2012 Soeken et al. [2] developed a technique to generate code stubs from these BDD test case specifications using state-of-the-art NLP techniques of the time. We propose a vision to take this idea further and generate functioning software components from these natural language specifications using state-of-the-art machine learning techniques like pre-trained transformer networks. We have strong reason to believe that this approach is viable since transformers have recently been shown to be highly effective in the generation of code from natural language [3] and the processing of code and natural language in general [4], [17], [18].
In addition to the reduction of effort required in the development of applications, we believe that our approach incentivizes the implementation of meaningful test cases and therefore the adoption of QA practices which are generally regarded as integral to the development of high-quality software.
Having a general-purpose solution that can reliably generate application code from BDD-test case specifications in an arbitrary context (application type, programming language, libraries, etc.) would be ideal but this defines a very complicated and nuanced research task which we do not aim to address here and leave it up to future work. Instead, we want to focus on the development of a system that works in a very defined context to reduce overall system complexity and research whether the general approach is feasible.
We propose to build a prototype that works in the context of frontend development. Virtually all applications developed in an enterprise scenario that are expected to be used by multiple concurrent users (except for mobile applications) are web applications. This means they are comprised of a backend running on one or more servers and a frontend running inside of the client's browser. While it is possible to build a frontend using plain HTML, JavaScript (JS), and CSS, this is very cumbersome and most of the time developers opt for the use of a Framework when developing modern frontends. Such frameworks ease the use of browser APIs and provide a way of structuring applications in a reliable way while providing much utility to the developer. For this research effort, we will be using the open-source Angular [19] framework developed by Google.

We base this decision on several reasons:

1) **Popularity:** Angular has been on the rapidly changing market of JS frameworks for over two decades and has consistently been among the top 2 most used frontend frameworks in the past 7 years. [20] This implies that there is presumably a large amount of publicly available, high-quality code in open-source repositories that can be used as training data for this project. Furthermore, this implies that reducing the development effort of angular applications would be relevant to many people and companies that rely on it today and most likely in the future.
2) **Opinionated Framework:** In contrast to other frameworks like React [21], Angular is a very opinionated framework. This means that there exist many conventions regarding Angular applications resulting in the fact that the overall structure of most applications and components is roughly the same. Having less variance in the code artifacts that are to be generated should make it easier for the applied ML model to perform well, even with sparse training data. We assume this because this way the model does not need to pick up on multiple ways and approaches a problem can be solved but rather learn what is dictated by convention.
3) **Enforced BDD:** As Angular is very opinionated, it also imposes conventions around the testing methodology on users. When creating a new Angular component using the Angular CLI (which is generally the preferred way of doing so), a test file for the new component is automatically generated. All Angular tests are based on Jasmine [22] which is a BDD framework for JavaScript and TypeScript. Because of this, we believe it is reasonable to assume that there exist many valuable samples of BDD specification and component code pairs in publicly available open-source projects that can be used for the training of our model. Also, because of this convention, we believe that the adoption of our proposed system should require minimal effort for a developer that is already accustomed to the Angular ecosystem.

To process the acquired data and use it for ML, we need an ML model that trains on the provided data so that it eventually becomes capable of generating the desired output given an input. We propose an approach that uses transformers [10]. Transformers are a relatively novel alternative to other deep neural network architectures such as convolutional (CNN) and recurrent neural networks (RNN) that promises benefits in terms of scalability and performance. Recent work has shown that transformers can be adopted for code generation and prediction with the results outperforming the state-of-the-art. [3]–[5], [17]

III. Method

As our proposed system will be built using a state-of-the-art ML model, we require sample data for training, testing, and validation. Details about how we plan to acquire and pre-process this data can be found in Section III-A. Details about the ML model can be found in Section III-B. The intricacies of our plans for the code generation process can be found in Section III-C. With our proposed approach, we plan to answer the following research questions:

- **RQ1**: What is the best way to curate data to train a model on angular test cases?
- **RQ2**: Can our proposed approach generate high-quality code?
- **RQ3**: How high is the instruction/branch coverage of generated application code?

The proposed system will be evaluated to find answers to the aforementioned research questions which is further elaborated in Section III-D.

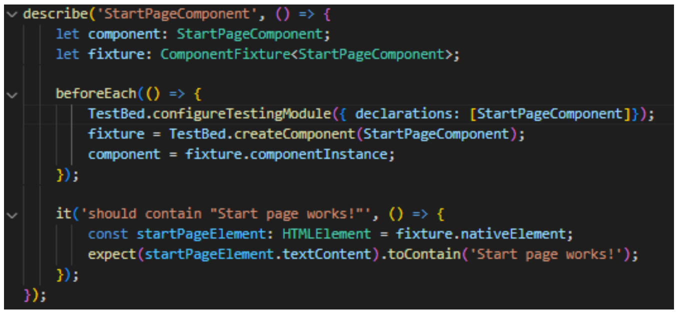
A. Data Aquisition and Pre-Processing

As stated at the beginning of this section, we need data for the training, testing, and validation of our model and we plan to extract this data from open-source projects on GitHub.

There are several considerations to be made during the acquisition of data in this specific context. First of all, not all parts of an angular application are created equal. In general, code containing files in an Angular project can be of different types. This includes the non-exhaustive list of *Components*, *Services*, and *Modules*. Explaining the function of each of these file types is beyond the scope of this paper. Here we provide a brief explanation. *Components* are where usually most of the application code resides. A *Component* describes a piece of user interface along with its markup, styling, and logic that can be arranged hierarchically. *Services* are singleton objects that make data and functionality available inside of a *Module*. *Modules* (as the name suggests) define code modules that encapsulate multiple *Components*, *Services*, etc. in a defined scope.

Being able to generate all of these different parts of an application is very desirable but for our approach, we propose to first keep the complexity as low as possible and only focus on the generation of *Components* and establish a baseline to incrementally build upon. If the first results appear promising then one could think about extending the system to also be able to generate *Services* and eventually all other application parts. We focus on the generation of *Components* since they represent the basic building block in angular applications. It is possible to build an angular application solely out of *Components* which is not true for any of the other application parts.

Filtering the scraped code to only contain *Component* definitions is trivial to implement because by convention all *Component* files are named `<COMPONENT_NAME>.component.ts` and their accompanying test specifications `<COMPONENT_NAME>.component.spec.ts`. Using a regular expression, picking up on this naming convention is straightforward and yields a set of *Component* definition and test specification pairs. An example of a component test with one test case can be found in Figure 2.



```
describe('StartPageComponent', () => {
    let component: StartPageComponent;
    let fixture: ComponentFixture<StartPageComponent>;

    beforeEach(() => {
        TestBed.configureTestingModule({ declarations: [StartPageComponent]});
        fixture = TestBed.createComponent(StartPageComponent);
        component = fixture.componentInstance;
    });

    it('should contain "Start page works!"', () => {
        const startPageElement: HTMLElement = fixture.nativeElement;
        expect(startPageElement.textContent).toContain('Start page works!');
    });
});
```

Fig. 2. Example Angular component test

A first analysis has shown that using the described method $\sim 2.5M$ angular *Component* code and test pairs can be found on GitHub which is a promising first result. Details of this analysis can be found in Table I.

Type	# in thousands
Angular Repositories	383
Component Test Pairs	2 547
Service Test Pairs	1 312

TABLE I
First analysis of potential training data on Github

To ensure that the data is of high quality, we propose to filter the data further and remove samples from the data set that do not meet certain criteria. Possible exclusion criteria could be the number of test cases per test file (e.g. minimum of 3 test cases) or the amount of forks of the associated GitHub project (e.g. minimum of 5 forks). We believe these filters to be promising, since angular auto generates one test case per *Component* and we are of the opinion that more are needed for the *Component* to be considered high quality. Also we assume that GitHub users are more likely to interact with a project if they think it contains high quality code. Additionally, it is most likely beneficial to bring all of the data into a standardized form. Angular component definitions are oftentimes split up into different files each containing only either markup, styling or logic. By inlining the markup and styling into the logic part where necessary, we can omit this issue.

Several research efforts have shown promising results in ML code processing tasks when the code is converted to tokens or an AST [23]. Depending on the model that is ultimately used, this is something that we propose to explore. Since Angular 2+ code is always written in TypeScript [24] which is a compiled language that has open source tooling, the tokenization and AST conversion of our data can be achieved with production-ready, publicly available solutions.

B. Model

At the core of our proposed approach is a generative ML model and because of the recent ubiquity and performance of transformer-based models in generation and sequence-to-sequence transformation tasks [3], [4], [10], [17], [18], using a transformer in our described scenario becomes an obvious choice.

Because of the comparatively low effort involved, as a first approach, we propose to use OpenAI's Codex model which is based on GPT-3 [3]. Codex has been used in very successful code generation applications like GitHub's Copilot [25]. Because it is pre-trained on a vast amount of natural language and code and implements a high-performing architecture, using Codex can potentially reduce the development time of our system by a very large amount when compared to implementing and training a custom model. Through the use of the Codex fine-tuning API, our data set can be used to train the model and customize it to our needs.

Additionally, we suggest the use of CodeBERT [17] and CodeT5 [26] which are other state-of-the-art pre-trained transformer models designed and trained for the processing and generation of code. Comparing the performance of these models will likely give further insights into which style of transformer-based model is best suited for this type of generation task.

C. Post-Processing

Because we will use test case specifications as the input to our generation task, we have a high amount of information regarding the desired output. We can leverage this and have our model generate code samples which we then execute the tests on in a sandboxed environment. Generated samples that do not meet all assertion criteria of our tests will be discarded and only the ones that do are presented to the user. Since the test implementations of our input data can be used in this way and jasmine tests are comprised of strings containing NL and the test code as Figure 2 shows, we propose to explore the idea of only using the NL part of the test specifications as the input to our model and use the test implementation for the selection of generated results.

When generating Angular *Component* code, there will most likely be issues with *Component* dependencies. Angular relies on an Inversion of Control (IoC) mechanism, specifically Dependency Injection. It is a common pattern to have functionality or data contained in a *Service* which is then injected into multiple *Components* where the functionality or data is then accessed. When *Component* code is generated for a *Component* that depends on another part of the application that is not implemented yet, compiler errors will arise when the execution of the tests is prepared. This might be the case even though the generated code is correct and this dependency is desired by the user. To combat this issue, we propose that in another post-processing step code stubs for unmet dependencies are created. This means that if e.g. a dependency on a *Service* that does not exist yet is generated by our model, then a stub for this *Service* will also be created. Similarly, if a dependency on a method of a *Service* is generated and the *Service* exists but the method does not, a stub for the method is created in the existing *Service*.

The realization of this functionality should be straightforward because Angular provides powerful tooling for the programmatic creation and modification of parts of an existing application with Angular Schematics [27].

D. Evaluation

To answer our posed research questions, the proposed system needs to be evaluated therefore we'll describe possible evaluation strategies in the following section.

RQ1 and **RQ2** are concerned with the quality of the generated code but assessing the quality of such generation results is no easy task. Ren et al. [28] show that match-based metrics like the BLEU score are insufficient for the evaluation of generative models for code since they have problems capturing semantic features specific to code. To evaluate the output quality of our proposed system the CodeBLEU score described by Ren et al. could be used but it had to be extended to be usable with TypeScript. Additionally, an evaluation set of input-output pairs would need to be created, similar to the HumanEval set in [3]. Another possible evaluation strategy would be to have a representative group of experienced developers that assess the quality of the generated code manually.

To find a satisfactory answer to **RQ1** we propose a search over the pre-preprocessing and filter parameters (Section III-A) training multiple models and evaluating code quality in one of the ways mentioned above.

To find an answer to **RQ2** we suggest defining a threshold score that indicates whether a piece of code can be considered high quality when measured against a sample from the evaluation set. Whether the proposed system can reliably generate such code answers the research question.

Answering **RQ3** is straightforward since test coverage analysis is part of the angular tooling. Such analysis could be run after the generation and the results aggregated to calculate min, max, and average branch and instruction coverage on test data.

IV. Discussion on the impact of our vision

In previous chapters, we have explained in detail how our proposed idea can be executed. We will now highlight the potential value of this work.

If enough high-quality data could be mined (**RQ1**) to successfully train an ML model that can generate high-quality code (**RQ2**) the viability of our approach would be demonstrated. Because of Angular's popularity, [20] this would imply potentially huge time and therefore cost savings for many people and companies.

Additionally, it would open the door to further research that tries to achieve something similar on another platform. An interesting aspect that emerges from the fact that we rely on BDD tests is that in an enterprise scenario, the BDD test cases often define the acceptance criteria for user stories of the stakeholders. This implies that if our approach can successfully generate application code that satisfies the tests, the code has already passed quality assurance and is accepted from a business perspective, theoretically eliminating all initial programming work that deals with the implementation of actual application code.

But even if the BDD test cases do not directly correspond with acceptance criteria, we theorize that there are more benefits to our approach than simple time-saving. During development, software developers usually take the most time-efficient route when solving a problem. This is especially true when the project is on a tight schedule, as they oftentimes are. What appears as the most time-efficient route sometimes does not correspond with the optimal route from a software engineering standpoint. The result of this is that sometimes developers do not focus on writing enough meaningful tests, since tests do not directly contribute to application functionality. If our approach succeeds in generating high-quality code from test specifications, developers would be provided with a very time-efficient way of solving their problems fast that requires them to write tests. Additionally, we theorize that higher quality output can be generated by our proposed method when higher quality input i.e. test specifications are provided which would further incentivize developers to write more meaningful tests. In case our approach yields equal or higher test coverage (**RQ3**) than what is generally achieved in Angular projects, one could argue that the adoption of our techniques tends to increase software quality which is a very desirable trait.

Furthermore, we theorize that using just the natural language section of the BDD tests will generate better results than using the tests in their entirety. Even if our theory is ultimately wrong, knowing the answer to this question will be beneficial for other researchers in the field potentially saving work or sparking research in different directions.

V. Conclusion

The field of ML-based code generation is just emerging and to our knowledge, there does not exist any research that tries to achieve what we propose to do. We describe a possible method for generating Angular frontend components based on BDD test specifications by utilizing ML techniques. Our approach aims to reduce the development time needed for web applications and introduce new research ideas toward automatic code generation.

From our impact analysis, we believe that this approach has the potential to bring many benefits to academia and industry, which is why we advocate for further research in this direction.

VI. Acknowledgement

This research work has been funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE and Crossing SFB.

References

[1] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 213–222. [Online]. Available: https://doi.org/10.1145/1595696.1595728

[2] M. Soeken, R. Wille, and R. Drechsler, "Assisted behavior driven development using natural language processing," in *Objects, Models, Components, Patterns*, C. A. Furia and S. Nanz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 269–287.

[3] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[4] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 150–162.

[5] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[6] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: Ai-assisted code completion system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 2727–2735.

[7] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "Cscc: Simple, efficient, context sensitive code completion," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 71–80.

[8] C. Wang, F. Pastore, A. Goknil, and L. Briand, "Automatic generation of acceptance test cases from use case specifications: an nlp-based approach," *IEEE Transactions on Software Engineering*, 2020.

[9] M. J. Escalona, J. J. Gutierrez, M. Mejías, G. Aragón, I. Ramos, J. Torres, and F. J. Domínguez, "An overview on test generation from functional requirements," *J. Syst. Softw.*, vol. 84, no. 8, p. 1379–1393, aug 2011. [Online]. Available: https://doi.org/10.1016/j.jss.2011.03.051

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.

[11] A. Trudova, M. Dolezel, and A. Buchalcevová, "Artificial intelligence in software test automation: A systematic literature review," in *ENASE*, 2020.

[12] P. Stevens, "Understanding the differences between bdd & tdd." [Online]. Available: https://cucumber.io/blog/bdd/bdd-vs-tdd/

[13] S. Fox, "All you need to know about behaviour-driven software." [Online]. Available: https://web.archive.org/web/20150901151029/http://behaviourdriven.org/

[14] D. North, "Introducing bdd." [Online]. Available: https://dannorth.net/introducing-bdd/

[15] Cucumber, "Cucumber." [Online]. Available: https://cucumber.io/

[16] diaspora, "diaspora," https://github.com/diaspora/diaspora, 2022.

