On the Coupling between Vulnerabilities and LLM-generated Mutants: A Study on Vul4J dataset

Aayush Garg
University of Luxembourg
aayush.garg@uni.lu

Renzo Degiovanni
University of Luxembourg
renzo.degiovanni@uni.lu

Mike Papadakis
University of Luxembourg
michail.papadakis@uni.lu

Yves Le Traon
University of Luxembourg
yves.letraon@uni.lu

***Abstract*—With the release of powerful language models trained on large code corpus (e.g., CodeBERT, trained on 6.4 million programs), a new family of mutation testing tools has arisen that promises to generate more "natural" mutants, where the mutated code aims at following the implicit rules and coding conventions produced by programmers.**

In this paper, we empirically study the observable behavior of CodeBERT-generated mutants and to what extent are these coupled with software vulnerabilities. To do so, we carefully analyze 45 reproducible vulnerabilities from the Vul4J dataset to determine whether the mutants and vulnerabilities fail the same tests and whether the failures are for the same reasons or not. Hence, we define different degrees of vulnerability-coupling classes. *Strongly coupled* mutants fail the same tests for the same reasons as the vulnerabilities, while *test coupled* mutants fail the same tests but for some different reason as the vulnerabilities. Partial coupling classes are also considered.

Overall, CodeBERT-generated mutants strongly coupled with 32 out of these 45 vulnerabilities (i.e. the mutants fail on the same tests for the same reasons), while another 7 vulnerabilities are test-coupled by CodeBERT mutants (i.e. the mutants fail on the same tests but not for the same reasons). Interestingly, CodeBERT mutants are diverse enough to couple vulnerabilities from 14 out of the 15 types of vulnerabilities explored, i.e., CWEs (Common Weakness Enumeration). Finally, we observe that strongly coupled mutants are scarce (1.17% of the killable mutants), test coupled mutants represent 7.2%, and 64.9% of the killable mutants are not coupled with the vulnerabilities.

I. Introduction

Research and practice with mutation testing have shown that it is one of the most powerful testing techniques [2], [18], [23], [41]. Apart from testing the software in general, mutation testing has been proven to be useful in supporting many software engineering activities which include improving test suite strength [1], [9], selecting quality software specifications [30], [31], [44], among others. However, its use in tackling software security issues has received little attention. A few works focused on model-based testing [6], [32] and proposed security-specific mutation operators to inject potential security-specific leaks into models that can lead to test cases capable of finding attack traces in internet protocol implementations. Other works proposed new security-specific mutation operators that aim to mimic common security bug patterns in Java [29] and C [26], [34]. These works empirically showed that traditional mutation operators are unlikely to exercise security-related aspects of the applications. Hence, they proposed operators that attempt to convert non-vulnerable code to vulnerable by mimicking common real-world security bugs. However, pattern-based approaches have two major limitations. On one hand, designing security-specific mutation operators is not a trivial task since it requires manual analysis and comprehension of the vulnerability classes that cannot be easily expanded to the extensive set of realistic vulnerability types (e.g. they restrict to memory [29] and web application [34] bugs). On the other hand, these mutation operators can alter the program semantics which may not be convincing for developers as they may perceive them as unrealistic/uninteresting [4], hence obstructing their usability.

In the literature, language models have been explored to accomplish code completion [28], test oracle generation [45], program repair [10], among other software engineering tasks. With the aim of producing more realistic and natural code, a new family of tools based on language models has recently arisen. Language models are being used for mutant generation yielding to several mutation testing tools such as SemSeed [40] and DeepMutation [46]. While these tools are subjected to expensive training on datasets containing thousands of buggy code examples, there is an increasing interest in using pre-trained language models for mutant generation [3], [17], [42]. The mutation testing tool *μBERT* [17] is one such example that uses CodeBERT [20] to generate mutants by masking and replacing tokens with CodeBERT's suggested replacements.

Since pre-trained language models were trained on large code corpus (e.g. CodeBERT was trained on more than 6.4 million programs), their predictions are typically considered representative of the code produced by the programmers. Hence, we wonder:
Are mutants generated by pre-trained language models coupled with software vulnerabilities?

A positive answer to this question can be promising for the use of such mutants to form an initial step towards defining vulnerability-focused testing requirements. We believe that these requirements are particularly useful when building regression test suites for security-intensive applications.

To answer this question we conduct a controlled experiment on the Vul4J dataset [8] of 45 reproducible vulnerabilities with severity ranging from medium to high. For each vulnerability, Vul4J provides the corresponding vulnerable and fixed (non-vulnerable) files, and the test suites with the Proof of Vulnerability (PoV), that is, a set of test cases for which some of them fail in the vulnerable version (i.e. trigger the vulnerability) but pass on the fixed one. Then, we run the PoV test suites on the mutants and vulnerabilities and analyze to what extent these behave similarly. To do so, we follow a similar procedure as Gay and Salahirad [24] to study the coupling between mutants and real-faults and perform a *manual* analysis to check whether the mutants and vulnerabilities break the same or different tests and if they fail for the same reasons or not.

Hence, we define different vulnerability-coupling classes between mutants and vulnerabilities measured in terms of the number and reasons for failing tests (PoVs). A mutant is said *strongly coupled* with a vulnerability if it fails on the same PoV tests and for the same reasons as vulnerabilities (i.e. same observable exceptions and error messages are thrown). A mutant is said *test coupled* with a vulnerability if it fails on the same PoV tests but at least one of the failures is for some different reason. Mutants that fail on a subset of the PoV tests for the same or different reasons as the vulnerability are said *partially coupled* or *partially test coupled*, respectively, to the vulnerability.

Overall, we found that for 39 out of 45 studied vulnerabilities (i.e., 87.7%) there exist mutants that are somehow coupled with vulnerabilities, i.e., such mutants fail one or more tests that are failed by the respective vulnerabilities irrespective of the reasons. More precisely, μ*BERT*-generated mutants strongly coupled with 32 out of these 39 vulnerabilities (i.e. the mutants fail on the same tests for the same reasons), while for the remaining 7 μ*BERT* generates some test coupled mutants (i.e. the mutants fail on the same tests but not for the same reason). In addition to the strongly and test coupled mutants, μ*BERT* generates mutants that partially couple with such vulnerabilities as well.

The Common Weakness Enumeration Specification (CWE) provides a common language of discourse for discussing, finding and dealing with the causes of software security vulnerabilities as they are found in code, design, or system architecture. Each individual CWE represents a single vulnerability type [12]. Interestingly, the LLM-generated mutants couple with a broad variety of vulnerabilities, coupling with 14 out of the 15 CWEs explored in our study. For instance, coupling with CWE-20 (Improper Input Validation), CWE-835 (Infinite Loop), CWE-79 (Improper Neutralization of Web Input), among others, missing only the CWE-287 (Improper Authentication). Finally, we study the distribution of coupled mutants and observe that: strongly coupled mutants represent 1.17% of the entire pool of killable mutants; test coupled mutants represent 7.2% of the overall killable mutants; while 64.97% of the killable mutants are not coupled with the vulnerabilities (i.e. they fail on tests that are not related with the vulnerabilities at all) and the remaining are partially (test) coupled with the vulnerabilities.