[17] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

[18] N. Chirkova and S. Troshin, "Empirical study of transformers for source code," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 703–715. [Online]. Available: https://doi.org/10.1145/3468264.3468611

[19] Angular, "Angular." [Online]. Available: https://angular.io/

[20] S. of JS, "State of js 2021." [Online]. Available: https://2022.stateofjs.com/en-US/libraries/front-end-frameworks/

[21] React, "React." [Online]. Available: https://reactjs.org/

[22] Jasmine, "Jasmine." [Online]. Available: https://jasmine.github.io/

[23] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *CoRR*, vol. abs/1709.06182, 2017. [Online]. Available: http://arxiv.org/abs/1709.06182

[24] TypeScript, "Typescript." [Online]. Available: https://www.typescriptlang.org/

[25] Github, "Github copilot." [Online]. Available: https://github.com/features/copilot

[26] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *CoRR*, vol. abs/2109.00859, 2021. [Online]. Available: https://arxiv.org/abs/2109.00859

[27] Angular, "Angular schematics." [Online]. Available: https://angular.io/guide/schematics

[28] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," 2020. [Online]. Available: https://arxiv.org/abs/2009.10297

Evaluating and improving transformers pre-trained on ASTs for Code Completion

Marcel Ochs*, Krishna Narasimhan†, Mira Mezini†
*Independent
ochs.marcel@gmail.com
†TU Darmstadt
kri.nara@st.informatik.tu-darmstadt.de, mezini@informatik.tu-darmstadt.de

***Abstract*—Automatic code completion is one of the most popular developer assistance features which is usually performed using static program analysis. But, studies have shown that static analysis can be riddled with false positives. Another issue with static-analysis based code completion is that the recommendation does not take into account the history/ context in which the software is operating and rely on type/ alphabetical ordering of suggestions. A recent development that has shown to be promising in this direction is the use of language models such as transformers that are trained on real-world code from Github to provide context sensitive, accurate code completion suggestions. Studies on transformer-based code completion have shown that such restrictions can be leveraged; i.e; training transformers on structural representations of code (specifically ASTs) could have a positive impact on the accuracy of code completion. To this end, the work by Kim et al. implemented and evaluated TravTrans which is based on the powerful text prediction language model GPT-2 by training on abstract syntax trees instead of treating code as plain texts. Using alternative source code representation such as AST provides the already potent language model with an additional layer of program semantic-awareness. But, TravTrans has adapted several rigid choices regarding various components of the transformer architecture like embedding sizes, sliding windows etc. TravTrans also suffers from issues related to running out of vocabulary. In this paper, we reproduce the TravTrans model and perform a deeper, fine-grained analysis of the impact of various architectural and code-level settings on the prediction. As a result of our fine-grained analysis, we also identify several aspects that need improvements like the fact that the model performs particularly poorly with code involving dictionaries and lists. We also offer solutions to a few of the issues like the out-of-vocabulary issue. Finally, our results motivates the need for a customizable-transformer architecture for coding tasks.**

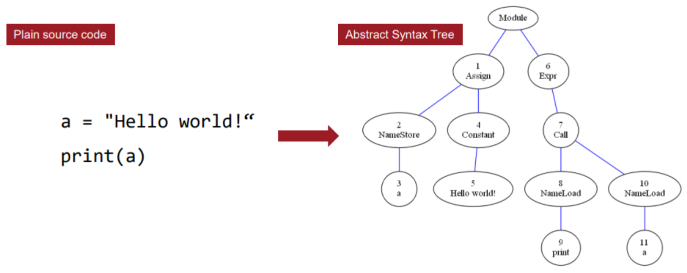
I. Introduction

Automatic code completion or prediction is the most widely used and integral developer assistance feature [8, 16, 11]. Besides reducing development time by offering quick completions for developers, it allows inexperienced developers to find their way around unknown frameworks and libraries. The traditional approach for automatic code completion, *static analysis* scans the code during compile time and offers suggestions on the next token that should follow. However, static analysis is known to suffer from false positives [7]. Another downside of this approach is the ordering of the suggestions: Oftentimes the suggestions are sorted alphabetically rather than sorted by relevance, forcing the developer to scroll through a list of suggestions, nullifying the development speed advantage sought after by code completion.

In the meantime, the last decade has seen a surge of learning-based, data-driven approaches that train language models on vast amounts of data which are then used to perform classification and prediction tasks. Such approaches are primarily used in the context of natural language text processing. Current state-of-the-art language models typically rely on the transformer architecture [17]. Transformer implementations such as the *generative pre-trained transformer* (GPT) or the *bidirectional encode representations from transformer* (BERT) were developed and tailored for natural language tasks such as text generation, question answering and machine translation [13]. Popular examples are the GPT-3 [1] and its open-source cousin GPT-2 [19], both of which have illustrated impressive abilities to generate, summarize and classify texts with close to human quality. A natural next-step is to train such language models for code and use them for tasks such as code completion. But treating code as text during the training process ignores several structural constraints that could aid with more precise results. Consider for example the fact that while there are numerous ways to describe a certain idea in natural language, there is only a limited number of ways to describe a process in a programming language. For example, changing the word orderings or even omitting entire words from a sentence can still get a message across in the world of natural languages. With programming languages, even the slightest changes such as punctuation or missing brackets would result in a non-compiling or error throwing program. But such restrictions can be harnessed for a language models benefit. Consider for e.g., variable `a` in the second line of the code snippet in Figure 1. A language model could interpret it as a variable, string literal, functional call etc. unless it has further structural and semantic information which are generally captured in intermediate representations of code like the abstract syntax tree (AST) [1]. Figure 1 illustrates such an example AST of the code snippet where the `a` is augmented with additional information in its ancestor nodes. Any model that is trained on ASTs can harness this information and predict that the `a` is parameter to the function call `print` and serves as reference

[1]https://en.wikipedia.org/wiki/Abstract_syntax_tree

to a string variable with the value `Hello world`.



The model presented by Facebook(now Meta) Research Group [8] and published at ICSE 2021 called *TravTrans* makes use of this finding: *TravTrans* is a slightly modified transformer model (GPT-2 to be exact) trained on abstract syntax trees. TravTrans introduces minimal modifications to the state-of-the-art GPT-2 architecture thereby reducing overhead and complexity. Although there are numerous code language models attempting to aid code completion they suffer from some disadvantages. Some of them are too large like GPT-3 [2] and GitHub's Copilot[2] and thereby prevent usability in low-resource settings such as research. The rest of them like IntelliCode Compose [15] are trained on textual representations of code that require lots of post-processing to handle corner cases. Due to its simplicity and superior performance which is also validated by a large scale study of transformers for code by Chirkova et al. [3], *TravTrans* is the ideal language model for our investigative study. *TravTrans+*, an "enhanced" variant of *TravTrans*, also introduced by Meta Research introduces significant performance and memory overhead in return for a mere 1% performance [8]. As the additional overhead does not justify the minor performance upgrade, we base our study on *TravTrans*. The goal of this paper is to perform a systematic, investigative study on transformers trained on ASTs for code with *TravTrans* as a basis with the goal of identifying potential issues that can serve as a basis for future researchers to build on. To this end, the contributions of our study are three-fold. Firstly, we apply a magnifying glass on each of the components of the transformer architecture. For e.g., the original work chooses, among other things, a particular embedding size and number of layers without elaborating on why these choices were made. Our work sheds light on the impact of various embedding size and number of layers on the overall performance. Secondly, we identify weak points of *TravTrans* model and discuss potential improvements. For e.g., the code completion for string nodes suffers from out-of-vocabulary-issue and also has extremely poor prediction with code involving *dict* and *list* types. For the out-of-vocabulary issue, we identify that utilizing an alternate tokenizer resolves the issue. Thirdly, the overall insights from our study clearly suggests the need for a modular, customizable transformer model for code-completion. Specifically, we answer the following research question:

RQ1: How does *TravTrans* perform when reproduced? Meta Research published their *TravTrans* model on GitHub but did not provide the accompanying training and evaluation scripts during the time of our investigation. This RQ aims to re-implement *TravTrans* from scratch and the evaluation score of the re-implemented model will serve as baseline for subsequent RQs.

RQ2: The *TravTrans* model architecture contains several properties whose values are not justified by the authors, such as the number of decoder layers or embedding sizes. This RQ aims to look into these settings and to evaluate whether **TravTans can be improved by adjusting architectural settings?**.

RQ3: Although the *TravTrans* model grouped their predictions by kinds of nodes such as *parameter name* and *string*, they miss several important categories such as *assignments*, *function calls* and *strings*. This RQ intends understand whether **additional node types should be tracked for model evaluation?**.

RQ4: Can the out-of-vocabulary issue be reduced by using an alternative tokenizer? Our investigation of RQ3 revealed that *string* predictions suffer particularly because the model has limited vocabulary. We use this RQ to identify the root-cause of this issue, offer a solution (i.e, an alternate tokenizer) and evaluate the results of our proposed improvement.

RQ5: During the process of flattening ASTs into sequences, *TravTrans* uses a sliding window which splits the sequences with a specific overlap size if an AST is larger than the model input window size. With this RQ, we understand whether **the overlap size has an impact on model performance?**

RQ6: Positional encoding captures the dependencies between tokens in transformer models which should in theory positively impact the prediction. Contrary to this, recent research suggests that the positional encoding layer in transformer models may be damaging to the model performance. The authors of *TravTrans* have therefore removed the positional encoding layer from the baseline GPT-2 model. This RQ attends to find out if **positional encoding improves the performance of the model?**.

A. Data availability

All our code, scripts and modifications to the original TravTrans model are open-sourced in our fork of the original repository [3] along with detailed guides on how to reproduce the results of all RQs [4]. For all of the RQs, reviewers can also observe how we generated our charts directly in the browser using demonstrator Jupyter notebooks which are linked at the end of respective RQs. For more accurate results, we recommend the reviewers run the scripts instead of using the demonstrator which is only intended as a quick showcase.

B. Organization

The rest of the paper is organized as follows. Section II describes on a high-level the setup of the original TravTrans

[2] https://copilot.github.com/

[3] https://github.com/derochs/code-prediction-transformer

[4] https://github.com/derochs/code-prediction-transformer/blob/master/guide.ipynb

model. Section III describes the results of each of the RQs. In the process, we also describe the modifications performed to the original model to obtain the results. Section IV presents related work and Section V concludes.

II. Original TravTrans Model

Our baseline model, *TravTrans* is a slightly adjusted GPT-2 model. A full description of the GPT-2 architecture is beyond the scope of this paper. An illustrative, easy-to-follow guide for GPT-2 is here [5]. The differences between the two models are as follows: The original GPT-2 model uses 12 layers and contains a positional encoding layer while TravTrans uses 6 layers without a positional encoding layer. *TravTrans* is trained from scratch on the *150k Python Dataset*[6] consisting of $150,000$ abstract syntax trees parsed from Python source code and converted into sequence using pre-order traversal [7]. By default, the *Py150k* dataset is split $66 : 33$ ($100,000$ training samples and $50,000$ evaluation samples). However, *TravTrans* utilizes a $70 : 15 : 15$ split ($105,000$ training samples, $22,500$ validation and $22,500$ evaluation samples). The extra validation samples are used to evaluate and compare models with different hyperparameters whereas evaluation samples are used to score the model with the best performing hyperparameters.

Any language model requires a fixed sized vocabulary which acts as a database for every word the model is able to predict. In *TravTrans*, the vocabulary is a Python dictionary which maps the $100,000$ most frequent AST node values to an integer ID. The higher the frequency the lower the ID (ranging from $99,999$ to 0). The most frequently occuring $100,000$ tokens are then pickled[8] and stored in a dictionary. In addition to the $100,000$ most frequents tokens there are two special tokens. One is a padding token, that does not contain any information and is just used to pad an input sequence to fill the entire model's input window. TravTrans ignores these nodes during predictions. The other one is used for tokens which are out-of-vocabulary: a node that does not belong to the top $100,000$ most frequent tokens in the vocabulary which are mapped to **unk_token**. The model will process this special token just like any other token.

Tokenizers deal with the process of splitting up sentences into words and assigning each word to an ID and vice versa. The *TravTrans* tokenizer is implemented as a *Word Level Tokenizer* that treats each AST node as a single token. Even if a node contains a string consisting of several words, this is treated as one single token. Such a Word Level Tokenizer is easy to implement but the downside is that unique tokens such as rare string values or variable names are hard to predict due to the aforementioned out-of-vocabulary issues. We will discuss the out-of-vocabulary issue in detail when presenting the results of **RQ4**.

The training and evaluation data comes from the Python 150k dataset which contains ASTs parsed from Python source code.

[5]https://jalammar.github.io/illustrated-gpt2/
[6]https://www.sri.inf.ethz.ch/py150
[7]The data-processing is detailed later in the paper
[8]https://docs.python.org/3/library/pickle.html

The data files are in *JSONL* (JSON Line) format and every AST is represented as a JSON object. Each line in the dataset contains one JSON object per AST. As the *TravTrans* expects the trees nodes to only contain node type (e.g., *NameLoad*, *Str*) **or** value (*e.g., string constants like "Hello World"*), any node that contains both type and value together are pre-processed to discard the type information.

After being processed, the trees have to be split up as *TravTrans* limits context size[9] to $1,000$ and the dataset contains ASTs which are larger than the context size. The larger trees are split up into smaller trees of size *max_length* (by default $1,000$) with an overlap of size $max\_length/2$ to track links between slices. Since *TravTrans* expects a sequence of token IDs rather than trees as JSON objects, the tree slices are traversed in pre-order sequence [10]. The resulting sequence is then stored in a comma separated text file, each line representing a single sequenced AST slice. The values that were previously stored in a text file have to be converted into their respective token IDs by the tokenizer in order to make them compatible with the transformer model. The conversion routine reads the previously generated text file and replaces every token with its corresponding ID by looking up the tokens in the vocabulary dictionary. The evaluation process clusters all tree nodes into a handful of evaluation categories in order to be able to distinguish between certain prediction types such as attribute access predictions, function call predictions or string predictions. Otherwise, there would only be a single performance score for the entire model. In order to allow such a classification of prediction types, the evaluation script utilizes two files: a datapoints(dps.txt [11]) file that contains the actual token IDs from the previous step and a ids file(ids.txt [12]) that contains a lookup table which returns the evaluation category of any node.

TravTrans is implemented with PyTorch (v1.9.0) using Python (v3.8.X) and the original model can be found in Meta's GitHub repository[13]. Their implementation is a fork of a basic GPT-2 PyTorch implementation [14]. For the purpose of evaluating the correctness of a predicted number of solutions, the `mean reciprocal rank` (MRR) is used [9].

$$MRR = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{rank_i}$$

where n is the total number of calculated ranks and $rank_i$ is the ith rank. As the trained model does not generate a single prediction for the upcoming word but rather generates a probability distribution for the entire vocabulary, **Beam Search** is used [15] for the selection of the right word. Beam Search has

[9]The amount of elements which can be passed to the model per iteration
[10]https://en.wikipedia.org/wiki/Tree_traversal
[11]https://github.com/derochs/code-prediction-transformer/blob/master/sample/dps.txt
[12]https://github.com/derochs/code-prediction-transformer/blob/master/sample/ids.txt
[13]https://github.com/facebookresearch/code-prediction-transformer
[14]https://github.com/graykode/gpt-2-Pytorch
[15]https://en.wikipedia.org/wiki/Beam_search

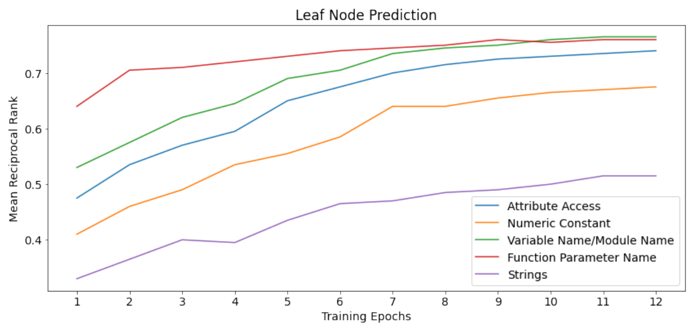


Figure 1. Scores for leaf node prediction

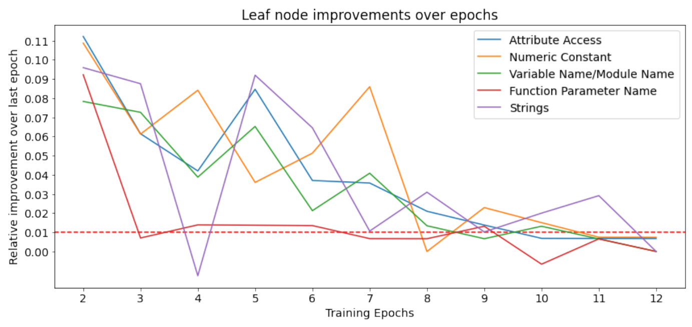


Figure 3. Relative leaf node score improvements over epochs

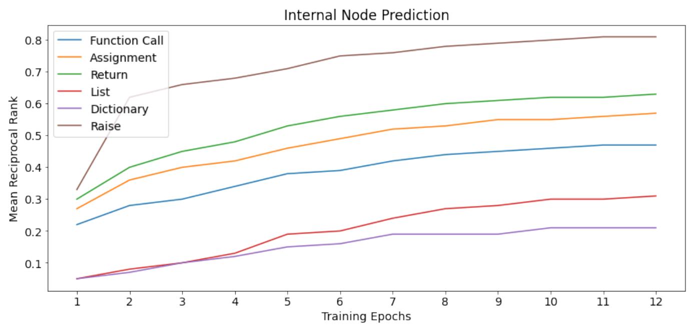


Figure 2. Scores for internal node prediction

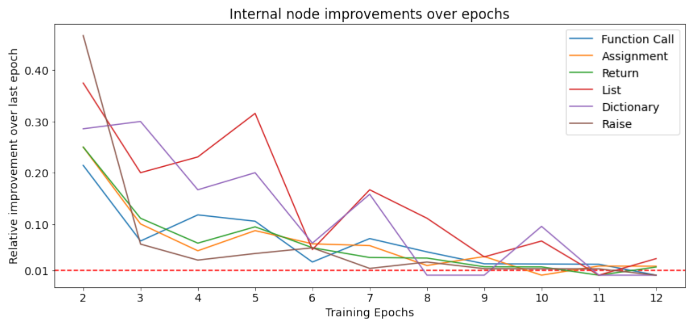


Figure 4. Relative internal node score improvements over epochs

shown to be better than the naive, greedy approach that picks the prediction with the highest probability.

III. Study Results

Now we describe the results of our study by answering RQs presented in Section I and present any modifications we performed to the original model.

A. *RQ1: How does the baseline model perform when reproduced?*

As training or evaluation scripts were not made publicly available by the work of Kim et al. [8] during the time of this work, they had to be implemented from scratch potentially providing minor deviations from the original results. We will use the results from our re-implementation as a basis for comparison in upcoming RQs. Figure 1 and Figure 2 present the score for leaf nodes and internal nodes respectively.

The y-axis shows the MRR score and the x-axis shows the current training epoch. During one training epoch, the entire training dataset is fed through the model. Having 12 epochs means that the entire training dataset was fed through the model 12 times. The leaf node predictions actually require two predictions (the parent node representing the node type and the child node representing the value), meaning that there are actually two scores. Figure 1 visualizes the mean of both scores. According to Kim et al. [8], the original *TravTrans* model was trained for 11 epochs. In order to capture the ideal number of epochs, we train our model for 12 epochs and record the relative improvement as shown in Figure 3 and 4. Given that after 10 epochs, both internal and leaf nodes do not improve more than 1%, we can assume this to be an ideal choice of epochs.

Tables I and II show the final scores of the reproduced model compared to the original work. While most of the scores and trends are similar (for example internal node prediction for “Dictionary” are both rather low compared to the other values), there are a few outliers. The reproduced model seems to perform slightly worse than the original model on internal node predictions, but for leaf node predictions it's the other way around. There are several aspects that could cause these differences:

- It is unclear how the original work merged the leaf node prediction score and the parent node predictions. While this paper used the mean of both values, the work by Kim et al. [8] could have used a different measure.
- Another reason for dissimilar results could be model parameters: Instead of a learning rate of $1e-3$, the reproduced model used a learning rate of $1e-5$ and instead of an Adam optimizer, the reproduced model uses the AdamW optimizer which handles weight the decay slightly different. These changes were applied as the settings presented by Kim et al. [8] resulted in worse results.

A demonstrator jupyter notebook illustrating how the charts were generated is made available [16]

[16] https://github.com/derochs/code-prediction-transformer/blob/master/rq1.ipynb

	Internal Node Predictions	
Application	Original Model	Reproduced Model
Function call	0.88	0.47
Assignment	0.78	0.57
Return	0.67	0.63
List	0.76	0.31
Dictionary	0.15	0.21
Raise	0.63	0.81

Table I
BASELINE MODEL TYPE PREDICTION AFTER TRAINING FOR 12 EPOCHS

	Leaf Node Predictions	
Application	Original Model	Reproduced Model
Attribute access	0.6	0.74
Numeric constant	0.57	0.68
Variable name, module name	0.63	0.77
Function parameter name	0.65	0.76
String	N/A	0.52

Table II
BASELINE MODEL VALUE PREDICTION AFTER TRAINING FOR 12 EPOCHS

Research Question 1

Despite the fact that training and evaluation routines had to be implemented from scratch, the results from the work by Kim et al. and our implementation are similar with expected exceptions.

B. RQ2: Can the baseline model be improved by adjusting architectural settings?

A transformer model consists of many parameters which can be tweaked individually, such as number of layers, embedding size, context size and number of attention heads to name a few. This RQ aims to investigate some parameters of the model architecture in order to discover whether the default *TravTrans* values work well or if there are alternative settings able to improve the overall model performance. The scores from RQ1 are treated as baseline scores and will be used for comparison. The two parameters investigated for this research question are the **number of decoder blocks** and the **embedding size**. Transformers contain various encoder and decoder layers stacked on top of each other. The number of layers in this RQ represents how many decoders are repeatedly stacked on top of itself. Radford et al. [13] who did a foundational work on language models concluded an ideal setting of 12 decoder blocks while *TravTrans* uses 6 decoder blocks. An embedding size is the size of the internal vector representation of a token in the transformer model. In case of *TravTrans*, each AST node is encoded into an integer ID which is mapped to a fixed size vector, whose size is known as the embedding size. A larger embedding size or a larger number of decoder blocks should improve the model performance. However, these changes come at a cost: An increase of embedding size or number of layers will also potentially increase training time. Figure 5 shows side-by-side how much training time was required and impact the two aspects mentioned above have on the model score. The red vertical line marks the default values chosen by Kim et al. [8]. Increasing the layer count from 6 to 9 increases the model performance by 5% and training time by 20%. Increasing the embedding size from 300 to 540 increases model performance by 5% and training time by 55%. A trade-off between model performance and training time has to be found which Kim et al. [8] seem to have achieved by using the optimal settings (6 layers, embedding size 300) as their default. A demonstrator jupyter notebook illustrating how the charts were generated is made available [17].

Research Question 2

The original evaluation of TravTrans has operated on an optimal number of layers and embedding sizes.

C. RQ3: Should additional node types be tracked for model evaluation?

TravTrans tracks the model performance for the following node types:

- **Internal nodes:**
  - `Call`
  - `Assign`
  - `Return`
  - `ListComp, ListLoad, ListStore`
  - `DictComp, DictLoad, DictStore`
  - `Raise`
- **Leaf nodes:**
  - `attr`
  - `Num`
  - `NameLoad, NameStore, NameParam`

These are only a small subset of all node types that can be found in the *py150k* dataset. Several important types of nodes including but not limited to conditionals (`if`), string (`str`), boolean operators (`CompareEq/ CompareIn` etc.) and tuples operations (`TupleDel/ TupleLoad` etc.) are not tracked. Including additional node types to be tracked during model evaluation allows deeper insights about its prediction accuracy. Figures 6 and 7 visualize the newly suggested evaluation types introduced by RQ3 alongside the baseline evaluation types. A demonstrator jupyter notebook illustrating how the charts were generated is made available [18]

Research Question 3

TravTrans has not tracked several important node types including string nodes which are shown to be particularly poor performing inspite of being some of the most commonly used node types.

D. RQ4: Can the out-of-vocabulary issue be reduced by using an alternative tokenizer?

The *TravTrans* model uses a *WordLevel* tokenizer. The advantage of a WordLevel tokenizer is its simplicity and

[17] https://github.com/derochs/code-prediction-transformer/blob/master/rq2.ipynb

[18] https://github.com/derochs/code-prediction-transformer/blob/master/rq3.ipynb

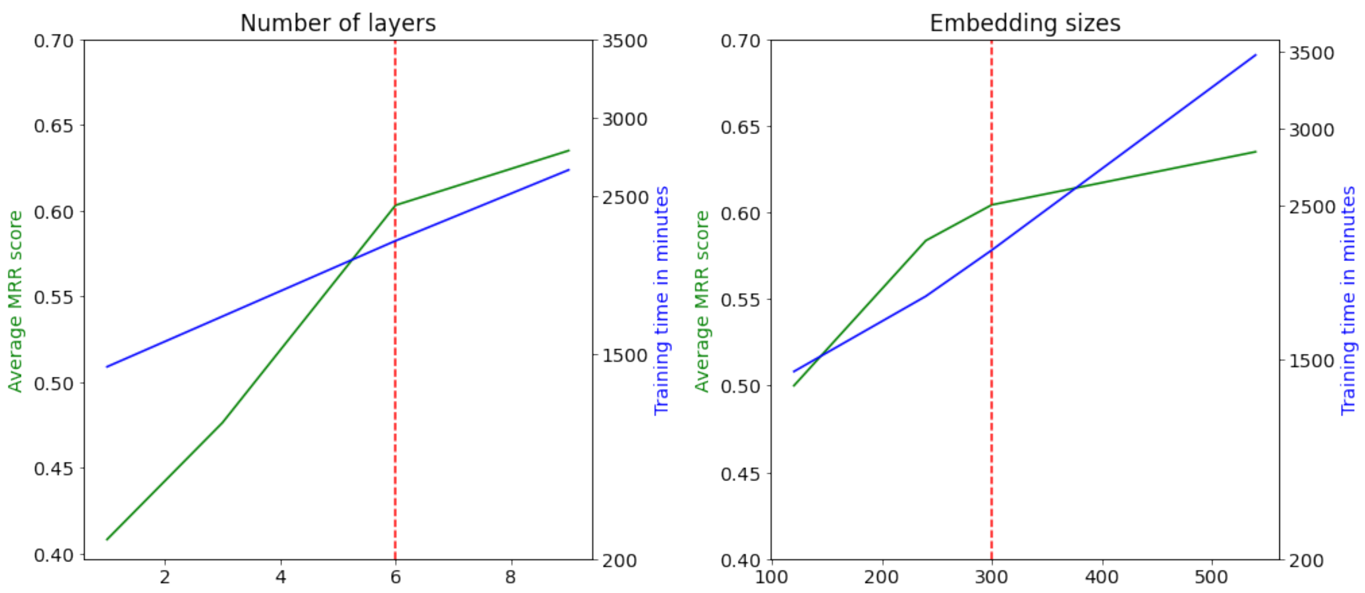


Figure 5. Alternative number of layers and embedding sizes

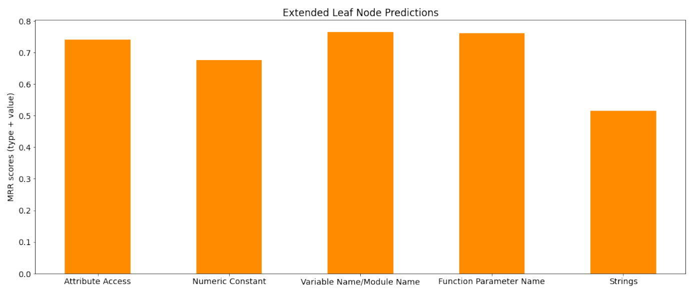


Figure 6. Model evaluation with new leaf node types introduced in RQ3

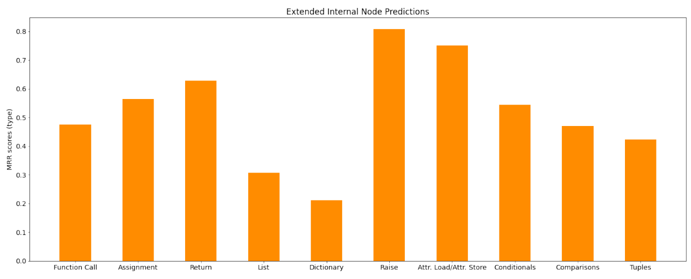


Figure 7. Model evaluation with new internal node types introduced in RQ3

predictability: One AST node encodes one integer ID and vice-versa using a simple dictionary. **The out-of-vocabulary issue**[8, 18] is one big downside of the WordLevel tokenizer: A token that is not available in the top $100,000$ frequently occuring tokens will be mapped to the unknown token resulting in loss of information. The likelihood of the exact same string or custom variable name appearing multiple times in the dataset and therefore being in the top $100,000$ most frequent tokens is low. Strings (and custom variable names) are therefore prone to be out-of-vocabulary which could also be the reason for their poor performance in RQ3. We hypothesize that a *WordPiece* tokenizer [18] that splits up tokens into the most frequent subwords can resolve this issue. For e.g., using a single string **Hello World**, a wordpiece tokenizer can track three substrings **Hello**, **World** and **Hello World**. The disadvantages of the WordPiece tokenizer are *complexity* and *determinism*: Encoding one AST node could result either in a single token or numerous subwords. This makes it harder to calculate and fit tokens into the context window. Using the baseline WordLevel tokenizer, one could count $1,000$ AST nodes and they would fit perfectly into the model with context size $1,000$. The WordPiece tokenizer on the other hand does not encode tokens $1:1$ since one token could result in numerous outputs (in the example above a single string token is tokenized into 3 subwords). Therefore, the entire dataset has to be tokenized and then split into subwords. Afterwards, the subwords can be counted in order to fit them into the model context size. The second problem is determinism: If the WordLevel based model predicts one token, it is assured to be the next AST node. However, when predicting tokens using the WordPiece approach, the generated prediction could be the correct following token or just a part of it. An additional check has to be performed in order to guarantee that the prediction is complete or if more subwords could follow. The model could also get stuck in a loop generating subwords which could introduce problems during training and evaluation. To this end, we built a new tokenizer pipeline using Huggingface's Tokenizer library[19] parametrized as follows:

- **Pre-tokenizer** : Pre-tokenizers tell the framework how unique words are identified. For this purpose, we use a A *CharDelimiterSplit* pre-tokenizer that uses a delimiter, in our case **the comma (,)**.

[19]https://huggingface.co/docs/tokenizers/python/latest/index.html

- **Actual tokenizer** : Hugging-face offers a variety of pre-implemented tokenizers to choose from. We chose the **subword** tokenizer that splits less frequent words into frequent subwords[14].

Firstly, the entire *py150k* dataset is prepared by using the same pre-processing routine from the baseline model which ensures that each AST node only contains either a type or a value, but never both. Afterwards, every comma character was escaped from the tree nodes (such as strings), as commas are used as delimiter. Raw node values and types are exported into a new file which ($150,000$ lines). This file was used to train the tokenizer, using the trainer script provided by the tokenizer framework. The vocabulary size was set to $30,000$. The trained tokenizer was then exported as a JSON file for later use. For training, the py150k dataset was split up into a train dataset containing $105,000$ samples. Similar to the dataset for the tokenizer training, the ASTs are pre-processed in order to ensure that each AST node contains either type or value but never both. In the baseline *TravTrans* data pre-processing, the next step would have been splitting up ASTs larger than $1,000$ nodes into subtrees. With the subword tokenizer however, a single node could be converted into multiple subwords, meaning that a tree with $1,000$ nodes could end up in a collection of far more tokens and therefore wouldn't fit into the context size of $1,000$. This means that before splitting up the ASTs, the types and values of each AST have to be encoded by using the previously trained tokenizer first. The output of tokenizing an AST is a list of integers, each integer being the ID of a subword. As the length of the integer list is final and will not increase, the converted ID sequence was split into chunks of length $1,000$ with an overlap of 50%. Finally, the baseline model is trained from scratch on the dataset based on the new tokenizer implementation. The model was trained over 10 epochs with a batch size of 4.

Figures 8 and 9 compare the leaf node and internal node prediction scores between *TravTrans* with WordLevel tokenizer (Baseline model) and *TravTrans* with the WordPiece tokenizer (Model with subword tokenizer). The results indicate that the string prediction accuracy doubled when using the subword tokenizer. Variable name prediction also experienced a slight accuracy increase. Other leaf node predictions show very similar results to the WordLevel tokenizer approach, without any notable differences.