II. Background

A. Mutation Testing

Mutation testing is a popular fault-based testing technique [2], [18]. It works by introducing slight syntactic modifications to the original program, a.k.a., *mutants*. These mutants are artificially seeded faults that aim at simulating bugs present in the software. The tester designs test cases in order to *kill* these mutants, i.e., to distinguish the observable behavior between a mutant and the original program. Thus, selecting specific mutants enables testing specific structures of a given language that the testing process seeks [23]. Due to this flexibility, mutation testing is used to guide test generation [38], to perform test assessment [37], to uncover subtle faults [9], and to infer strong assertions [22], [30].

B. μBERT

μ*BERT* [17] is a mutation testing tool that uses a pre-trained language model *CodeBERT* to generate mutants by masking and replacing tokens. μ*BERT* takes a Java class and extracts the expressions to mutate. It then masks the token of interest, e.g. a variable name, and invokes CodeBERT to complete the masked sequence (i.e., to predict the missing token). This approach has been proven efficient in increasing the fault detection of test suites [17] and improving the accuracy of learning-based bug-detectors [42] and thus, we consider it as a representative of pre-trained language-model-based techniques. For instance, consider the sequence `int total = out.length;` taken form Figure 1a, μ*BERT* mutates the object field access expression `length` by feeding CodeBERT with the masked sequence `int total = out.<mask>;`. CodeBERT predicts the 5 most likely tokens to replace the masked one, e.g., it predicts `total`, `length`, `size`, `count` and `value` for the given masked sequence. μ*BERT* takes these predictions and generates mutants by replacing the masked token with the predicted ones (per masked token creates five mutants). μ*BERT* discards non-compilable mutants and those syntactically the same as the original program (cases in which CodeBERT predicts the original masked token).

C. Code Vulnerabilities

Common Vulnerability Exposures (CVE) [16] defines a security vulnerability as "*a flaw in a software, firmware, hardware, or service component resulting from a weakness that can be exploited, causing a negative impact to the confidentiality, integrity, or availability of an impacted component or components.*". The inadvertence of a developer or insufficient knowledge of defensive programming usually causes these weaknesses. Vulnerabilities are usually reported in publicly available databases to promote their disclosure and fix. One such example is National Vulnerability Database, aka NVD [35]. NVD is the U.S. government repository of standards based vulnerability management data. All vulnerabilities in the NVD have been assigned a CVE (Common Vulnerabilities and Exposures) identifier. The Common Vulnerabilities and Exposures (CVE) Program's primary purpose is to uniquely identify vulnerabilities and to associate specific versions of codebases (e.g., software and shared libraries) to those vulnerabilities. The use of CVEs ensures that two or more parties can confidently refer to a CVE identifier (ID) when discussing or sharing information about a unique vulnerability.

```java
private static void decompress
 (final InputStream in, final byte[] out)
 throws IOException {
  int position = 0;
  final int total = out.length;
  while (position < total) {
   final int n = in.read();



   if (n > 128) {
    final int value = in.read();
    for (int i = 0; i < (n & 0x7f); i++) {
     out[position++] = (byte) value; }
   } else {
    for (int i = 0; i < n; i++) {
    out[position++] = (byte) in.read();}
   }
  }
 }
```

(a) Vulnerable Code (CVE-2018-17201)

```java
private static void decompress
 (final InputStream in, final byte[] out)
 throws IOException {
  int position = 0;
  final int total = out.length;
  while (position < total) {
   final int n = in.read();
   if (n < 0) {
    throw new ImageReadException("Error
         decompressing RGBE file"); }
   if (n > 128) {
    final int value = in.read();
    for (int i = 0; i < (n & 0x7f); i++) {
     out[position++] = (byte) value; }
   } else {
    for (int i = 0; i < n; i++) {
    out[position++] = (byte) in.read();}
   }
  }
 }
```

(b) Fixed Code

```java
private static void decompress
 (final InputStream in, final byte[] out)
 throws IOException {
  int position = 0;
  final int total = out.length;
  while (position < total) {
   final int n = in.read();
   if (n == 0) { // '<' modified to '=='
    throw new ImageReadException("Error
         decompressing RGBE file"); }
   if (n > 128) {
    final int value = in.read();
    for (int i = 0; i < (n & 0x7f); i++) {
     out[position++] = (byte) value; }
   } else {
    for (int i = 0; i < n; i++) {
    out[position++] = (byte) in.read();}
   }
  }
 }
```

(c) Vulnerability-coupled Mutant

Fig. 1: Vulnerability CVE-2018-17201 (Fig. 1a) that allows “Infinite Loop” making code hang, which further enables Denial-of-Service (DoS) attack is fixed with the conditional exception using “`if`” expression (Fig. 1b). The mutant (Fig. 1c) modifies the “`if`” condition that nullifies the fix and strongly couples with the vulnerability.

```java
void addPathParam(String name, String
     value, boolean encoded) {
 if (relativeUrl == null) {
  throw new AssertionError(); }





 relativeUrl = relativeUrl.replace("{" +
      name + "}",
      canonicalizeForPath(value,
      encoded));






}
```

(a) Vulnerable Code (CVE-2018-1000850)

```java
void addPathParam(String name, String
     value, boolean encoded) {
 if (relativeUrl == null) {
  throw new AssertionError(); }
 String replacement =
      canonicalizeForPath(value, encoded);
 String newRelativeUrl =
      relativeUrl.replace("{" + name +
      "}", replacement);
 if (PATH_TRAVERSAL
  .matcher(newRelativeUrl)
  .matches()) {
   throw new IllegalArgumentException(
   "@Path parameters shouldn't perform
        path traversal ('.' or '..'): " +
        value); }
 relativeUrl = newRelativeUrl;
}
```

(b) Fixed Code

```java
void addPathParam(String name, String
     value, boolean encoded) {
 if (relativeUrl == null) {
  throw new AssertionError(); }
 String replacement =
      canonicalizeForPath(value, encoded);
 String newRelativeUrl =
      relativeUrl.replace("{" + name +
      "}", replacement);
 if (PATH_TRAVERSAL
  .matcher(name)//passed argument changed
  .matches()) {
   throw new IllegalArgumentException(
   "@Path parameters shouldn't perform
        path traversal ('.' or '..'): " +
        value); }
 relativeUrl = newRelativeUrl;
}
```

(c) Vulnerability-coupled Mutant

Fig. 2: Vulnerability CVE-2018-1000850 that allows “Path Traversal”, which further enables access to a Restricted Directory (Fig. 2a) is fixed with the conditional exception in case `'.'` or `'..'` appears in the “`newRelativeUrl`” (Fig. 2b). The strongly coupled mutant (Fig. 2c) changes the “`newRelativeUrl`” passed as an argument by “`name`”, which nullifies the fix and re-introduce the vulnerable behavior.

III. Motivating Examples

Figures 1 and 2 show motivating examples of mutants coupling with vulnerabilities. Figure 1 demonstrates the example of high severity (7.5) vulnerability CVE-2018-17201 [14] that allows “Infinite Loop”, a.k.a., a loop with unreachable exit condition when parsing input files. This makes the code hang which allows an attacker to perform a Denial-of-Service (DoS) attack. The vulnerable code (Fig. 1a) is fixed with the use of an “`if`” expression (Fig. 1b) to throw an exception and moves out of the loop in case of such an event. Fig. 1c shows one *µBERT*'s mutant in which the “`if`” condition is modified, i.e., the binary operator “<” is modified to “==”. This modification makes the “`if`” condition never executed, nullifying the fix, and behaving exactly the same as the vulnerable code.