What stands out is that the subword tokenizer seems to perform poorly on some internal node predictions, namely `Return`, `Raise` and especially `Dict`. The evaluation category for `Dict` contains the following node types: `DictComp`, `DictLoad` and `DictStore`. Despite their similar naming, the subword tokenizer splits `DictLoad` and `DictStore` into two subwords (`Dict` and `Store`). `DictComp` on the other hand is not split during tokenizing process, implying that this type is more frequent in the dataset and therefore does not have to be split up. When the model tries to make a `Dict` prediction, it will most likely be successful predicting `DictComp`. However, if the model is supposed to predict `DictStore` or `DictLoad` the model is more likely to have a

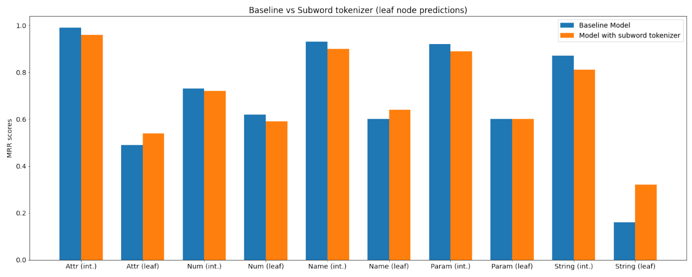


Figure 8. Comparison between baseline model and baseline model with subword tokenizer for leaf node prediction

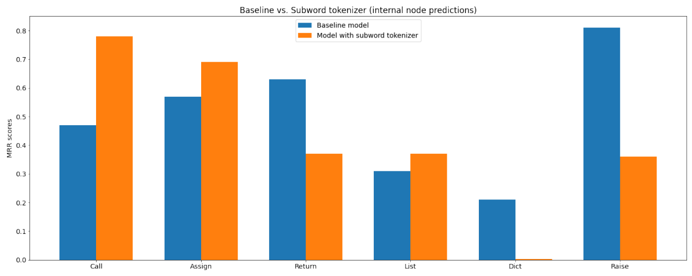


Figure 9. Comparison between baseline model and baseline model with subword tokenizer for internal node prediction

lower accuracy as it would have to make two correct predictions: First `Dict` followed by the prefix: `Load` or `Store`. This issue could explain the low accuracy for `Dict` predictions. Regarding the low accuracy of `Raise` predictions, it should be mentioned that `Raise` nodes are relatively rare in the python150k dataset. From $92,758,587$ AST nodes in the entire dataset, only $156,416$ nodes (0.1%) are of type `Raise` in the entire dataset. This makes it hard for the model to train the correct usage of this specific node type and therefore results in a lower prediction accuracy. Secondly, the rankinsg assigned by the tokenizers for similar words differ. The WordLevel tokenizer assigns the word `Raise` with the ID 58. A lower ID means that the word occurs more often in the dataset and the baseline tokenizer vocabulary contains $100,000$ words. The WordPiece tokenizer introduced in RQ4 on the other hand assigns the same word the ID 660 from a total of $30,000$ words. This may result in the worse prediction accuracy for this node type. Similar to the `Raise` type, the `Return` type is ranked lower in the subword tokenizer vocabulary than in the baseline vocabulary which could have an impact on the prediction accuracy. When it comes to the code completion task, the leaf node predictions are more important than internal node predictions as the user of a coding assistance system like IDEs will most likely expect suggestions which are stored in AST leaf nodes (for e.g., string values and variable names). This makes the subword tokenizer a well performing alternative to the baseline tokenizer used in RQ1. A demonstrator jupyter notebook illustrating how the charts are generated is made available [20].

[20]https://github.com/derochs/code-prediction-transformer/blob/master/rq4.ipynb

Overlap Size	Training dataset size	Training dataset, AST slice count
10%	1.4GB	174,870
30%	1.6GB	189,882
50%	1.9GB	220,162
75%	3.0GB	321,724

Table III
COMPARISON OF TRAINING DATASET SIZES GENERATED WITH DIFFERENT OVERLAP SIZES

Research Question 4

The subword tokenizer introduced in this research question is able to reduce out-of-vocabulary (OOV) issues, doubling the prediction accuracy for strings as well as variable name predictions but comes at the cost of reducing performance for certain internal nodes such as dictionaries, return and raise statements. Although leaf nodes are the primary interest for developers completing code, we recommend a combination of WordLevel and WordPiece tokenizer depending on the context.

E. RQ5: Which impact does the overlap size have on model performance?

During the baseline data pre-processing routine, larger ASTs sequences (traversed in pre-order sequence) had to be split into smaller slices with a maximum size of 1,000 tokens and a sliding window of 50%. This RQ aims to figure out why an overlap of 50% was chosen by Kim et al. [8] and which impact a larger or smaller overlap size would have on the training time or model performance respectively. In order to answer this question, a new data pre-processing routine was implemented which allows the customization of the overlap size. The following overlap sizes were investigated: 75%, 50%, 30% and 10%. This resulted in four differently sized training datasets which were then used to train four baseline transformer models over a span of 10 epochs. Finally, the four trained models were evaluated and compared. Using a larger overlap size results in a larger number of subtrees. This assumption can be validated by comparing the number of lines of all four training datasets. As training datasets contain one subtree per line, more subtrees mean more lines and a larger overall file size. Table III-E compares the different sizes of training datasets generated with different overlap sizes. The comparison shows that a smaller overlap size results in less AST slices and therefore a smaller dataset size. A large overlap on the other hand contains more redundant data, more AST slices and therefore a larger dataset size, which will presumably negatively impact training time. After having shown that a larger overlap size results in a larger dataset size, we analyzed the impact of the dataset size on the training time. As expected, a larger dataset results in an increased training time.

This relation is visualized in the training time vs. training file line count comparison in Figure 10. It shows the expected direct correlation between training time and line count with slight deviations on the 30% and 50% overlap model setups.

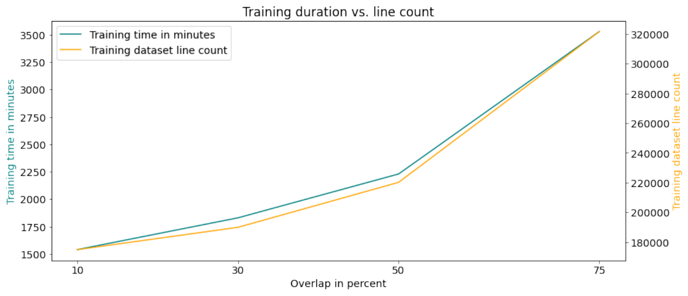


Figure 10. Training time compared to training dataset size

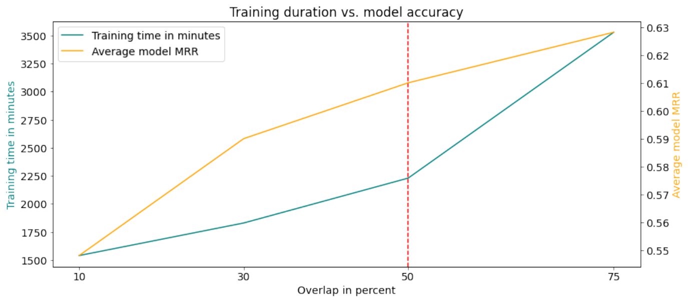


Figure 11. Training time compared to model accuracy

The teal colored plot shows how the training time increases exponentially with an increasing overlap size, while the orange plot shows how the training dataset size increases accordingly.

Finally, in order to evaluate the actual impact of different overlap sizes on the overall model performance, the four models trained on different training datasets were evaluated on the same evaluation dataset. The scores computed during evaluation are plotted in Figure 11. The teal colored plot shows the training time required by models using different overlap sizes. The orange plot illustrates the overall average model scores. While the training time seems to increase exponentially with increasing overlap sizes, the model accuracy improve correspondigly slower or occasionally even inversely: Smaller overlap sizes strongly increase model performance while only slightly increasing training time. Larger overlap sizes on the other hand result in great increases in training time while only slightly increasing model performance. This means that an optimal point between both extremes regarding overlap sizes has to be found in which training time is relatively low and model performance is relatively high. The red dotted line marks the default overlap value chosen by Kim et al. [8] for the baseline model, 50%. Increasing the overlap size above this line results in a relatively high training time increase of over 58%, while the model score experiences a relatively small improvement of less than 3%. The selected default value of 50% seems to be a good candidate for a balanced trade-off between training time and model accuracy. However, even a smaller overlap size of 30% would seem feasible as it reduces training time by almost 22% while the accuracy decreases only by 3%. Depending on the scenario the overlap size could be adjusted accordingly: Rolling out the model for production use would require a high accuracy. Training would probably be only required once, which justifies a high training time. In another scenario in which many models are trained and evaluated numerous times on expensive hardware (for e.g., for research purposes), a smaller overlap size would have been advantageous in order to speed up the training process while only slightly lowering accuracy. A demonstrator jupyter notebook illustrating how the charts were generated is made available [21].

Research Question 5

The original evaluation by Kim et al. [8] found a good balance between training time and model performance by using a default overlap size of 50%. However, an even better choice for frequently updated research settings would have been to reduce the overlap size to 30% which would reduce training time by almost 22% while only costing a decreased accuracy of 3%.

F. RQ6: Does positional embedding improve the performance of the model?

Research by Irie et al. [5] and Kim et al. [8] implies that the additional dependency information encoded by positional embedding or positional encodings are not required in transformers and could even be damaging to the overall model performance. Kim et al. [8] have consequently removed the positional embedding layer from the otherwise largely unmodified original GPT-2 model. Since *TravTrans* is a fork from a public GitHub project of a PyTorch GPT-2 implementation with slight modifications such as the removal of the positional embedding layer, we were able to re-introduce the original positional embedding implementation by fetching the implementation from the original GitHub project[22] in order to answer RQ6. Figure 12 and 13 visualize the leaf and internal node prediction accuracy of the baseline model with and without positional embedding respectively. It can be seen that the model with positional embedding achieves overall slightly worse results than the baseline model without positional embedding. There is no instance in which the model with positional encodings outperforms the baseline model and therefore there is no use case for the code prediction task. The reason for this could be the following. A model without positional encoding learns to automatically focus on the next word while the model with positional encoding is forced to focus on the first few words and therefore has a slight "blur". While positional encoding may be useful for forcing the attention of the language model on a certain part of the input sequence, it is not required for the task of text or code generation as the attention is already focused on the correct part of the sequence. A demonstrator jupyter notebook illustrating how the charts were generated is made available [23].

[21] https://github.com/derochs/code-prediction-transformer/blob/master/rq6.ipynb
[22] https://github.com/graykode/gpt-2-Pytorch

Research Question 6

The assumption that positional encoding hurts the model performance was confirmed by re-introducing the positional embedding layer from the original GPT-2 model.

G. Discussion

We have presented detailed discussions about the differences and implications of our results compared to the original model in the respective RQ subsections and about the differences in the implementation and design in our artifact guide. Here, we shortly summarize the overarching insights. Deep learning language models that operate on intermediate representations like the TravTrans which is trained on ASTs are a good next step forward in the world of code completion. But, as our study has shown, there needs to be a detailed discussion on the impact of various architectural and language-level configuration settings taken. This is especially useful if the language model needs to be used in diverse settings such as academia/ testing environments where resources may not be abundant. Our conclusions from all of the RQs especially from the use of an alternate tokenizer to resolve the out-of-vocabulary issue for string nodes and the fact that different sliding window sizes could be useful in different contexts indicate the need for a modular, customizable transformer for code completion where the user can change/ plug-and-play components and configurations (e.g., using a WordPiece tokenizer if the code completion heavily requires string completion and a WordLevel tokenizer otherwise, using a 30% sliding window overlap for research/ testing purposes and 50% for deployment).

IV. Related Work

Code completion can be regarded as a text generation problem: Given an input sentence, our code completion model should generate the next word. There are language models which achieve great results [13] in text generation. NLP deals with analyzing and processing texts for the purpose of achieving human-like language processing for a wide range of tasks or applications [10]. While the entire NLP topic can be divided into the sub fields *natural language processing* and *natural language generation*, this paper focuses on the latter topic as the main objective is the generation of text or, more specifically, source code. Language models are trained on large amounts text which enables them afterwards to generate probability distributions for the next word of a sentence. This means that given an input sentence, the model could generate a probability distribution for every word in its vocabulary, the word with the highest probability being the most likely word to continue the given sentence. Sequence-to-sequence or *seq2seq* models can be very successful at machine translation, speech

[23] https://github.com/derochs/code-prediction-transformer/blob/master/rq8.ipynb

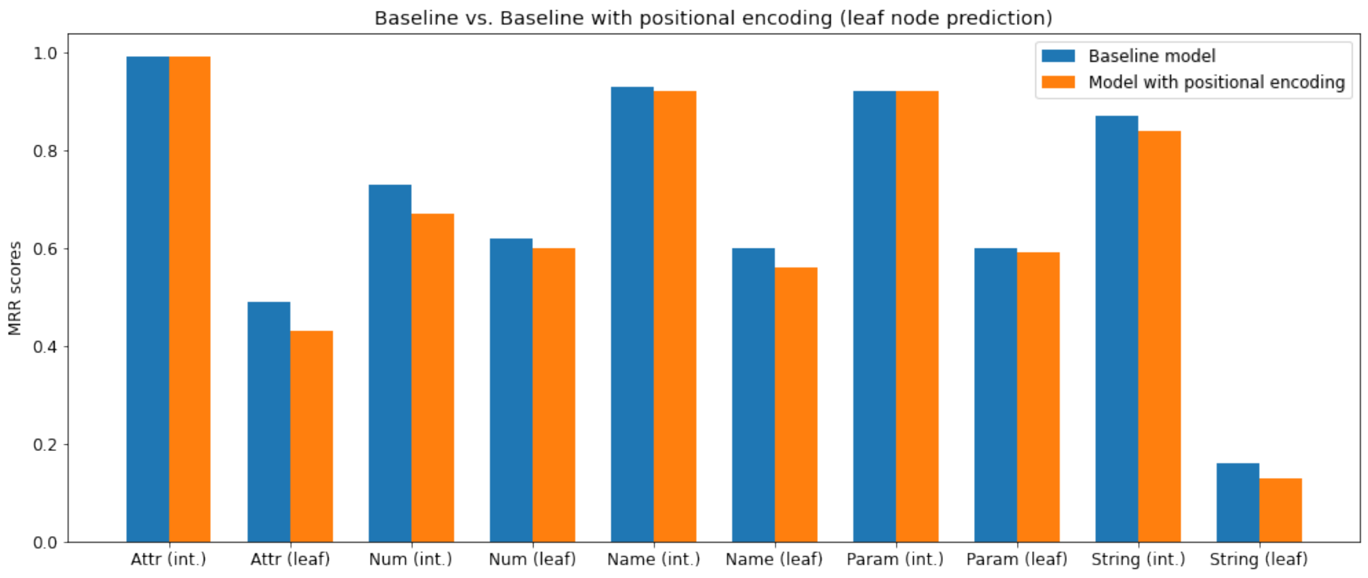


Figure 12. Baseline model vs model with positional encoding leaf node prediction

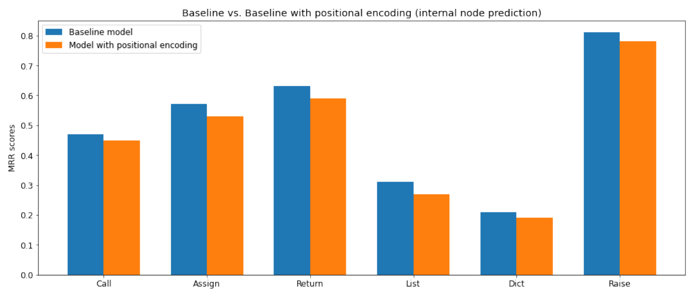


Figure 13. Baseline model vs model with positional encoding internal node prediction

recognition, image and video captioning, text summarization and dialogue modeling tasks [6]. Given an input sequence of arbitrary length which may be text or even images, the model will generate an independently sized output sequence, hence the name sequence-to-sequence model. But, transformers have been shown to illustrate higher performance when it comes to code completion. More recently program analysis leverages the power of data-driven artificial intelligence approaches to complement the traditional model-driven approaches. Recently transformers have been trained for code and shown to be effective and improve the state of the art in tasks such as code completion and variable misuse detection. An overview of transformers for source code was performed by Chirkova et al. [3]. They conclude that transformers do indeed aid with coding tasks when encoded with syntactic information in the training models. Although code completion can be a useful feature for cybersecurity related development, it is an open question whether this approach generalizes to secure programming, e.g., by predicting the appropriate secure API usage method calls. Elnaggar et al. [4] proposed an encoder-decoder transformer model to help summarize and understand source code. They do this by generating documentation for code, summarizing the source code, generating comments, git commit messages among other things. One of the cutting-edge transformer adaptations for code completion was accomplished in Microsoft's IntelliCode project [15]. They went beyond the simple next token prediction to being able to predict full statements with fair ease but this came at the cost of having to deal with several problems arising out of treating code as text. A pre-cursor to transformer technologies involved traditional Bayesian networks for code completion tasks. A successful version of this was explored by Porksch et al. [12].

V. Conclusion and Future work

In this paper, we reproduced the state of the art in transformers trained on ASTs for code completion, TravTrans by Meta Research. We first established the reasoning behind our choice of TravTrans and systematically reproduce their results. Then, we explore the impact of various architectural and code level settings on the performance of the model like embedding sizes and window overlap sizes. We also take a fine-grained look into the predictions for individual node types and observe that for important node types like string nodes, the performance drops significantly. We hypothesize the need for an alternate tokenizer to solve issues related to string nodes and validate our hypothesis. Finally, we also shed light into whether positional encoding is infact an asset to language models trained for code completion. Overall, our results indicate the need for a customizable, plug-and-play transformer for code completion where experiments such as ours can be easily conducted depending on the context, dataset and task. As a future work, we are currently exploring the impact of different program representations such as CFGs and Code property graphs, and different traversal methods (in-order/ post-order) on the model performance.

Acknowledgements

This work has been co-funded by the Crossing SFB 119 and through the support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: *CoRR* abs/2005.14165 (2020). arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

[2] Tom B. Brown et al. *Language Models are Few-Shot Learners*. URL: http://arxiv.org/pdf/2005.14165v4.

[3] Nadezhda Chirkova and Sergey Troshin. "Empirical Study of Transformers for Source Code". In: *In Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece* (2020). DOI: 10.1145/3468264.3468611. URL: http://arxiv.org/pdf/2010.07987v2.

[4] Ahmed Elnaggar et al. "CodeTrans: Towards Cracking the Language of Silicone's Code Through Self-Supervised Deep Learning and High Performance Computing". In: *CoRR* abs/2104.02443 (2021). arXiv: 2104.02443. URL: https://arxiv.org/abs/2104.02443.

[5] Kazuki Irie et al. "Language Modeling with Deep Transformers". In: *Interspeech 2019*. ISCA: ISCA, 2019, pp. 3905–3909. DOI: 10.21437/Interspeech.2019-2225.

[6] Navdeep Jaitly et al. "An Online Sequence-to-Sequence Model Using Partial Conditioning". In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee et al. Vol. 29. Curran Associates, Inc., 2016. URL: https://proceedings.neurips.cc/paper/2016/file/312351bff07989769097660a56395065-Paper.pdf.

[7] Brittany Johnson et al. "Why don't software developers use static analysis tools to find bugs?" In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 672–681. DOI: 10.1109/ICSE.2013.6606613.

[8] Seohyun Kim et al. "Code Prediction by Feeding Trees to Transformers". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021, pp. 150–162. DOI: 10.1109/ICSE43902.2021.00026.

[9] Li Li, Milos Doroslovacki, and Murray H. Loew. "Approximating the Gradient of Cross-Entropy Loss Function". In: *IEEE Access* 8 (2020), pp. 111626–111635. DOI: 10.1109/ACCESS.2020.3001531.

[10] Elizabeth D. Liddy. "Natural Language Processing". In: *Encyclopedia of Library and Information Science, 2nd Ed. NY. Marcel Decker, Inc.* (2001).

[11] Sebastian Proksch, Johannes Lerch, and Mira Mezini. "Intelligent Code Completion with Bayesian Networks". In: *ACM Transactions on Software Engineering and Methodology* 25.1 (2015), pp. 1–31. ISSN: 1049-331X. DOI: 10.1145/2744200.

[12] Sebastian Proksch, Johannes Lerch, and Mira Mezini. "Intelligent code completion with Bayesian networks". In: *Software Engineering 2016, Fachtagung des GI-Fachbereichs Softwaretechnik, 23.-26. Februar 2016, Wien, Österreich*. Ed. by Jens Knoop and Uwe Zdun. Vol. P-252. LNI. GI, 2016, pp. 25–26. URL: https://dl.gi.de/20.500.12116/761.

[13] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: 2018.

[14] Mike Schuster and Kaisuke Nakajima. "Japanese and Korean voice search". In: *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2012, pp. 5149–5152. DOI: 10.1109/ICASSP.2012.6289079.

[15] Alexey Svyatkovskiy et al. "IntelliCode Compose: Code Generation Using Transformer". In: New York, NY, USA: Association for Computing Machinery, 2020. ISBN: 9781450370431. URL: https://doi.org/10.1145/3368089.3417058.

[16] Alexey Svyatkovskiy et al. "IntelliCode compose: code generation using transformer". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Ed. by Prem Devanbu, Myra Cohen, and Thomas Zimmermann. New York, NY, USA: ACM, 2020, pp. 1433–1443. ISBN: 9781450370431. DOI: 10.1145/3368089.3417058.

[17] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[18] Yonghui Wu et al. *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. URL: http://arxiv.org/pdf/1609.08144v2.

[19] Yao Yibin et al. "Accuracy assessment and analysis for GPT2". In: *Acta Geodaetica et Cartographica Sinica* 44.7 (2015), p. 726.

Protocol-based Test Generation for API Misuse Detection Tools

Krishna Narasimhan*, Stefan Krüger†, Rakshit Krishnappa Ravi‡, Eric Bodden§, Mira Mezini*
kri.nara@st.informatik.tu-darmstadt.de, krueger.stefan.research@gmail.com, rakshit.krishnappa.ravi@rigs-it.de, eric.bodden@uni-paderborn.de and mezini@cs.tu-darmstadt.de
* TU Darmstadt, Germany, † Independent, ‡ Rigs IT, Germany, § University of Paderborn, Germany

Abstract—Misuses of application interfaces (APIs) can cause serious program flaws. Past research has proposed to identify and thus avoid misuses through static and dynamic API misuse detection tools. These tools, however, can be faulty as well, causing distracting false warnings or otherwise overlooking actual misuses.
In this work, we thus present a novel approach that facilitates test automation for API misuse detectors by automatically generating test cases from existing API usage specifications. Our specific prototype APITESTGEN supports test generation from an API specification language. Its realization required us to overcome interesting challenges because the specifications cover all of the API usage protocol types classified by previous studies; ie they not only constrain method parameters but also method-call sequences and combinations of both on multiple related API objects.
We evaluated APITESTGEN by generating tests for the Java's entire default crypto API, the JCA, and using the tests to validate the static analysis implemented in a state of the art static API misuse checker, COGNICRYPT$_{SAST}$. The evaluation found the tests generated by APITESTGEN were of high quality, significantly improved on the existing manual test writing process and revealed previously unknown bugs in the analyzer.

Index Terms—API, Test generation, Static Analysis, Cryptographic misuse detection.

1 Introduction

Software libraries provide developers with access to reusable components for their needs. But, often they are hard to understand and use, lack comprehensive and/or easy-to-understand documentation [1]. This is the case even with professionally developed APIs such as Oracle's Java Cryptographic Architecture (JCA) [2]. As a result, misuses of APIs are frequent with potentially serious consequences, e.g., when they pertain to integrating crypto APIs into applications [3, 4, 5].

To address these issues, several approaches to automate both upfront correct integration and after-the-fact misuse detection [3, 4, 5, 6] have been proposed. Misuse detectors are themselves complex pieces of software, and a thorough quality assurance process is needed for them to be effective— analyses with dozens of false positives per true positive are abandoned quickly by developers [7], and analyses with false negatives can cause developers to rightfully lose trust in the tooling. Comprehensive test suites with correct and incorrect API uses can help to assess and increase correctness, but building and maintaining them requires tedious manual effort and consumes significant resources. Hence, it is natural to ask whether the creation of comprehensive high-quality test suites may be automated.

This paper presents APITESTGEN, a novel approach to generate test cases for static API misuse detectors. It relies on explicitly specified API usage protocols as outlined by Robillard et al. [8]. Along approaches to inferring API properties, Robillard et al. surveyed and summarized different approaches to encode API usage patterns (see Section 2.1 for an overview). Given the specification of such a pattern, every test case that APITESTGEN produces, is a code snippet that encodes a correct, respectively incorrect, use of the API, followed by an assertion of what a misuse detection tool is expected to signal (an oracle). For illustration, Listing 1 shows two test cases generated by APITESTGEN, containing both a correct use of a `KeyGenerator` initialized as expected with the correct cryptographic algorithm, and a misuse where the key is initialized with an incorrect value. Running the test implies running the API misuse detector on the code snippet and comparing the errors thrown by the detector with the assertions. Test suites generated in this way not only alleviate the manual efforts of test case writers, they also provide a ground-truth data-set of correct/incorrect API uses which allows one to compare the capabilities of different misuse detectors [6].

To demonstrate the feasibility of the proposal and to empirically evaluate it, we instantiated the approach for the API usage specification language CRYSL [5], which is expressive enough to cover the patterns surveyed by Robillard et al. [8]. CRYSL specifications go beyond simple invariants; they include restrictions on method parameters, the order in which methods must be invoked, and how different objects of an API must be composed. To establish this, we summarize API usage protocols by Robillard et al. and their relation to CrySL and compare CrySL with other API specification DSLs. The complex kinds of API usage protocols expressible in CRYSL pose a number of interesting challenges for the test generation. They include generating accurate method orders from finite state machine rules, representing method sequences, generating accurate method

Listing 1: A synthetic test case generated by APITESTGEN.

```java
@Test
private void KeyGeneratorTestProperUse()
    throws GeneralSecurityException {
  //Valid use of the API
  KeyGenerator keyGen = Cipher.getInstance("
      AES");
  keyGen.generateKey();

  //Tool specific assertion
  Assertions.mustBeInAcceptingState(keyGen);
}

@Test
public void KeyGeneratorTestMisuse() throws
    GeneralSecurityException {
    //Invalid use of the API
    KeyGenerator keyGen = KeyGenerator.
        getInstance("trololo");

    //Tool specific assertion
    Assertions.mustNotBeInAcceptingState(
        keyGen);
}
```

parameters based on complex constraints, and generating accurate compositions of multiple objects of different types based on the composition predicates.

We evaluated APITESTGEN by generating tests from existing CRYSL specifications. Several crypto APIs have been modelled in CRYSL. In particular, there exist a comprehensive set of CRYSL rules covering the whole JCA.[1] These specifications are readily available to be used as an input to APITESTGEN.

We use the tests generated for protocols(rules) covering the JCA as inputs to COGNICRYPT$_{SAST}$ [5] - considered as a state of the art api misuse detector [9], [10], [11]. Given protocols and some application code, COGNICRYPT$_{SAST}$ checks the code for compliance with specifications. COGNICRYPT$_{SAST}$ comes with a set of manually crafted test cases that can be used to compare against test cases generated by APITESTGEN.

The results show that APITESTGEN significantly improves upon manual test creation. In few seconds, it generates 370 test cases. In comparison, over the past three years, 16 contributors to COGNICRYPT$_{SAST}$ have written 71 test cases altogether. The generated tests cover 4x the number of JCA classes covered by the existing manually written tests. Moreover, the generated tests prove to be of high quality. The oracle annotation of the vast majority of generated tests (309 or 83.5%) is in agreement with the result of running COGNICRYPT$_{SAST}$ on the code snippet of the tests. A manual analysis of the remaining 61 generated tests exposed 3 unique bugs in COGNICRYPT$_{SAST}$ and 3 incorrectly written CRYSL specifications (arising from 25 out of 61 disagreeing tests). We have reported these bugs as issues to the tool's maintainers. A thorough manual analysis of the remaining cases revealed some shortcomings in the prototype of APITESTGEN, all of which can be addressed with reasonable effort. The resulting accuracy of the current implementation of APITESTGEN is 90.2% (stemming from 309 agreeing and 25 accurate, disagreeing tests).

In summary, this paper's contributions are as follows:

1) APITESTGEN, a novel approach to generate test cases for API misuse detectors using API usage protocols (Section 3),
2) A study with the static analysis tool COGNICRYPT$_{SAST}$ and CRYSL (Section 1) specifications for the JCA API, evaluating the approach on whether it improves the manual test creation process in terms of coverage and time consumed as well as the quality of generated tests (Section 4)
3) A comprehensive test case suite for the full JCA API consisting of 309 APITESTGEN tests whose oracle is in agreement with the static analysis results and which can be used to check other API misuse detectors.

1. https://github.com/CROSSINGTUD/Crypto-API-Rules.

2 BACKGROUND

We briefly discuss the API usage patterns surveyed by Robillard et al. [8] (Section 2.1) followed by a brief introduction to the CRYSL language with insights on how it allows expressing each of the API usage patterns (Section 2.2).

2.1 API Usage Patterns

Robillard et al. performed a consolidator survey of 60 techniques for analysing API usages spanning ten years. As a result of this study, among other things came a categorization of API usage patterns (or properties). Very briefly, they are the following:

1) **Unordered usage pattern**: These describe what kind of API elements (classes, methods etc.) co-occur and in what frequency. For example, the usage pattern (`m1`, `m2`) indicates that some client code that calls the API method `m1` will frequently also call method `m2`.
2) **Sequential usage pattern**: These usage patterns also consider the call order of the API operations. For example, the usage pattern `m1` $\rightarrow$ `m2` describes that method `m1` is called before `m2`. A variant to this class of usage patterns, called **multi-object properties**, involves multiple API elements or even multiple API objects at once.
3) **Behavioral specification**: These patterns describe, among other things, contracts on parameters to methods that construct API objects. For example, the key provided to instantiate a specific cryptographic object must not be null or constraints that say an object is an instance of a particular type.

2.2 CRYSL

Each CRYSL rule is defined on the level of an individual Java class and consists of various sections. On a high level, every section in a rule corresponds to a particular type of API misuse violation:

- **OBJECTS** and **EVENTS** gather parameters and methods that can be violated for a particular type. These allow expressing **Unordered usage patterns**.
- **ORDER** describes in what order methods of a particular type must be invoked. This allows one to express **sequential usage patterns**.

Listing 2: CrySL rule for using the JCA class `javax.crypto.KeyGenerator`

```
1  SPEC javax.crypto.KeyGenerator
2
3  OBJECTS
4      int secretKeySize;
5      java.security.spec.AlgorithmParameterSpec
           params;
6      javax.crypto.SecretKey key;
7      java.lang.String secretKeyAlgorithm;
8      java.security.SecureRandom ranGen;
9
10 EVENTS
11     g1: getInstance(secretKeyAlgorithm);
12     g2: getInstance(secretKeyAlgorithm, _);
13     Gets := g1 | g2;
14
15     i1: init(secretKeySize);
16     i2: init(secretKeySize, ranGen);
17     i3: init(params);
18     i4: init(params, ranGen);
19     i5: init(ranGen);
20     Inits := i1 | i2 | i3 | i4 | i5;
21
22     gk: key = generateKey();
23
24 ORDER
25     Gets, Inits?, gk
26
27 CONSTRAINTS
28     secretKeyAlgorithm in {"AES", "HmacSHA224"
           , "HmacSHA256", "HmacSHA384", "
           HmacSHA512"};
29     secretKeyAlgorithm in {"AES"} =>
           secretKeySize in {128, 192, 256};
30
31 REQUIRES
32     randomized[ranGen];
33
34 ENSURES
35     generatedKey[key, secretKeyAlgorithm];
```

- **CONSTRAINTS** describe the allowed range of values for parameters to method calls. This allows one to express **Behavioral specifications**.
- **ENSURES** and **REQUIRES** clauses describe how objects of different types must be correctly composed. By rely/guarantee-reasoning, they allow one to express the **multi-object properties** variant of **Sequential usage patterns**, which span multiple API objects.

Listing 2 shows the CrySL rule of the JCA class `KeyGenerator`, which may be used to generate symmetric keys. The first section is **SPEC**, which defines the full name of the class this particular rule is defined for (Line 1). It is followed by the **OBJECTS** section that defines all objects that are used in the later sections of the rule, either as a parameter or return values (Lines 4–8).

The **EVENTS** section defines all methods that may contribute to the successful usage of the class under specifications as *method event patterns* (Lines 11–22). Each pattern in the **EVENTS** section is of the form `<label: method(parameters)>`. These labels can further be grouped using aggregates of the form `<aggregate := labels>`. The events section further defines named method parameters whose values can be constrained later, in the **CONSTRAINTS** section.