Figure 2 demonstrates the example of another high severity vulnerability CVE-2018-1000850 [13] that allows “Directory Traversal” that can result in an attacker manipulating the URL to add or delete resources otherwise unavailable to him/her. The vulnerable code (Fig. 2a) is fixed with the use of an “`if`” expression (Fig. 2b) to throw an exception in case `'.'` or `'..'` appears in the “`newRelativeUrl`” (Fig. 2b). Fig. 2c shows one *µBERT*'s mutant in which the passed argument is changed from “`newRelativeUrl`” to “`name`” which changes the matching criteria, hence nullifying the fix, and introducing same vulnerability behavior.

IV. Research Questions

We start our analysis by investigating how many vulnerabilities in our dataset can be behaviourally coupled by one or more *µBERT* mutants, i.e., mutants failing the PoVs (tests that were failed by the respective vulnerabilities). Therefore we ask:

RQ1 How many vulnerabilities (CVEs) and their types (CWEs) can be coupled by LLM-based mutants?

For this task, we rely on *Vul4J* dataset [8] (section V-A) for obtaining vulnerable projects with vulnerabilities, corresponding fixes, and PoV test suites, and on *µBERT* [17] (section II-B) for generating mutants. In the *Vul4J* dataset,

TABLE I: The table records the Vulnerability dataset details that include CVE ID, CWE ID and description, Severity level (that ranges from 0 to 10), number of Files and Methods that were modified during the vulnerability fix, and number of Tests that are failed by the vulnerability a.k.a. Proof of Vulnerability (PoV).

CVE (Vulnerability)	**CWE (Vulnerability type)**	**CWE description (Description of Vulnerability cause)**	**Severity (0 - 10)**	**# Files modified**	**#Methods modified**	**Failed Tests (PoV)**
CVE-2017-18349	CWE-20	Improper Input Validation	9.8	1	1	1
CVE-2013-2186	CWE-20	Improper Input Validation	7.5	1	1	2
CVE-2014-0050	CWE-264	Permissions, Privileges, and Access Controls	7.5	2	5	1
CVE-2018-17201	CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')	7.5	1	1	1
CVE-2015-5253	CWE-264	Permissions, Privileges, and Access Controls	4.0	1	1	1
HTTPCLIENT-1803	CWE-noinfo	No information provided by NIST	NA	1	1	1
PDFBOX-3341	CWE-noinfo	No information provided by NIST	NA	1	1	1
CVE-2017-5662	CWE-611	Improper Restriction of XML External Entity Reference	7.3	1	2	1
CVE-2018-11797	CWE-noinfo	No information provided by NIST	5.5	1	1	1
CVE-2016-6802	CWE-284	Improper Access Control	7.5	1	1	3
CVE-2016-6798	CWE-611	Improper Restriction of XML External Entity Reference	9.8	1	2	1
CVE-2017-15717	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	6.1	1	2	2
CVE-2016-4465	CWE-20	Improper Input Validation	5.3	1	1	1
CVE-2014-0116	CWE-264	Permissions, Privileges, and Access Controls	5.8	1	4	1
CVE-2016-8738	CWE-20	Improper Input Validation	5.8	1	1	2
CVE-2016-4436	CWE-noinfo	No information provided by NIST	9.8	1	2	1
CVE-2016-2162	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	6.1	1	2	1
CVE-2018-8017	CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')	5.5	1	2	1
CVE-2014-4172	CWE-74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')	9.8	2	2	1
CVE-2019-3775	CWE-287	Improper Authentication	6.5	1	1	1
CVE-2018-1002200	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	5.5	1	1	1
CVE-2017-1000487	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	9.8	3	17	12
CVE-2018-20227	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	7.5	1	5	1
CVE-2013-5960	CWE-310	Cryptographic Issues	5.8	1	2	15
CVE-2018-1000854	CWE-74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')	9.8	1	2	1
CVE-2016-3720	CWE-noinfo	No information provided by NIST	9.8	1	1	1
CVE-2016-7051	CWE-611	Improper Restriction of XML External Entity Reference	8.6	1	1	1
CVE-2018-1000531	CWE-20	Improper Input Validation	7.5	1	1	1
CVE-2018-1000125	CWE-20	Improper Input Validation	9.8	1	4	1
APACHE-COMMONS-001	CWE-noinfo	No information provided by NIST	NA	1	1	1
CVE-2013-4378	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	4.3	1	1	1
CVE-2018-1000865	CWE-269	Improper Privilege Management	8.8	1	3	1
CVE-2018-1000089	CWE-532	Insertion of Sensitive Information into Log File	7.4	1	2	1
CVE-2015-6748	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	6.1	1	1	1
CVE-2016-10006	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	6.1	1	1	1
CVE-2018-1000615	CWE-noinfo	No information provided by NIST	7.5	1	1	1
CVE-2017-8046	CWE-20	Improper Input Validation	9.8	2	5	1
CVE-2018-11771	CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')	5.5	1	1	2
CVE-2018-15756	CWE-noinfo	No information provided by NIST	7.5	1	5	2
CVE-2018-1000850	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	7.5	1	2	3
CVE-2017-1000207	CWE-502	Deserialization of Untrusted Data	8.8	1	3	1
CVE-2019-10173	CWE-502	Deserialization of Untrusted Data	9.8	1	7	1
CVE-2019-12402	CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')	7.5	1	1	1
CVE-2020-1953	CWE-noinfo	No information provided by NIST	10.0	1	7	2

the fixes (for the vulnerabilities) passed the corresponding project's test suite (containing the PoV tests) in 45 cases for which we mention the details in Table I. μ*BERT* produces mutants from the fixed version, which are checked for coupling the observable behavior of corresponding vulnerability by executing the mutants on the PoV test suites. We manually analyze whether the generated mutants and vulnerabilities break the same tests for the same reasons, and determine how semantically similar the generated mutants are, i.e., if they are strongly, partially, or not coupled at all.

We also study the prevalence and distribution of μ*BERT* mutants that couple with the vulnerabilities. Hence, we ask:

RQ2 What is the prevalence and distribution of these LLM-based mutants that couple with the vulnerabilities?

V. Experimental Setup and Methodology

A. Vul4J*: the set of reproducible vulnerabilities under study*

There exist several vulnerability datasets for many programming languages [5], [19], [21]. However, they do not contain bug-witnessing test cases to reproduce vulnerabilities, i.e., Proof of Vulnerability (PoV). Such test cases are essential for this study in order to determine whether generated mutants are Vulnerability-coupled mutants, as explained in the section above. In general, it is hard to reproduce exploitation (i.e., PoV) for vulnerabilities. *Vul4J* [8] is a dataset of real vulnerabilities, with the corresponding fixes and the PoV test cases, that we utilized for this study. However, due to a few test cases failing even after applying the provided vulnerability-fixes, we had to exclude a few vulnerabilities. In total, we conducted this study on 45 vulnerabilities. In Table I, we mention the details of considered vulnerabilities that include CVE ID, CWE ID and description, Severity level (that ranges from 0 to 10, provided by National Vulnerability Database [35]), number of Files and Methods that were modified during the vulnerability fix, and number of Tests that are failed by the vulnerability a.k.a. Proof of Vulnerability (PoV).