The **ORDER** section defines a regular expression of method-event patterns as a *usage pattern* (Line 25). The methods used in this section are labels/ aggregates defined in the **EVENTS** section. In our example (Line 24), a correct use of a `KeyGenerator` object should contain one call to the methods aggregated as `Gets` followed optionally by a call to `Inits` and finally a call to methods aggregated by `gk`. Any deviation to this sequence would result in an `Incomplete Operation Error`, and a transition to the wrong state would result in an `Incorrect State Error`.

The **CONSTRAINTS** section (Lines 28–29) defines all parameter constraints for parameters defined in the **EVENTS** section. The constraints are specified either as a list or range of values. Any violation of these constraints would result in a `Constraint Error`.

The last section is **ENSURES**, which defines predicates the rule guarantees, provided that all the constraints in the **CONSTRAINTS** section, and the usage pattern defined in the **ORDER** section, are satisfied (Line 35). In turn, the **REQUIRES** section defines those predicates that must be ensured via other CrySL rules by their **ENSURES** section (Line 35). It is those predicates from the **ENSURES** and **REQUIRES** sections that govern the interactions between different classes. Missing any of the required predicates would lead to a `Required Predicate Error`.

A detailed documentation of the CrySL language can be found on its official website.[2]

3 Generating Test Cases From Usage Specifications

In this section, we describe how APITestGen generates test cases out of CrySL specifications. Before we describe our approach in detail, we describe the kind of test cases it may generate.

3.1 Types of test cases

The main goal of APITestGen is to generate test cases for program-analysis tools specifically API misuse detectors; a process currently manually performed. Every test for the misuse detector is a code snippet representing a use of the API followed by an assertion to check whether the right error is thrown (or not) based on whether the code snippet represents a misuse or not. We classify these test cases into atomic and composite depending on the number of object types they compose, each of which in turn could be subcategorized into test cases with a proper use and test cases with a misuse of the API. An overview is shown in Figure 1.

3.1.1 *Atomic test cases*

Atomic test cases use only those methods from the respective CrySL rule whose parameters does not require any predicate to be generated as specified in the **REQUIRES** section.

2. CrySL website: https://www.eclipse.org/cognicrypt/documentation/crysl/

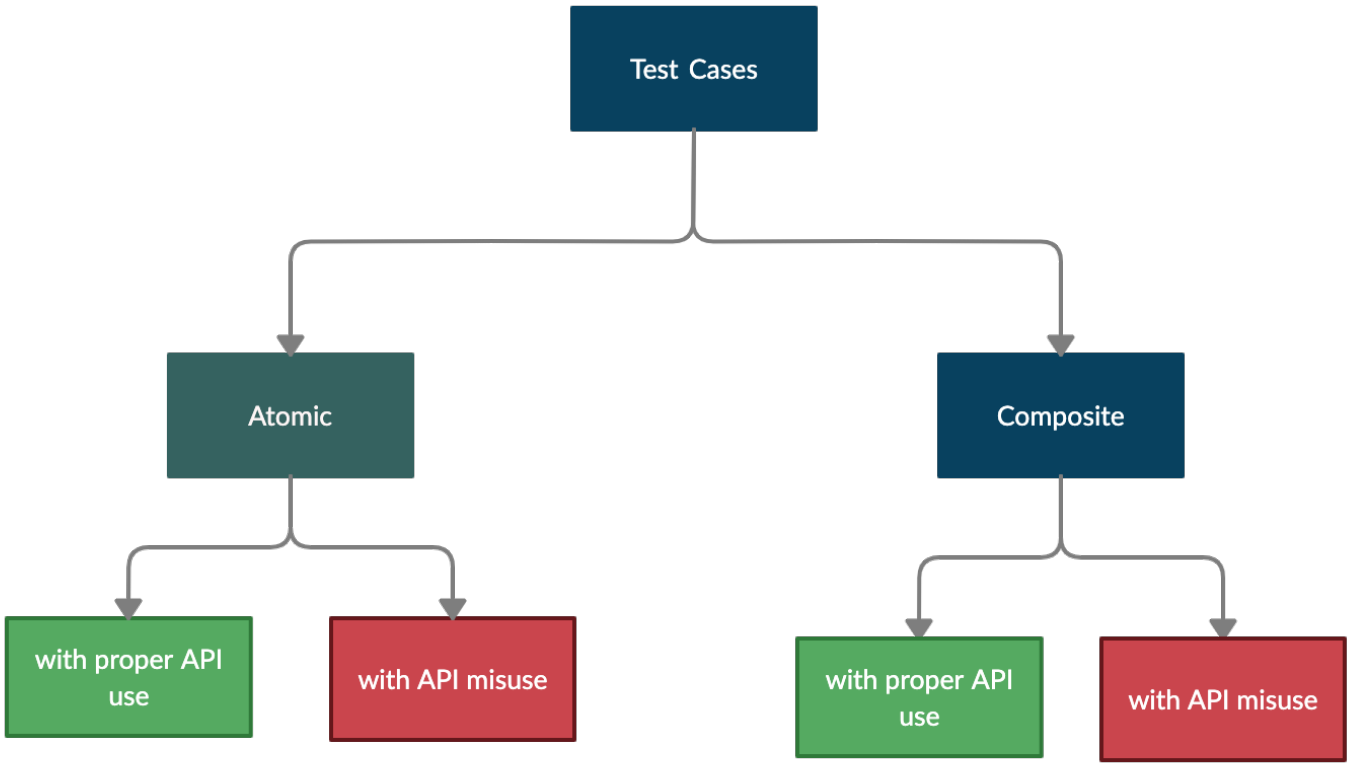


Fig. 1: Types of tests generated by APITESTGEN

1. Atomic test cases with a correct API use: Listing 3 illustrates a test case for the JCA class `KeyGenerator` containing a correct use of the class. An object for the `KeyGenerator` class is created using the `getInstance()` method. The value of the parameter for this method is resolved from its corresponding constraints in the **CONSTRAINTS** section of the `KeyGenerator` rule. After the object creation, the `generateKey()` method is invoked on the object as per the usage pattern defined in the **ORDER** section of the `KeyGenerator` rule. Hence the `KeyGenerator` object is in an accepting state and has the ensured the predicate for the `secretKey` as per Line 35 of the corresponding CRYSL rule in Listing 2. The above test case for class `KeyGenerator` is atomic because it does not require any other JCA classes other than `KeyGenerator`. It is a correct use since it complies with all sections of the `KeyGenerator` rule.

2. Atomic test cases with an API misuse: Listing 4 illustrates a test case representing a misuse of the `KeyGenerator` class: The method `generateKey()` is not invoked on the `KeyGenerator` object. According to the CRYSL rule of `KeyGenerator`, this call is mandatory after the call to `init()`. Hence there is a violation of **ORDER** section. Therefore, the `KeyGenerator` object is not in an accepting state resulting in a test case containing a misuse.

3.1.2 Composite test cases

Composite test cases compose objects from multiple classes. These test cases use methods from the respective CRYSL

Listing 3: An example illustrating a test case with a proper use of the class `java.security.KeyGenerator`

```java
@Test
public void keyGeneratorValidTest() throws
    NoSuchAlgorithmException {
     KeyGenerator keyGenerator0 = KeyGenerator.
         getInstance("AES");
     SecretKey secretKey = keyGenerator0.
         generateKey();
     Assertions.hasEnsuredPredicate(secretKey);
     Assertions.mustBeInAcceptingState(
         keyGenerator0);
}
```

Listing 4: An example illustrating a test case containing a misuse for class `java.security.KeyGenerator`

```java
@Test
public void keyGeneratorInvalidTest() throws
    NoSuchAlgorithmException {
     KeyGenerator keyGenerator0 = KeyGenerator.
         getInstance("AES");
     keyGenerator0.init(128);
     Assertions.mustNotBeInAcceptingState(
         keyGenerator0);
}
```

Listing 5: An example illustrating a test case with a proper use for class `javax.crypto.Cipher`

```
@Test
public void cipherValidTest() throws
    GeneralSecurityException{
    SecureRandom secureRandom0 = new
        SecureRandom();
    Assertions.hasEnsuredPredicate(
        secureRandom0);
    Assertions.mustBeInAcceptingState(
        secureRandom0);

    KeyGenerator keyGenerator0 = KeyGenerator.
        getInstance("AES");
    keyGenerator0.init(128, secureRandom0);
    SecretKey secretKey = keyGenerator0.
        generateKey();
    Assertions.hasEnsuredPredicate(secretKey);
    Assertions.mustBeInAcceptingState(
        keyGenerator0);

    byte[] plainText = null;

    Cipher cipher0 = Cipher.getInstance("AES/
        CBC/PKCS5Padding");
    cipher0.init(1, secretKey);
    byte[] cipherText = cipher0.doFinal(
        plainText);
    Assertions.hasEnsuredPredicate(cipherText)
        ;
    Assertions.mustBeInAcceptingState(cipher0)
        ;
}
```

rule, whose parameters require a predicate to be generated as specified in the **REQUIRES** section.

1. Composite test cases with a correct API use: Listing 5 illustrates a test case for JCA class `Cipher` with a correct use. An object for `Cipher` is created using the `getInstance()` method. The value of the parameter for this method is resolved from corresponding constraints in the **CONSTRAINTS** section of the `Cipher` rule. After the object creation, the `init()` and `doFinal()` methods are invoked on the object as per the usage pattern defined in the **ORDER** section of the `Cipher` rule. Hence the object `cipher` is in the accepting state. The `init()` method requires a `Key` object that is generated by the `KeyGenerator` class. This class depends on the `SecureRandom` object for key initialization. Both of these objects are constructed securely by following their CRYSL rules resulting in their accepting states and ensured predicates. Finally, the `generateKey()` method returns a secure `SecretKey` object that is passed to the `init()` method of the `Cipher`. The above test case for the `Cipher` class is composite because it uses the `KeyGenerator` and the `SecureRandom` classes and is a proper use because all of them comply with their respective CRYSL rules.

2. Composite test cases with an API misuse: These are test cases that are similar to composite test cases containing a proper use but contain a misuse use of the API.

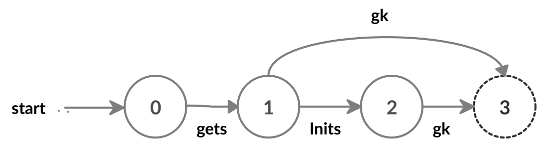


Fig. 2: State machine for the usage pattern of `KeyGenerator` rule

3.2 Test-generation Approach

APITESTGEN may adopt one of several different strategies to generating test cases for an API. The strategies differ in terms of complexity and targeted coverage of the CRYSL rules. We discuss the intricacies of the strategies APITESTGEN supports in Section 3.3. In this section, we focus on how APITESTGEN generates test cases generating a `Select First` strategy, that is, whenever it has multiple viable options to choose from, it deterministically takes the first one.

To construct a test case, APITESTGEN retrieves the *usage pattern* from the CRYSL rule. The usage pattern is a regular expression modeling a Finite State Machine (FSM) defined in the **ORDER** section of the CRYSL rule. Figure 2 shows the usage pattern of the `KeyGenerator` rule (Gets, Inits?, gk) and its corresponding FSM. For each usage pattern, APITESTGEN attempts to generate possible test cases with proper uses and misuses.

3.2.1 Generation of test cases with proper uses of the API

APITESTGEN creates a test case with a proper API use for every complete sequence in the FSM. A complete sequence comprises a list of transition edges starting from an initial state and ending at any of the accepting states in the FSM. Figure 4 shows the complete sequences that are derived from the FSM in Figure 2. Figure 3 shows the process flow for generating proper use test cases.

For each sequence, APITESTGEN first determines the imports necessary for selected methods in the sequence. Subsequently, it computes the predicates associated with the sequence, which is used to select the methods involved in the sequence. APITESTGEN constructs a method call for each method and generates the corresponding method invocations. Into these calls, APITESTGEN resolves parameters in a three-step approach we discuss below. After it has fully constructed all invocations for a particular test case, APITESTGEN generates them along with the test's oracle.

Constructing predicate connections for composite test cases: For creating composite test cases, APITESTGEN needs to understand how the **ENSURES** predicates of one CRYSL rule connect to **REQUIRES** predicate of another CRYSL rule. To this end, APITESTGEN first determines the predicates that should be ensured for each transition. For each transition in each valid sequence, APITESTGEN retrieves all predicates from the current CRYSL rule. During this step, APITESTGEN takes note of such conditions, the corresponding method calls, and their predicates. **ENSURES** predicates

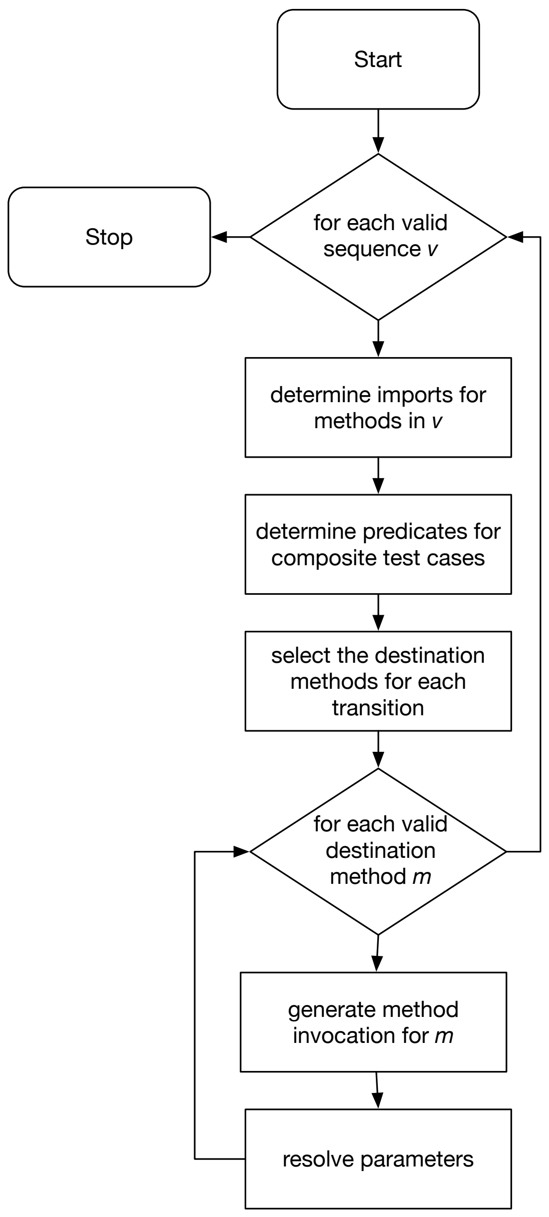

Start
Stop
for each valid sequence *v*
determine imports for methods in *v*
determine predicates for composite test cases
select the destination methods for each transition
for each valid destination method *m*
generate method invocation for *m*
resolve parameters

Fig. 3: Generation process for tests with proper API use

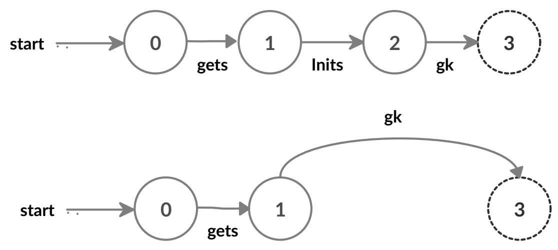


Fig. 4: All *valid* sequences for the state machine in Figure 2

can be conditional, that is, a specification can indicate that a predicate can be ensured only upon invocation of a certain method call. The noted conditions and predicates are used to make a decision on which methods can be connected using specific transition edges.

Selecting test methods from FSM transitions: Each transition is composed of transition edges. APITESTGEN selects a method for each transition edge such that the resulting method sequence ensures the predicate returned by the previous step. To summarize, this step returns the first possible method for the edge such that all required predicates are ensured.

The previous two steps that determine the predicate relationships result in a Rule Dependence Tree. Every node in the tree represents a CRYSL rule, and every edge connecting two nodes indicates the dependency between them. All edges are directed, which shows the direction of dependency. Figure 6 shows, for instance, a dependence of `Cipher` to `KeyGenerator` and another one from `KeyGenerator SecureRandom`. Figure 5 shows the complexity of the Rule Dependency Tree (RDT) for the JCA ruleset.