B. Vulnerability-Coupling Classes

Similarly to the procedure followed by Gay and Salahirad [24] to study the coupling between mutants and real-faults, we determine the degree of coupling between a mutant with a vulnerability by analyzing the failing tests and the reasons of the failures. Hence, given a mutant, a vulnerability, and the developer-written test suite with the corresponding PoVs, we define the following classes:

- *Strong Coupling*: if the mutant and the vulnerability fail on exactly the same PoV tests for the same reasons (i.e. same exception/error is thrown). In the case where the mutant fails on additional tests not triggering the vulnerability (i.e. not PoVs), we denote it by *Strong Coupling + Additional*.
- *Test Coupling*: if the mutant and the vulnerability fail on exactly the same PoV tests but one or more fail for differing reasons. Whether the mutant fails on additional tests, we denote it by *Test Coupling + Additional*.
- *Partial Coupling*: if the mutant and the vulnerability fail on some, but not all, PoV tests for exactly the same reasons. If the mutant also fails on additional not PoV tests, we denote it by *Partial Coupling + Additional*.
- *Partial Test Coupling*: if the mutant and the vulnerability fail on some, but not all the PoV tests but one or more fail for differing reasons. We use *Partial Test Coupling + Additional* to indicate that the mutant fail on additional tests.
- *No Coupling*: if the mutant is only killed by tests not triggering the vulnerability (i.e. the PoVs do not kill the mutant).

It is worth clarifying that our manual analysis focuses on the killable mutants since our similarity metrics rely on the observable behavior of test executions (non-killable mutants are trivially dissimilar to vulnerabilities' behavior).

C. Experimental Procedure

Our empirical analysis goes as follows:

1) We started by installing and analyzing the vulnerabilities from *Vul4J*. To perform mutation analysis we required a passing test suite after applying the fix of the vulnerability. We noticed that some test cases failed even after applying the vulnerability-fixes provided by *Vul4J*. Hence, we had to exclude these few cases. We finally ended up considering 45 vulnerabilities for our study, shown in Table I, for which the PoV test suites fail on the vulnerable versions and pass on the corresponding fixes.
2) For each vulnerability, we execute the PoV test suite and record the failing tests and the reasons for failure (i.e., exceptions or error messages).
3) We generate the mutants by running *μBERT* [17] on the modified files of the vulnerability-fixes for the 45 projects, producing in total *7,725* mutants killable by the provided Vul4J developer-written suites.
4) We re-execute the vulnerabilities test suites on each mutant and record the failing tests and the reasons for failure.
5) Finally, we assess the behavioral similarities between the mutants and vulnerabilities to determine their coupling category according to our definition in Section V-B.

Once the analysis is complete, to answer *RQ1*, we first check which mutants failed the same tests as the vulnerability, and then we do a manual analysis to determine if they fail for the same reasons or not. This will allow us to determine how many vulnerabilities are (strongly or partially) coupled by the generated mutants. To answer *RQ2*, we focus on the number of mutants coupling with the vulnerabilities to determine their prevalence and distribution.

VI. Experimental Results

A. RQ1: Vulnerabilities Coupling

Figure 3.a summarises the number of vulnerabilities for which at least one mutant couples with them. Precisely, we can observe that *μBERT* mutants strongly or partially couple with 39 out of the 45 vulnerabilities analyzed, while for the remaining 6 vulnerabilities, no generated mutant shares a behavioral similarity with the vulnerabilities exhibited by the developer-written test suites in Vul4J. In the following, we discuss in detail the different categories of coupling observed.

a) Strong Coupling: Figure 3.b shows that for 21 out of the 45 studied vulnerabilities, *μBERT* generated at least one mutant that strongly coupled with those vulnerabilities. This means that these strongly coupled mutants break exactly the same tests for the same reasons as the vulnerabilities. We also observe that for 11 more vulnerabilities (24 in total) there is at least one mutant that behaves the same as the vulnerability but also fails on additional tests (Strong Coupling + Additional in Figure 3.b). Overall, considering these two sets of mutants together, we can observe that *μBERT* generates mutants that behave the same or almost the same as 32 out of the 45 vulnerabilities. Thus, guiding the mutation testing process with these CodeBERT-generated mutants can lead to test suites that potentially find code vulnerabilities.

b) Test Coupling: Figure 3.c indicates that for 11 out of the 45 vulnerabilities, *μBERT* can generate at least one mutant that fails the same tests as the vulnerability but a few of the failures are for a different reason. This number can go to 32

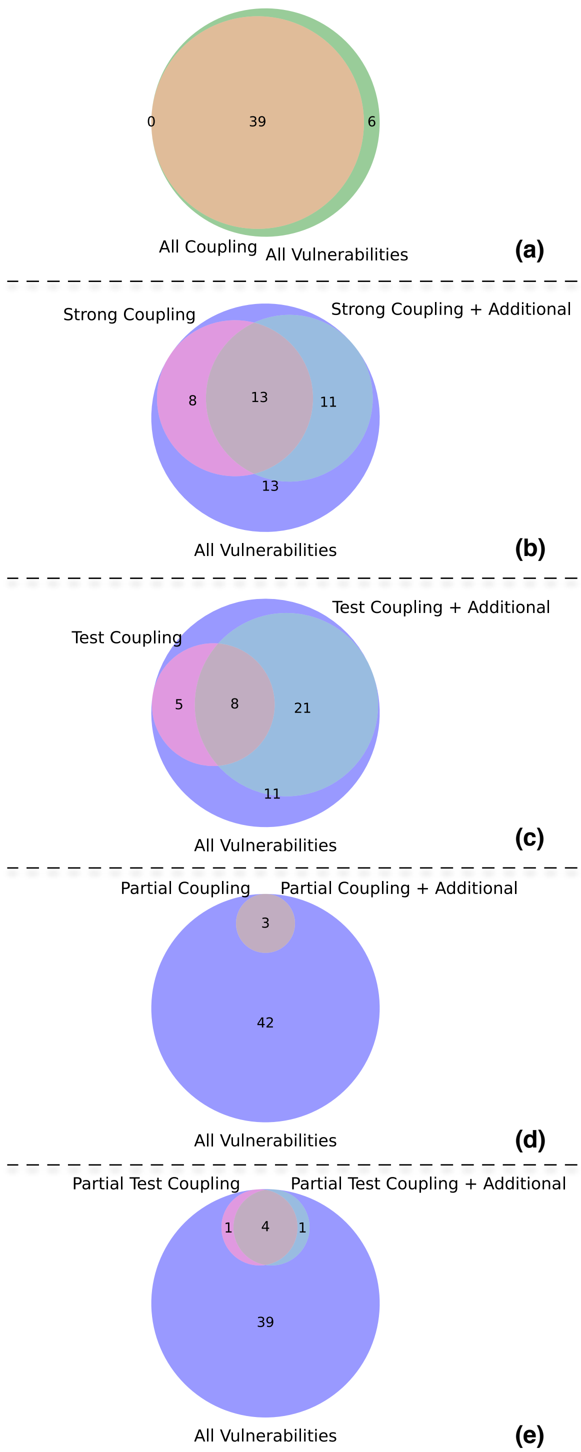

Fig. 3: Vulnerability coupling distribution

if we also consider the mutants that fail some additional tests than the vulnerabilities. Overall, test coupled mutants can help to detect closely similar, but not the same, program deviations as the vulnerabilities, which makes them a very promising set of test requirements for security-aware testing.

c) Partial Coupling: Figure 3.d shows that for 3 vulnerabilities, μ*BERT* generated mutants partially couple with vulnerabilities. This infers that these mutants break some of

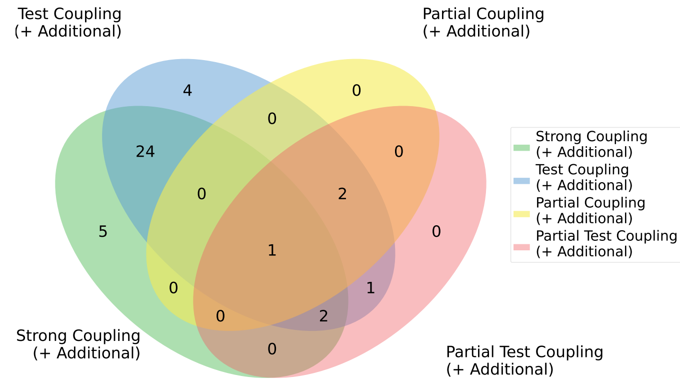

Fig. 4: Coupling complementarity w.r.t. CVE

the tests (but not all) that are triggered by the vulnerabilities and for the same reasons as well. We also observed that these 3 vulnerabilities are also partially coupled with the mutants (Partial Coupling + Additional) that, in addition to breaking some of the vulnerability-triggering tests, also break other tests as shown in the Figure 3.d. As these mutants capture partially the deviations induced by the 3 coupled vulnerabilities, these may also serve as partial test requirements for designing security-specific test suites.

d) Partial Test Coupling: Figure 3.e shows that, for 5 vulnerabilities, mutants generated by μ*BERT* couple with vulnerabilities by breaking some of the tests that are also triggered by the vulnerabilities irrespective of the reasons for failures (Partial Test Coupling mutants). Furthermore, for 1 additional vulnerability, the generated mutants break some of the tests triggered by the vulnerability along with breaking other tests (Partial Test Coupling + Additional in Figure 3.e). As these mutants break some of the tests triggered by the 5 vulnerabilities, irrespective of the reasons for the failures, these can also help in the mutation testing process as test requirements to target potential vulnerabilities.

e) Coupling Complementarity: We also study the complementary nature of the mutants belonging to different vulnerability-coupling classes. Figure 4 shows the different kinds of coupling for each vulnerability. For just 1 vulnerability (CVE-2016-6802), we observed that μ*BERT*-generated mutants couple with all vulnerability-coupling classes (i.e., strongly, partially, and test coupling). We also observe that for 5 vulnerabilities, μ*BERT*-generated mutants fully or partially "test couple" with the vulnerability, but no mutant neither strongly nor partially coupled with it. Interestingly, for 5 vulnerabilities we observed that μ*BERT*-generated mutants either strongly couple or have no relation at all with the vulnerabilities. Interestingly, for the 39 vulnerabilities that are coupled somehow by a mutant, we observe that there always exists at least one μ*BERT*-generated mutant that fails all the same tests as the vulnerabilities (and possibly some additional tests) irrespective of the reasons. That is, the vulnerability-coupling classes "Strong Coupling (+ Additional)" and "Test Coupling (+ Additional)" cover the 39 coupled vulnerabilities. This is very appealing as it indicates that the context-aware

TABLE II: Scope of Mutant-coupling with Vulnerability Types

Vulnerability Types (CWE)	LLM-generated Mutants								
	Strong Coupling	**Strong Coupling + Additional Tests Failed**	**Test Coupling**	**Test Coupling + Additional Tests Failed**	**Partial Coupling**	**Partial Coupling + Additional Tests Failed**	**Partial Test Coupling**	**Partial Test Coupling + Additional Tests Failed**	**Overall Coupling Score**
CWE-20	2	6		4					6/7
CVE-2013-2186		✓		✓					
CVE-2016-4465		✓		✓					
CVE-2016-8738		✓		✓					
CVE-2017-18349		✓		✓					
CVE-2017-8046									
CVE-2018-1000125	✓	✓							
CVE-2018-1000531	✓	✓							
CWE-22	2	3	1	3			1	1	3/3
CVE-2018-1000850	✓	✓	✓	✓			✓	✓	
CVE-2018-1002200		✓		✓					
CVE-2018-20227	✓	✓		✓					
CWE-264		1		2					2/3
CVE-2014-0050									
CVE-2014-0116		✓		✓					
CVE-2015-5253				✓					
CWE-269	1	1		1					1/1
CVE-2018-1000865	✓	✓		✓					
CWE-284	1	1	1	1	1	1	1	1	1/1
CVE-2016-6802	✓	✓	✓	✓	✓	✓	✓	✓	
CWE-287									0/1
CVE-2019-3775									
CWE-310	1	1							1/1
CVE-2013-5960	✓	✓							
CWE-502	1	1		1					1/2
CVE-2017-1000207									
CVE-2019-10173	✓	✓		✓					
CWE-532	1	1							1/1
CVE-2018-1000089	✓	✓							
CWE-611	2	1	2	2					3/3
CVE-2016-6798	✓		✓	✓					
CVE-2016-7051		✓		✓					
CVE-2017-5662	✓		✓						
CWE-74	2		2	1					2/2
CVE-2014-4172	✓		✓	✓					
CVE-2018-1000854	✓		✓						
CWE-78				1	1	1	1	1	1/1
CVE-2017-1000487				✓	✓	✓	✓	✓	
CWE-79	2	2	1	4					5/5
CVE-2013-4378				✓					
CVE-2015-6748		✓		✓					
CVE-2016-10006	✓			✓					
CVE-2016-2162	✓	✓		✓					
CVE-2017-15717			✓						
CWE-835	4	1	3	3				1	4/4
CVE-2018-11771	✓	✓	✓	✓				✓	
CVE-2018-17201	✓		✓	✓					
CVE-2018-8017	✓		✓						
CVE-2019-12402	✓			✓					
CWE-noinfo	2	5	3	6	1	1	2	1	8/9
APACHE-COMMONS-001	✓	✓							
CVE-2016-3720		✓		✓					
CVE-2016-4436									
CVE-2018-1000615			✓						
CVE-2018-11797		✓	✓	✓					
CVE-2018-15756				✓	✓	✓	✓	✓	
CVE-2020-1953				✓			✓		
HTTPCLIENT-1803	✓	✓		✓					
PDFBOX-3341		✓	✓	✓					

mutations are very effective in deviating the program behavior similar to vulnerabilities.

f) Vulnerability Type (CWE) Coupling: We observe that μ*BERT*-generated mutants coupled with 14 out of 15 vulnerability types (CWE) representing multiple instances (CVEs) of the vulnerabilities. Table II details the scope of mutant-coupling with vulnerability types. An exception to this is the case of CWE-287, which is concerned with the Improper Authentication practices in the source code for which we have 1 instance, i.e., CVE-2019-3775 in our vulnerability set. For vulnerability instances (CVE) where NVD [35] contains no information under which vulnerability type these can be categorized, we group such instances under CWE-noinfo. We observe that μ*BERT*-generated mutants either Strongly couple or Test Couple (with Additional Tests Fail) with 8 out of 9 instances of CWE-noinfo. This observation of ours is also valid for other CWEs in our vulnerability set, where most mutants are either Strongly coupled or Test coupled (with