Constructing method invocations: After completing the previous two steps, APITESTGEN constructs the invocations for all methods to be part of the test case. We encountered two issues that cause this step to not be as straightforward as we initially expected. First, APITESTGEN needs to determine if a method is a constructor, static method or instance method since such information are not preserved inside CRYSL rules. Currently, APITESTGEN uses a name-based heuristics to ascertain the type of method. Methods named `getInstance()` are classified as a static method, and their invocation is generated using only the class name without the **`new`** keyword. When a method name corresponds to its class name, APITESTGEN classifies it as a constructor, and generates its invocation using the class name along with the **`new`** keyword. APITESTGEN classifies the remaining methods as instance methods and generates them using the instance object.

The second problem we encountered revolves around exceptions. Since CRYSL rules do not specify any exceptions, APITESTGEN retrieves this information through other means . Using Java reflection, APITESTGEN collects all constructors and methods of the class to which the method belongs and fetches exceptions thrown by the method or constructor. It finally annotates the test case as throwing these exceptions as well.

Resolving parameters: APITESTGEN needs to generate valid parameter combinations for the constructed method invocations from previous step in a way that there is no violation to the **`CONSTRAINTS`** section of the CRYSL rule. For each parameter, APITESTGEN checks whether it requires a predicate. If that is the case, APITESTGEN gets the rule that ensures it from the RDT populated previously. Subsequently, APITESTGEN checks if another method invocation in the test code already ensure this predicate. In this case, APITESTGEN inserts this invocation's base object as the parameter. Otherwise, it puts the code generation for the current rule on hold and instead triggers it recursively on the rule that is ensuring the predicate. Once APITESTGEN completes generating this auxiliary code, it finally places

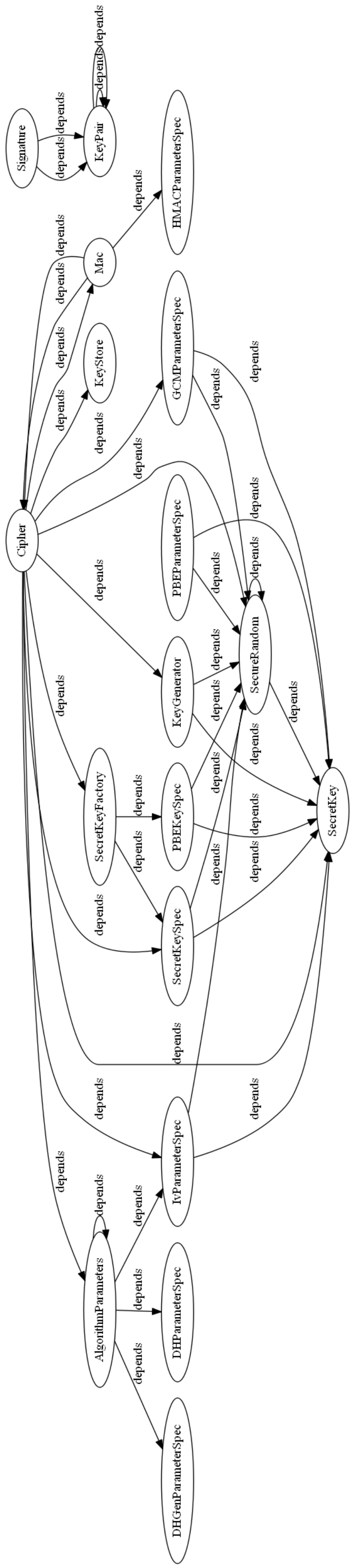


Fig. 5: Rule dependency tree for JCA rules

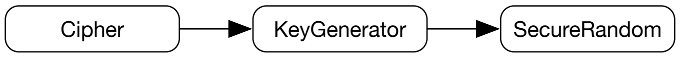


Fig. 6: Rule dependency for Cipher

the helper object as the parameter. As part of resolving parameters, APITESTGEN needs to determine appropriate combination of parameters for each of the three kinds of constraints:

Value Constraint: A value constraint is of the form `variable in value 1, value 2, value 3,...,` `value N`, where `variable` contains any of the values from `value 1` to `value N`.

Comparison Constraint: A comparison constraint is of the form `variable operator value` where the operator is a comparison operator such as $<, \leq, >, \geq, \neq, =$.

Predicate: Predicate in the **CONSTRAINTS** section is similar to a function that returns a value based on its parameters. Some of the predicates supported in the **CONSTRAINTS** section are `instanceOf, callTo, noCallTo, neverTypeOf, length, notHardCoded`.

Constraints can also be combined using logical operators such as $\&\&, ||, \Rightarrow$. APITESTGEN uses a recursive strategy to resolve a combination of constraints. It fetches the rightmost operator in the constraint and fetches its left and right operands. Such operands can further be a combination of constraints. Depending on the type of logical operator used for combining the constraints, it traverses down the operator tree until it reaches a simple constraint. Based on the type of simple constraint, the algorithm resolves the parameter. If the constraint is of type `Value Constraint`, our approach usually returns the first value from the set of possible values. In the exceptional case for a parameter such as transformation in the `Cipher` rule, APITESTGEN appends the mode and padding to the selected algorithm before returning it. If the constraint is of type `Comparison Constraint`, it retrieves its left and right operands. APITESTGEN generates a secure random number based on the type of constraint and the numerical value in one of these operands as described in Algorithm 1. If the constraint is of type `Predicate`, APITESTGEN fetches the first parameter. It instead selects an *empty value* if the constraint type is unsupported or some operator is invalid. After resolving the parameter, APITESTGEN replaces it with the resolved value in the method invocation. After replacing the value, the test generator continues with the next parameter of the method invocation.

3.2.2 Generation of test cases with API misuses

APITESTGEN creates a test case with a misuse for every incomplete sequence in the FSM. An incomplete sequence can either be a transition with a list of transition edges starting from an initial state but not ending at any of the accepting states in the FSM or it can be a complete sequence with a missing intermediate state. Figure 7 shows the incomplete sequences that are derived from the FSM in Figure 2. It shows only incomplete sequences that end at a non-accepting state. `Gets` is either a constructor call or static factory method and hence cannot be skipped. `Inits` is an optional transition, and skipping it still results in a complete sequence. Hence for FSM in Figure 2, there cannot be any incomplete sequences with missing the intermediate state. Once we have the incomplete sequences, the process of selecting methods, generating method invocation for each of the methods, and resolution of their parameters remain the same as shown in Figure 3, but with a misuse. In our example, since three incomplete sequences are derived, the test generator creates three test cases with misuses. For generation of these misuse test cases based on constraints, APITESTGEN selects random values for parameters based on their type and those that do not comply with the listed constraint range.

Algorithm 1 Algorithm to resolve parameter using its constraints

```
 1: operator ← constraint.getOperator()
 2: leftOperand ← constraint.getLeft()
 3: rightOperand ← constraint.getRight()

 4: if leftOperand is number then
 5:     value ← leftOperand
 6: else if rightOperand is number then
 7:     value ← rightOperand

 8: if operator = $>$ OR $\geq$ then
 9:     return random number greater than or equal to value
10: else if operator = $<$ OR $\leq$ then
11:     return random number lesser than or equal to value
12: else if operator = $\neq$ then
13:     return random number not equal to value
14: else
15:     return value
```

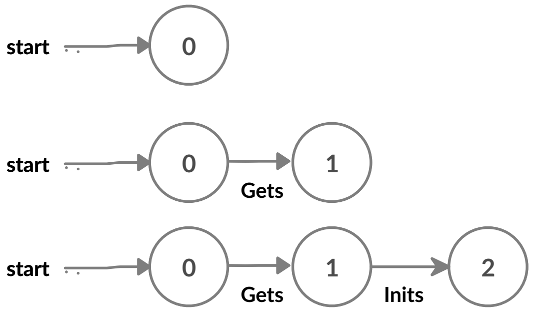


Fig. 7: Incomplete sequences for the state machine in Figure 2

3.3 Generation based on different strategies

As mentioned above, we have devised three test-generation strategies for APITESTGEN: `Select First`, `Select Random` and `Select All`. The goal of APITESTGEN is to generate *usage pattern tests* that exercise different transitions of the FSM defined in the **ORDER** section and cover a comprehensive range of constraints in the **CONSTRAINTS** section. In the following, we discuss our test-generation strategies from the perspective of transitions. They treat constraints accordingly. Each transition consists of one or more transition edges. When following the `Select First` strategy, APITESTGEN selects the first method from the list of methods for a given FSM edge. Using the `Select`

Transition Type	Transition	Select First	Select Random	Select All
Proper use	Gets, Inits, gk	g1, i1, gk	g*, i1, gk g1, i*. gk	g1, i1, gk g2, i1, gk g1, i2, gk g1, i3, gk g1, i4, gk g1, i5, gk
	Gets, gk	g1, gk	g*, i1, gk g1, i*, gk	g1, gk g2, gk
Misuse	Gets, Inits	g1, i1	g*, i1 g1, i*	g1, i1 g2, i1 g1, i2 g1, i3 g1, i4 g1, i5
	Gets	g1	g*	g1 g2

TABLE 1: Strategy based transitions for the usage pattern in Figure 2

`Random` strategy, our approach selects a method randomly. Lastly, APITESTGEN generates one test per method when the `Select All` strategy is enabled. Table 1 describes transitions obtained for the usage pattern in the `KeyGenerator` rule using different test generation strategies. The first column represents whether the transition described is for a sequence with a proper use of the API or an API misuse, the second column describes the possible transition based on whether they are for a proper use or misuse, the next three columns indicate the results for the transition and the type the usage patterns generated for each strategy. The `Select First` strategy allows the test-case developer to generate the bare minimum amount of tests to cover all rules specified in the CRYSL rule set. This is beneficial in a scenario like regression testing and during scenarios where runtime and space consumption are of concern. `Select Random` is beneficial when a better coverage is desired. When one aims for a comprehensive test set after making major changes to the back-end static analysis, one should opt for the `Select All` strategy.

4 EVALUATION

We evaluate APITESTGEN along the following research questions:

1) **RQ1 (Improvements to manual process)**: Is APITESTGEN able to generate test cases that improve the current manual process in terms of rule coverage, amount of tests and time taken?
   ⋆ **Stability**: How stable is APITESTGEN when considering an evolving CRYSL ruleset?
2) **RQ2 (Contributions to correctness of static analysis tool and specifications)**: What issues with the static analysis tools and specifications did APITESTGEN expose?

4.1 Setup

We generate test cases for JCA. Among Java cryptographic libraries JCA is the most frequently used one and is shipped with the Java Development Kit (JDK) [2]. Moreover, the JCA CRYSL ruleset maintained by COGNICRYPT$_{SAST}$ comes with manually defined test cases, which we use for comparison. Our experimental setup is available online.[3]

4.2 Improvements to manual process (RQ1)

To answer the first research question, we compare the overall number of generated tests with the number of tests that contributors to COGNICRYPT$_{SAST}$ have manually written over the years as well as the respective numbers of covered JCA classes/ object types. Next, we analyse the quality of the generated tests: we say that a generated test is *agreeing*, if its oracle is in agreement with the classification by COGNICRYPT$_{SAST}$ of the usage encoded by the test. We analyse disagreeing test cases manually.

4.2.1 Overview of the results.

The numbers of generated and previously existing manual test cases are depicted on the left-hand side of Figure 8 – red is used for disagreeing test cases, green for agreeing ones and grey for disagreeing ones, which turned out to be accurate tests that exposed bugs with COGNICRYPT$_{SAST}$. APITESTGEN generated 98 test cases with the "`Select First`" strategy – 50 with a correct API use and 48 with an API misuse. Using the "`Select All`" strategy, APITESTGEN generated 370 test cases – 191 with a correct API use and 179 with an API misuse. In comparison, over the past three years, 16 contributors to COGNICRYPT$_{SAST}$ have written 71 test cases.

From the perspective of class coverage (Figure 8, right-hand side), manual tests cover only seven classes whereas APITESTGEN generated test cases for 31 out of the 38 classes for which crysl rules are available. The classes that are not covered by APITESTGEN are for interfaces or abstract classes like `SSLEngine` or `SecretKey`. In theory, it is possible to instantiate chosen subclasses of interfaces and generate tests using the same. But currently, APITESTGEN does not make any assumptions on which of the subclasses/ implementations of an interface/ abstract classes need to be chosen to be instantiated and therefore ignores these rules. Out of 98 test cases generated with `Select First` 81 (82.6%) agree. Out of 370 test cases generated with `Select All` 309 (83.3%) agree.

In summary, APITESTGEN was able to generate hundreds of more tests and was able to cover more than four times the number of rules. It does so in matter of seconds. APITESTGEN takes 18 seconds to generate test cases with the `Select First` strategy and 45 seconds for the `Select All` strategy. Such a difference in runtime can be attributed to the number of test cases generated with the `Select All` strategy, which is thrice as many as the test cases generated with the `Select First` strategy. APITESTGEN consumes around 507 MB for generating tests using the `Select First` strategy and 522 MB for generating tests using the `Select All` strategy. However, these readings also include the underlying Eclipse process that is notorious for being heavy-weight in terms of memory consumption[4].

3. The CRYSL ruleset used for our experiments is available in the `testgen-develop-ruleset` branch at https://github.com/CROSSINGTUD/Crypto-API-Rules, the version of APITESTGEN used for evaluation purpose is open-sourced at https://github.com/CROSSINGTUD/CogniCrypt_TESTGEN.
4. https://stackoverflow.com/questions/1490803/how-to-reduce-eclipses-memory-usage

We further examine the generated test cases with disagreeing oracles. Since `Select all` subsumes `Select first`, we analyse the disagreeing tests from the `Select All` strategy. There are 61 disagreeing tests overall. We identify four categories. Figure 9 gives an overview of the reasons for the disagreements.

The yellow category consists of cases, where the disagreement revealed bugs of CogniCrypt$_{SAST}$ (25), including some incorrect specifications. We have reported these issues to the maintainers of CogniCrypt$_{SAST}$ and elaborate on them in Section 4.3. The other three categories (**blue**, **red**, **gray**) reveal current limitations of APITestGen, which we discuss next.

4.2.2 *Incorrectly generated test cases*

We remind the readers that in CrySL, the **REQUIRES** and **ENSURES** predicates are used to define how objects of multiple types can be properly composed. These predicates can also be parametrized. Consider the following snippet of the rule for the class `Cipher`

```
REQUIRES
    generatedKey[key, alg(transformation)];
    randomized[ranGen];
```

APITestGen does not support generation of assertions of whether predicates are ensured or not if the parameters involve an object of a non-primitive type. This issue is responsible for 2 out of 61 test cases generated by APITestGen that disagree with CogniCrypt$_{SAST}$. Another issue is that APITestGen currently does not support two step generation which is responsible for 17 out of 61 test cases. To understand the need for two-step generation, consider the following listing with a snippet of the rule for the class `Cipher`.

```
EVENTS
SPEC javax.crypto.Cipher

OBJECTS
    java.lang.String transformation;
    java.security.Key key;
    ...

EVENTS
    g1: getInstance(transformation);
    g2: getInstance(transformation, _);
    ...
CONSTRAINTS
    instanceOf[key, javax.crypto.SecretKey] =>
         alg(transformation) in {"AES", "
        PBEWithHmacSHA224AndAES_128", ...}
    ....
```

The constraint is a conditional constraint, meaning that if the type `key` is `javax.crypto.SecretKey`, then the transformation algorithm must be one of the options specified. But, the type of `key` can be only deduced after parsing the `init` method and therefore would require a first step generation of the `init` method and another generation step to deduce the type of `key`. One step generation may result in incorrect parameters for the `getInstance` method.

Finally, APITestGen behaves buggy when dealing with specifications that use the pipe(choice) operator in the presence of nested finite state machines which is responsible for 17 out of the 61 disagreeing test cases. An example of a rule that generates buggy test cases involves the **ORDER** section of `Keystore` rule

```
ORDER
    Gets, Loads, ((GetEntry?, GetKey) | (
        SetEntry, Stores))*
```

All mentioned current limitations can be easily addressed given sufficient engineering capacities.

We conclude that APITestGen generates a high number of agreeing test cases (83%). In 25 instances, the generated tests revealed bugs in CogniCrypt$_{SAST}$. This implies that in total (309 + 25), tests generated by APITestGen are accurate in 90.2% cases, implying they are of high quality. APITestGen also generates some inaccurate tests, but these small portion of tests can be fixed with effort that is negligible in comparison to writing hundreds of tests by hand.