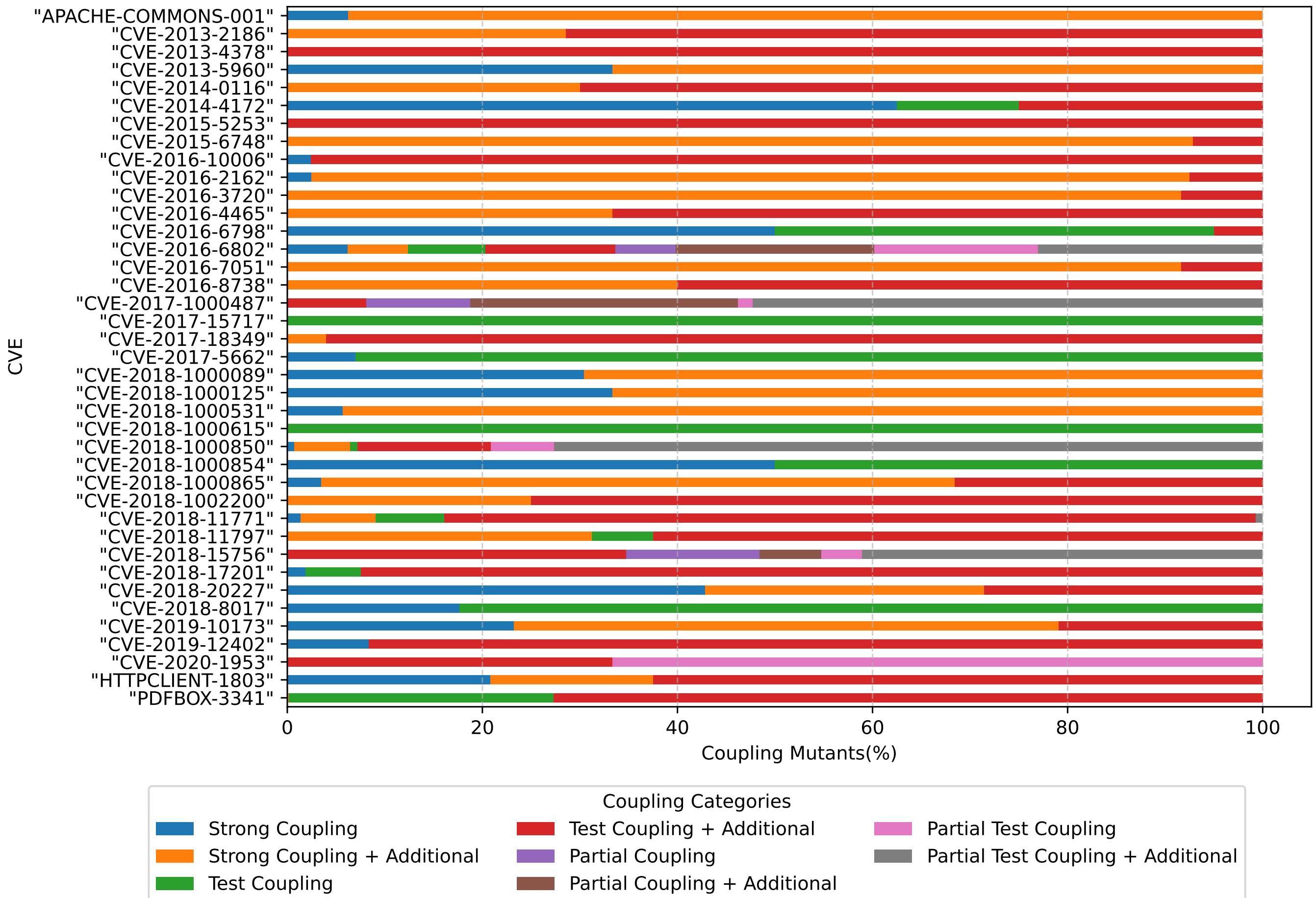

Fig. 5: Coupling Mutants' distribution across Vulnerabilities

additional tests failed) with the CWEs and complement each other.

> Answer to RQ1: *μBERT*-generated mutants couple with 39 out of 45 vulnerabilities, where 32 can be strongly coupled (i.e. the mutants fail on the same tests for the same reasons) and the remaining 7 can be test coupled (i.e. the mutants fail on the same tests but not necessarily for the same reason). This finding provides evidence that pre-trained language models have the capability to generate test requirements (mutants) that induce program behavioral deviations similar to vulnerabilities, making possible an effective vulnerability-aware mutation testing process.

B. RQ2: Prevalence and Distribution of Vulnerability-coupling Mutants

Figure 5 and Table III show the distribution of the different coupled mutants across all the studied vulnerabilities. Out of the *7,725* killable mutants generated by *μBERT*, a total of 90 mutants strongly coupled with the vulnerabilities representing the 1.17% of the killable mutants. If we also consider the 288 mutants (3.73%) that strongly couple with the vulnerabilities but also break other tests (Strong Coupling + Additional), we end up with a total of 378 mutants (4.9%) that behave almost the same as the vulnerabilities. Moreover, we can observe that 556 (7.2%) and 1,341 (17.36%) mutants break the same tests, plus some additional tests respectively, as the vulnerabilities, leading to a total of 1,897 mutants (24,56%) that *test couple* with the vulnerabilities. We can also observe that 431 mutants (41, 83, 37, and 270, respectively) maintain some kind of partial coupling or test coupling with the vulnerabilities, constituting 5.58% on the killable mutants. The remaining 5,019 mutants of the killable mutants, i.e. the 64.97%, fail on tests that are not related to the vulnerabilities at all.

Given the scarcity of *μBERT*-generated mutants that strongly couple with vulnerabilities (only 1.17% of the total killable mutants), it might be important in the future to account for an approach that can prioritize or predict which mutants are more likely to couple with vulnerabilities. This may constitute an open and challenging problem for future research.

TABLE III: Distribution of Mutants coupling with Vulnerabilities

CVE (Vulnerabilities)	**LLM-generated Mutants**									
	Strong Coupling	**Strong Coupling + Additional Tests Failed**	**Test Coupling**	**Test Coupling + Additional Tests Failed**	**Partial Coupling**	**Partial Coupling + Additional Tests Failed**	**Partial Test Coupling**	**Partial Test Coupling + Additional Tests Failed**	**Other Tests Failed (No Coupling)**	**Total Killable**
APACHE-COMMONS-001	1	15							57	73
CVE-2013-2186		2		5					68	75
CVE-2013-4378				2					25	27
CVE-2013-5960	1	2							62	65
CVE-2014-0050									270	270
CVE-2014-0116		3		7					42	52
CVE-2014-4172	10		2	4					57	73
CVE-2015-5253				46					10	56
CVE-2015-6748		26		2					222	250
CVE-2016-10006	1			40					159	200
CVE-2016-2162	1	36		3					45	85
CVE-2016-3720		11		1					191	203
CVE-2016-4436									41	41
CVE-2016-4465		3		6					21	30
CVE-2016-6798	10		9	1					294	314
CVE-2016-6802	7	7	9	15	7	23	19	26	42	155
CVE-2016-7051		11		1					191	203
CVE-2016-8738		4		6					23	33
CVE-2017-1000207									9	9
CVE-2017-1000487				16	21	54	3	103	37	234
CVE-2017-15717			77						229	306
CVE-2017-18349		1		24					96	121
CVE-2017-5662	6		80							86
CVE-2017-8046									9	9
CVE-2018-1000089	7	16							53	76
CVE-2018-1000125	14	28							64	106
CVE-2018-1000531	2	33							82	117
CVE-2018-1000615			38							38
CVE-2018-1000850	1	8	1	19			9	101	74	213
CVE-2018-1000854	1		1							2
CVE-2018-1000865	2	37		18					202	259
CVE-2018-1002200		3		9					143	155
CVE-2018-11771	2	11	10	119				1	605	748
CVE-2018-11797		5	1	10					92	108
CVE-2018-15756				33	13	6	4	39	89	184
CVE-2018-17201	2		6	98					65	171
CVE-2018-20227	3	2		2					2	9
CVE-2018-8017	3		14							17
CVE-2019-10173	10	18		1					564	593
CVE-2019-12402	1			11					113	125
CVE-2019-3775									9	9
CVE-2020-1953				1			2		2	5
HTTPCLIENT-1803	5	4		15					316	340
PDFBOX-3341		2	308	826					344	1,480
Total	90 (1.17%)	288 (3.73%)	556 (7.2%)	1,341 (17.36%)	41 (0.53%)	83 (1.07%)	37 (0.48%)	270 (3.5%)	5,019 (64.97%)	7,725

Answer to RQ2: Only 90 *μBERT*-generated mutants (i.e., 1.17% of mutant set) strongly couple with the vulnerabilities, and a further 288 mutants (i.e., 3.73%) behave the same as the vulnerabilities but also fail a few additional tests. Moreover, a total of 556 mutants (i.e., 7.2% of the mutant set) *test couple* with the vulnerabilities (by failing the same tests but for different reasons). Considering the scarcity of strongly coupled mutants, it may be imperative to employ an effective mutant selection method to facilitate a rather efficient vulnerability-aware mutation testing process.

VII. Threats to Validity

External Validity: Threats may relate to the generalization of our results w.r.t. the vulnerabilities that we were unable to consider for our study due to their non-existing PoV tests. We consider this threat of low importance as our evaluation expands to 15 different types (CWEs) of vulnerabilities with severity ranging from 10.0 (highest in the severity scale) to 4.0 (low). Additionally, our considered vulnerabilities have their fixes spanning from a single method to multiple methods modified during the fix (as shown in Table I). We verify all the vulnerabilities and their fixes by executing PoV tests provided by *Vul4J* [8]. Other threats may relate to the mutant generation tool, i.e., *μBERT* that we used. This choice was made since *μBERT* relies on CodeBERT to produce mutations that look natural and are effective for mutation testing. We consider this threat of low importance since *μBERT* effectively coupled 39 vulnerabilities, and by using a better mutant generation tool one can produce more vulnerability-coupled mutants, augmenting the chances of coupling other vulnerabilities.

Internal Validity: Threats may relate to the developer-written test suites we used and the mutants considered as vulnerability coupled. We used well-tested projects provided by the Vul4J dataset [8]. To be more accurate, our underlying assumption is that the extensive pool of tests including the Proof-of-Vulnerability (PoV) available in our experiments is a valid approximation of the program's test executions, especially the proof of a vulnerability and its verified fix, and capture the developers' intentions of which parts of the program are worth to be tested.

Construct Validity: Threats may relate to our metric to measure the similarity of a mutant and a vulnerability, i.e., the vulnerability-coupling classes. We relied on the failing tests and reasons for failure because it is widely known in the fault-seeding community as a representative metric to capture the semantic similarity between a seeded and real fault [24]. In the context of this study, the seeded fault is a mutant and the real fault is a vulnerability. We consider this threat of low importance since the failed tests and failure reasons of a mutant and a vulnerability represent their observable behavior and serves its purpose for this study.

VIII. Related Work

The *coupling effect* between seeded and real faults have been widely studied by the community, focused specially in traditional grammar-based mutations [24], [27], [39]. Some recent studies also provided evidence that mutations generated by LLMs, like μ*BERT*, can effectively couple with real-faults [36]. Despite there are several approaches that design specific patterns to try to inject specific vulnerabilities (discussed below), there is no evidence whether the mutations generated by pre-trained language models can couple or not with vulnerabilities behavior.

The unlikelihood of standard PIT [11] operators to produce security-aware mutants was observed by Loise et al. [29] where the authors designed pattern based operators to target specific vulnerabilities. They relied on static analysis for validation of generated mutants to have similarities with their targeted vulnerabilities.

Fault modeling related to security policies was explored by Mouehli et al. [33] where they designed mutation operators corresponding to fault models for access control security policies. Their designed operators targeted modifying user roles and deleting policy rules to modify application context, targeting the implementation of access control policies.

Mutating high-level security protocol language (HLPSL) models to generate abstract test cases was explored by Dadeau et al. [15] where their proposed mutations targeted to introduce leaks in the security protocols. They relied on the automated validation of Internet security protocols and applications tool set to declare the mutated protocol unsafe and capable of exploiting the security flaws.

Targeting black box testing by mutating web applications' abstract models was explored by Buchler et al. [7] where they produced model mutants by removing authorization checks and introducing noisy (non-sanitized) data. They relied on model-checkers to generate execution traces of their mutated models to create of intermediate test cases. Their work was focused on guiding penetration testers to find attacks exploiting implementation-based vulnerabilities (e.g., a missing check in a RBAC system, or non-sanitized data leading to XSS attacks).

Similar to Loise et al., Nanavati et al. [34] also show that traditional mutation operators only simulate some simple syntactic errors. Hence, they designed memory mutation operators to target memory faults and control flow deviation. They focused on programs in C language and relied on memory allocation primitives in specific to C. Similarly, Shahriar and Zulkernine [43] and Ghosh et al. [25] also defined mutation operators related to the memory faults. Their designed operators also exploited memory manipulation in C programs (such as buffer overflows, uninitialized memory allocations, etc.), which security attacks may exploit. These works also focused on programs in C language.

Unlike the above-mentioned related works, we do not target a specific vulnerability pattern/type. Also, since we rely on a pre-trained language model (employed by μ*BERT*), we do not require to design specific mutation operators to target specific security issues. Additionally, our validation of vulnerability-coupled mutants is not based on a static analysis, but rather a dynamic proof as our produced/predicted vulnerability-coupled mutants fail tests that were failed by respective vulnerabilities, a.k.a., Proof-of-vulnerability (PoV).

IX. Conclusion

In this study, we showed that a large language model based mutation testing tool can effectively generate mutants that couple with the observable behavior of vulnerabilities. We showed that μ*BERT* can generate mutants that break the same tests and for the same reasons as 32 out of the 45 studied vulnerabilities. Additionally, μ*BERT* can generate mutants that, although for not the same reasons, break the same tests as the other (remaining) 7 vulnerabilities. Overall, the Large Language Model based mutation managed to "strongly couple" or "test couple" a total of 39 out of the 45 vulnerabilities. This provides evidence that LLMs can produce mutations that deviate program behaviors in the same way as the vulnerabilities. We also observed that strongly coupled mutants are a few, i.e., 1.17% of the entire mutant set. Thus, there is a need to prioritize or select these mutants to facilitate a rather efficient vulnerability-aware mutation testing process. We plan to explore this line of research in the near future.

X. Data Availability

The dataset consisting of our executable scripts and the source code of all projects (vulnerable and fixed), individual classes modified during the fix, i.e., vulnerable and fixed (where fixed classes were used for mutation), and the generated mutants are publicly available in our GitHub repository[1].

Acknowledgment

This work is supported by the Luxembourg National Research Fund (FNR) through the CORE project grant C20/IS/14761415/TestFlakes.

[1]https://github.com/garghub/VulnerabilityCouplingMutants

References

[1] Paul Ammann, Márcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 21–30. IEEE Computer Society, 2014.