4.2.3 *Stability*

Even though APITestGen does not currently support incremental generation of tests based on differences in specifications, we wanted to ensure stability of APITestGen across different versions of the CrySL ruleset. For this reason, we evaluated it with a more recent version of the ruleset of the JCA which is made available in the `testgen-eval-ruleset` branch of the repo maintained by CogniCrypt$_{SAST}$ for the CrySL specifications[5]. It contains all the changes added in the past year to the old ruleset. In addition to 39 specifications in the old ruleset, 8 new specifications are available in the newer ruleset. APITestGen was able to generate test cases for all classes from the latest ruleset modulo interfaces, abstract classes and classes that depend on an object of an interface type. It generated a total of 121 test cases with the `Select First` strategy and 417 test cases with the `Select All` strategy in 29 and 47 seconds respectively.

APITestGen is able to adapt to an evolving rule set and generate more than a hundred new tests for the later rule-set with considerable ease in a matter of seconds.

4.3 Contributions to correctness of static analysis tool and specifications (RQ2)

Recall that the ultimate goal of APITestGen is to facilitate testing the correctness of API misuse detectors. In the following, we report the issues that the application of generated test cases revealed with CogniCrypt$_{SAST}$ and with the specifications themselves. These issues were detected by studying disagreeing cases, where the generated test was alright, but either the classification by CogniCrypt$_{SAST}$ was wrong, or the specification was buggy.

5. https://github.com/CROSSINGTUD/Crypto-API-Rules/tree/testgen-eval-ruleset

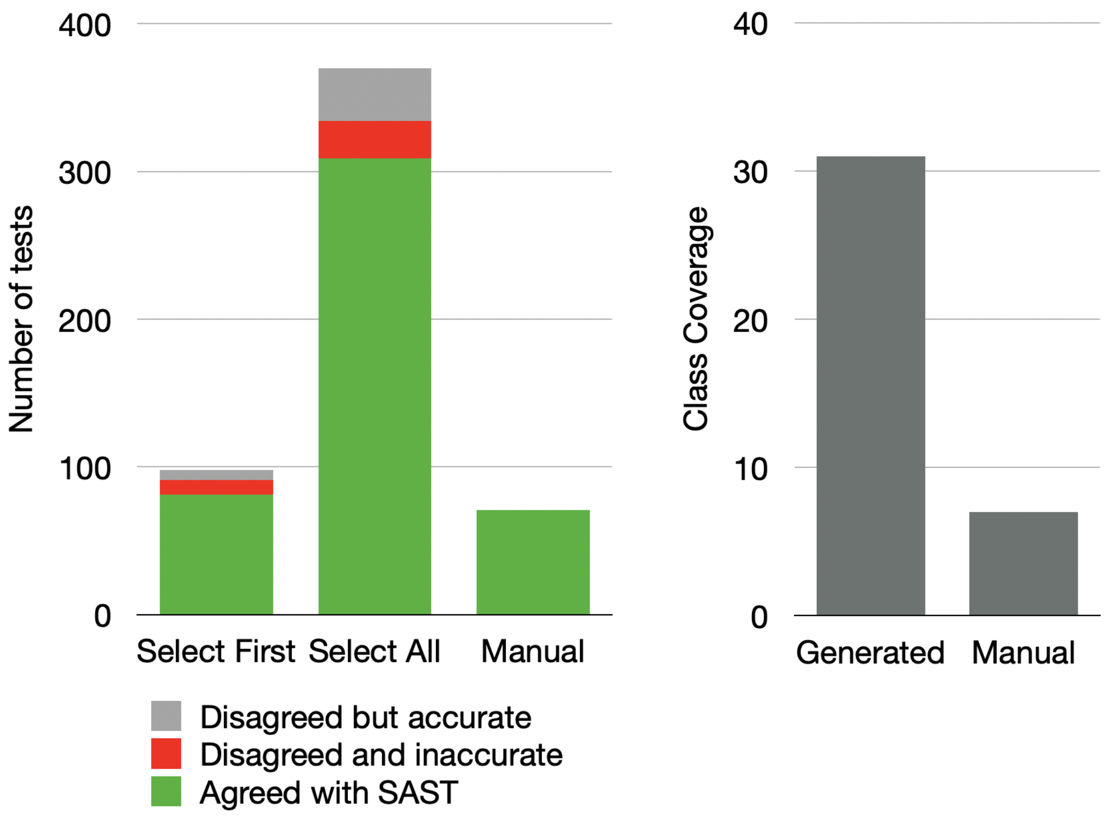


Fig. 8: Results of running COGNICRYPT$_{SAST}$ on test cases

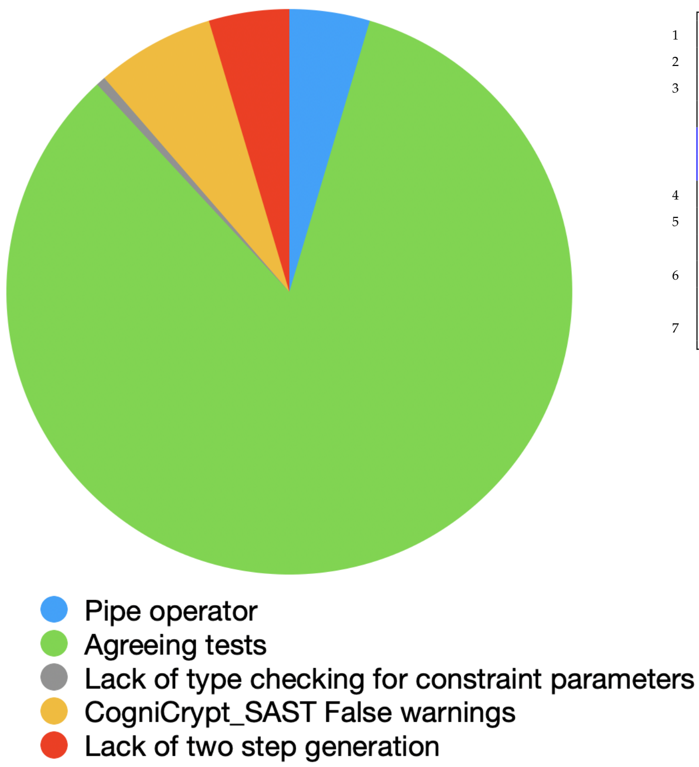


Fig. 9: Reasons for failure of tests generated by APITESTGEN under the `Select All` strategy

Listing 6: A false positive detected by APITESTGEN.

```
@Test
public void sSLParametersValidTest1() {
    SSLParameters sSLParameters0 = new
        SSLParameters(new String[] { "
        TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
        " },
        new String[] { "TLSv1.2" });
    Assertions.hasEnsuredPredicate(
        sSLParameters0); // FAILS
    Assertions.mustBeInAcceptingState(
        sSLParameters0); // FAILS
}
```

4.3.0.1 Correctly generated tests disagreeing with COGNICRYPT$_{SAST}$: Listing 6 reveals one such test case for `SSLParameters` wrongly flagged by COGNICRYPT$_{SAST}$ because it cannot handle multiple start symbols for the **ORDER** section's FSM. We have an issue was reported for the cases where COGNICRYPT$_{SAST}$ wrongly flagged correct tests as failed[6].

4.3.0.2 Issues with CRYSL specifications.: APITESTGEN also exposed issues with incorrect CRYSL specifications. First, some CRYSL specifications contain protected methods in their **EVENTS** section. For example, `SecureRandom`'s `next` method is protected, and one cannot use this method unless their implementation class also

6. https://github.com/CROSSINGTUD/CryptoAnalysis/issues/296

inherits the class or any of its ancestors[7] which as of writing this paper is reviewed and accepted.

Second, we observed duplicate rules in the specifications. For instance, `getInstance(String algorithm)` was defined twice in the `Mac` rule instead of its other variants `getInstance(String algorithm, Provider provider)` and `getInstance(String algorithm, String provider)`.

We have corrected the affected specification and contributed a pull request[8] which as of writing this paper is reviewed and accepted.

The reported issues provide further evidence that APITESTGEN generated tests help to validate not only the correctness of the underlying static analysis but also correctness of the rules themselves.

5 Related Work

We discuss related work along the following sub-categories:

5.1 Test generation from DSLs

Although there are not many DSLs as expressive as CRYSL, especially for API usage specification, there are a few languages allowing specification of APIs in some form, and test case generation approaches have utilized these languages. For example, Oriat et al. [12] developed a tool to generate unit tests for Java classes using the Java Modeling Language (JML). JML allows API designers to specify annotations to their java comments to specify design contracts (pre- and post-conditions). Our approach generates test cases that verify correctness of the static analysis tools that check for API misuses, while their approach unit tests that verify if implementation of the API behaves as expected. Jartege [13] applies random testing to generate test cases also from JML. Their tool does generate sequences of calls, which could be considered inputs to test static analysis tools but they are randomly generated and require careful controlling of parameters to ensure correctness. But since our approach relies on a specification language that explicitly defines usage patterns, we avoid many bogus test cases. Although not strictly a specification language, Törsel et al. [14] discuss a model-based testing approach for black-box testing of web applications. The model is not a specification of an API, but rather the structure of the web application and its navigational connections. Their test cases only test the front-end of web applications. Douibi et al. [15] developed a specification-based test generation from OpenAPI specifications for REST APIs. OpenAPI specifications' scope is limited expressing how APIs can be produced and consumed. Prasanna et al. [16], Ranjita et al. [17] and many other works [18, 19, 20] have proposed generation of test cases from UML Diagrams. Although these works generate test cases from specifications, their approaches test either the implementation of the API or the target application and not the static analysis tools.

7. https://github.com/CROSSINGTUD/Crypto-API-Rules/pull/81
8. https://github.com/CROSSINGTUD/Crypto-API-Rules/pull/80

5.2 Test generation from source code

Although specification languages are more expressive sources of knowledge about how to use an API, some approaches have also attempted to generate test cases directly from source code. Mahadik et al. [21] have employed a search-based test data generation technique to reach high code coverage in object-oriented code. The system takes multiple Java class files as input. It generates instances of those classes, followed by a sequence of method calls. It uses an evolutionary strategy to provide complete code coverage. Evosuite [22] automatically generates test cases with high code coverage for Java classes. To do this, it uses an evolutionary search approach that generates and evolves whole test suites. Evosuite has accomplished close to full coverage of the target program. APITESTGEN's *Select All* strategy can serve as an input for approaches like Evosuite.

5.3 Test generation from FSM

ConData [23] generates test cases from finite state machines specified using Protocol specification language (PSL). Bochmann et al. [24] use SDL specification, which has an underlying model of EFSM, to derive tests for conformance testing with high fault coverage. Andrews et al. [25] have described an approach of building hierarchies of FSMs that model subsystems of the Web applications. TAG [26] can automatically generate test cases for deterministic, partially, or completely specified FSM. It derives test cases based on the transition identification approach and aims for transition coverage. A state identification technique such as harmonized state identification (HSI) sets are used to verify the tail states of the transitions for fault coverage. However, their approach cannot generate test cases for non-deterministic FSM. Petrenko et al. [27] generate tests for both deterministic or non-deterministic, partially or completely specified FSM that may contain indistinguishable states using State-Counting approach. Although all of these approaches generate test cases from FSM just like part of our approach, our scope is broader and relies on more input than just the FSM.

6 Conclusion

Program analyses are widely used to detect API misuses. However, such analyses are difficult to get sound and precise. As a consequence, it is advisable to thoroughly test them, resulting in significant manual effort in creating and maintaining test suites.

In this paper, to automate this process, we have presented APITESTGEN, a specification-based approach to generate test cases for API-misuse detectors. APITESTGEN leverages API-usage specifications in the specification language CRYSL to construct and generate sample usages of the API along with test oracles. We have empirically evaluated APITESTGEN by generating test cases for the misuse detector COGNICRYPT$_{SAST}$ and found them to be effective at revealing bugs within COGNICRYPT$_{SAST}$.

More broadly speaking, the comprehensive test-case sets APITESTGEN generates may be used both to verify the correctness of a static or dynamic analysis that detects misuses with respect to these specifications, as well as a data-set of correct and incorrect usage templates that can be used as a benchmark for future studies or to foster the understanding of API usage patterns.

Although our approach relies on CRYSL and we evaluate it with rules for JCA and the misuse detector COGNICRYPT$_{SAST}$, our contributions generalize to other APIs and other API misuse detectors for other domains. This is because unlike the CRYSL JCA rule set, CRYSL itself is not specific to the crypto domain. As our comparison of CRYSL with other API specification DSLs reveals, its design has been rather guided by findings on API usage properties inferred by earlier research [28].

Within the crypto domain, the tests generated by APITESTGEN for the rules for the JCA for the study in this paper can be re-used as a benchmark to test other crypto misuse detectors [3]. Given CRYSL rules for other APIs from the crypto or other domains, APITESTGEN can generate test usages for those APIs, too. This way, the generated tests can complement the manually crafted MuBench benchmark for API misuse detetion [29].

REFERENCES

[1] G. Uddin and M. P. Robillard, "How api documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.

[2] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops: Why do java developers struggle with cryptography apis?" in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 935–946. [Online]. Available: https://doi.org/10.1145/2884781.2884790

[3] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. Yao, "Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects," 11 2019, pp. 2455–2472.

[4] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," 11 2013, pp. 73–84.

[5] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and R. Kamath, "Cognicrypt: Supporting developers in using cryptography," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 931–936. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155681

[6] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2019.

[7] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 672–681.

[8] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated api property inference techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2013.

[9] M. Hazhirpasand, M. Ghafari, S. Krüger, E. Bodden, and O. Nierstrasz, "The impact of developer experience in using java cryptography," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–6.

[10] M. Hazhirpasand, M. Ghafari, and O. Nierstrasz, "Cryptoexplorer: An interactive web platform supporting secure use of cryptography apis," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 632–636.

[11] Z. Xu, X. Hu, Y. Tao, and S. Qin, "Analyzing cryptographic api usages for android applications using hmm and n-gram," pp. 153–160, 2020.

[12] C. Oriat, "Jartege: A tool for random generation of unit tests for java classes," 01 2005.

[13] G. Leavens, A. Baker, and C. Ruby, "Jml: A notation for detailed design," 02 1970.

[14] A.-M. Törsel, "Automated test case generation for web applications from a domain specific model," 07 2011, pp. 137–142.

[15] H. Ed-douibi, J. Canovas Izquierdo, and J. Cabot, "Automatic generation of test cases for rest apis: A specification-based approach," 10 2018, pp. 181–190.

[16] M. Prasanna, "Automatic test case generation for uml class diagram using data flow approach."

[17] R. K. Swain, P. K. Behera, and D. P. Mohapatra, "Generation and optimization of test cases for object-oriented software using state chart diagram," 2012.

[18] V. Santiago Júnior, N. Vijaykumar, D. Guimarães, A. Amaral, and E. Souza, "An environment for automated test case generation from statechart-based and finite state machine-based behavioral models," 05 2008, pp. 63 – 72.

[19] S. Kansomkeat and W. Rivepiboon, "Automated-generating test case using uml statechart diagrams," 09 2003, pp. 296–300.

[20] R. Swain, P. Behera, and D. Mohapatra, "Minimal test-case generation for object-oriented software with state charts," *International Journal of Software Engineering & Applications*, vol. 3, 08 2012.

[21] P. Mahadik and D. Thakore, "Search-based junit test case generation of code using object instances and genetic algorithm," *International Journal of Software Engineering and Its Applications*, vol. 10, pp. 95–108, 05 2016.

[22] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," 09 2011, pp. 416–419.

[23] E. Martins, S. Sabiao, and A. Ambrosio, "Condata: A tool for automating specification-based test case generation for communication systems," vol. 8, 01 2000, p. 10 pp. vol.1.

[24] G. Bochmann, A. Petrenko, O. Bellal, and S. Maguiraga, "Automating the process of test derivation from sdl specifications." 01 1997, pp. 261–276.

[25] A. Andrews, J. Offutt, and R. Alexander, "Testing web applications by modeling with fsms," *Software and System Modeling*, vol. 4, pp. 326–345, 07 2005.

[26] Q. Tan, A. Petrenko, and G. Bochmann, "A test generation tool for specifications in the form of state machines," 03 1996.

[27] A. Petrenko and N. Yevtushenko, "Testing from partial deterministic fsm specifications," *Computers, IEEE Transactions on*, vol. 54, pp. 1154– 1165, 10 2005.

[28] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated api property inference techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2013.

[29] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "Mubench: A benchmark for api-misuse detectors," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 464–467.