[2] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[3] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. Code generation tools (almost) for free? A study of few-shot, pre-trained language models on code. *CoRR*, abs/2206.01335, 2022.

[4] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What it would take to use mutation testing in industry - A study at facebook. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP)*, pages 268–277. IEEE, 2021.

[5] Guru Prasad Bhandari, Amara Naseer, and Leon Moonen. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In Shane McIntosh, Xin Xia, and Sousuke Amasaki, editors, *PROMISE '21: 17th International Conference on Predictive Models and Data Analytics in Software Engineering, Athens Greece, August 19-20, 2021*, pages 30–39. ACM, 2021.

[6] Matthias Büchler, Johan Oudinet, and Alexander Pretschner. Security mutants for property-based testing. In Martin Gogolla and Burkhart Wolff, editors, *Tests and Proofs - 5th International Conference, TAP@TOOLS 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings*, volume 6706 of *Lecture Notes in Computer Science*, pages 69–77. Springer, 2011.

[7] Matthias Büchler, Johan Oudinet, and Alexander Pretschner. Semi-automatic security testing of web applications from a secure model. In *Sixth International Conference on Software Security and Reliability, SERE 2012, Gaithersburg, Maryland, USA, 20-22 June 2012*, pages 253–262. IEEE, 2012.

[8] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*, pages 464–468. ACM, 2022.

[9] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 597–608. IEEE / ACM, 2017.

[10] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Trans. Software Eng.*, 47(9):1943–1959, 2021.

[11] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: A practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 449–452, New York, NY, USA, 2016. Association for Computing Machinery.

[12] Common weakness enumeration. https://nvd.nist.gov/vuln/categories, (accessed January 02, 2024).

[13] Cve-2018-1000850. https://nvd.nist.gov/vuln/detail/CVE-2018-1000850, (accessed January 10, 2023).

[14] Cve-2018-17201. https://nvd.nist.gov/vuln/detail/CVE-2018-17201, (accessed January 10, 2023).

[15] Frédéric Dadeau, Pierre-Cyrille Héam, Rafik Kheddam, Ghazi Maatoug, and Michaël Rusinowitch. Model-based mutation testing from security protocols in HLPSL. *Softw. Test. Verification Reliab.*, 25(5-7):684–711, 2015.

[16] Definition of vulnerability. https://www.cve.org/ResourcesSupport/Glossary/#, (accessed January 10, 2023).

[17] Renzo Degiovanni and Mike Papadakis. μbert: Mutation testing using pre-trained language models. In *15th IEEE International Conference on Software Testing, Verification and Validation Workshops ICST Workshops 2022, Valencia, Spain, April 4-13, 2022*, pages 160–169. IEEE, 2022.

[18] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[19] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A C/C++ code vulnerability dataset with code changes and CVE summaries. In Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup, editors, *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, pages 508–512. ACM, 2020.

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics, 2020.

[21] Aayush Garg, Renzo Degiovanni, Matthieu Jimenez, Maxime Cordy, Mike Papadakis, and Yves Le Traon. Learning from what we know: How to perform vulnerability prediction using noisy historical data. *Empir. Softw. Eng.*, 27(7):169, 2022.

[22] Aayush Garg, Renzo Degiovanni, Facundo Molina, Maxime Cordy, Nazareno Aguirre, Mike Papadakis, and Yves Le Traon. Enabling efficient assertion inference. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 623–634, 2023.

[23] Aayush Garg, Milos Ojdanic, Renzo Degiovanni, Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. Cerebro: Static subsuming mutant selection. *IEEE Transactions on Software Engineering*, pages 1–1, 2022.

[24] Gregory Gay and Alireza Salahirad. How closely are common mutation operators coupled to real faults? In *IEEE Conference on Software Testing, Verification and Validation, ICST 2023, Dublin, Ireland, April 16-20, 2023*, pages 129–140. IEEE, 2023.

[25] Anup K. Ghosh, Tom O'Connor, and Gary McGraw. An automated approach for identifying potential vulnerabilities in software. In *Security and Privacy - 1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3-6, 1998, Proceedings*, pages 104–114. IEEE Computer Society, 1998.

[26] Philipp Görz, Björn Mathis, Keno Hassler, Emre Güler, Thorsten Holz, Andreas Zeller, and Rahul Gopinath. Systematic assessment of fuzzers using mutation analysis. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4535–4552, Anaheim, CA, August 2023. USENIX Association.

[27] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 654–665. ACM, 2014.

[28] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, page 473–485, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Thomas Loise, Xavier Devroey, Gilles Perrouin, Mike Papadakis, and Patrick Heymans. Towards security-aware mutation testing. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*, pages 97–102. IEEE Computer Society, 2017.

[30] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. Fuzzing class specifications. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1008–1020. ACM, 2022.

[31] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. Evospex: An evolutionary algorithm for learning postconditions. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1223–1235. IEEE, 2021.

[32] Tejeddine Mouelhi, Franck Fleurey, Benoit Baudry, and Yves Le Traon. A model-based framework for security policy specification, deployment and testing. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *Lecture Notes in Computer Science*, pages 537–552. Springer, 2008.

[33] Tejeddine Mouelhi, Yves Le Traon, and Benoit Baudry. Mutation analysis for security tests qualification. In *Testing: Academic and*

Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007), pages 233–242, 2007.

[34] Jay Nanavati, Fan Wu, Mark Harman, Yue Jia, and Jens Krinke. Mutation testing of memory-related operators. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, pages 1–10. IEEE Computer Society, 2015.

[35] National vulnerability database. https://nvd.nist.gov, (accessed January 10, 2023).

[36] Milos Ojdanic, Aayush Garg, Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Syntactic versus semantic similarity of artificial and real faults in mutation testing studies. *IEEE Trans. Software Eng.*, 49(7):3922–3938, 2023.

[37] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 354–365. ACM, 2016.

[38] Mike Papadakis and Nicos Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*, pages 121–130. IEEE Computer Society, 2010.

[39] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are mutation scores correlated with real fault detection?: a large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 537–548, 2018.

[40] Jibesh Patra and Michael Pradel. Semantic bug seeding: a learning-based approach for creating realistic bugs. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE 2021*, pages 906–918. ACM, 2021.

[41] Stuart Reid. Software fault injection: Inoculating programs against errors, by jeffrey voas and gary mcgraw, wiley, 1998 (book review). *Softw. Test. Verification Reliab.*, 9(1):75–76, 1999.

[42] Cedric Richter and Heike Wehrheim. Learning realistic mutations: Bug creation for neural bug detectors. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 162–173, 2022.

[43] Hossain Shahriar and Mohammad Zulkernine. Mutation-based testing of buffer overflow vulnerabilities. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2008, 28 July - 1 August 2008, Turku, Finland*, pages 979–984. IEEE Computer Society, 2008.

[44] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. Evolutionary improvement of assertion oracles. In *ESEC/FSE '20, USA, November 8-13, 2020*, pages 1178–1189. ACM, 2020.

[45] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. Generating accurate assert statements for unit test cases using pretrained transformers. In *IEEE/ACM AST@ICSE 2022, Pittsburgh, PA, USA, May 21-22, 2022*, pages 54–64. ACM/IEEE, 2022.

[46] Michele Tufano, Jason Kimko, Shiya Wang, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, and Denys Poshyvanyk. Deepmutation: A neural mutation tool. In *ICSE: Companion Proceedings*, ICSE '20, page 29–32, New York, USA, 2020. ACM.