Szegedi Tudományegyetem

Informatikai Intézet

Sérülékenységet tanúsító egységtesztek automatikus generálásának vizsgálata

TDK dolgozat

Készítette:
Isztin Martin
programtervező informatikus
III. évf. BSc hallgató

Témavezetők:
Dr. Antal Gábor
adjunktus

Dr. Bán Dénes
tudományos munkatárs

Szeged
2024

Tartalomjegyzék

Absztrakt

Egy szoftver fejlesztésének életciklusában fontos minőségbiztosítási szerepet játszik a tesztelés. Megfelelő tesztekkel a kód lefedettségének növelésén és a regressziók elkerülésén felül arról is megbizonyosodhatunk, hogy egy esetleges sérülékenység jelen van-e a szoftverünkben - és hogy azt egy potenciális javítás valóban javítja-e. Ám az ilyen tesztek elkészítése bonyolult, költséges és manuális folyamat.

Hogy segítsük mind a tesztelők, mind a biztonsági szakértők munkáját, ebben a dolgozatban az egyik legelterjedtebb nyelvi modell, a ChatGPT automatikus egységteszt generálási képességét kutatjuk a sérülékenységek szemszögéből. A VUL4J nevezetű, tanúsított CWE sebezhetőségekkel és hozzájuk tartozó javításokkal rendelkező Java projektgyűjtemény egy részhalmazán vizsgáljuk, hogy a GPT képes-e a javítás előtti és utáni kódállapot együttes ismeretében szintaktikailag és/vagy szemantikailag helyes egységteszteket generálni az adott sérülékenység kivédésének bizonyítékaként. Figyelmet fordítunk ezen belül a kódkontextus bőségének hatására, a GPT ön-hibajavítási képességének hatékonyságára és a generált tesztesetek szubjektív hasznosíthatóságára.

Eredményeink azt mutatják, hogy a GPT domain-specifikus előtanítás nélkül is szintaktikailag korrekt teszteseteket generál az esetek 50%-ában. És habár a javítások szemantikai helyessége csak az esetek 15%-ban volt automatikusan validálható, szubjektív kiértékelésünk alapján a GPT az esetek többségében olyan teszt sablont generál, ami minimális emberi finomhangolással teljes értékű sérülékenység igazolássá fejleszthető tovább. Tehát a kis mennyiségű rendelkezésre álló adat ellenére már ezek a korai eredmények is arra engednek következtetni, hogy a GPT sikerrel használható a sérülékenység-tesztelésben - ha nem is teljesen autonóm módon, de egy intelligens támogatói folyamat részeként mindenképp.

Kulcsszavak : generált egységtesztek, nyelvi modell, sérülékenység, CWE, kontextus szintek

1. fejezet

Bevezetés

A mai napig rengeteg olyan szoftver lát napvilágot a fogyasztói világban, amelyben sérülékenységek találhatók. Ezek mindaddig észrevétlenek maradnak, amíg valaki ki nem használja őket. Sajnos nem elég csupán szemmel ellenőrizni egy programkód hitelességét egy tapasztalt személynek sem, mivel akár olyan kódrészlet is okozhat sebezhetőséget, amely egy másik környezetben teljesen ártalmatlan. Az ilyen hibák kijavítása rengeteg idő és pénzügyi áldozattal jár. Egy szoftverben rejlő sebezhetőségek szerencsére hamar észrevehetők és kiküszöbölhetők megfelelő mértékű teszteléssel, azonban ezeknek a megírása szintén erőforrásigényes feladat.

Léteznek olyan adatbázisok, amelyek gyakori nyilvánosságra hozott sérülékenységeket dokumentálnak. A CVE (*Common Vulnerabilities and Exposures*) [1] valós rendszerek kiberbiztonsági sérülékenységeiről tartalmaz információkat. Az itt található bejegyzésekhez társul egy azonosító is, amely egy adott sérülékenységet reprezentál. Ezen sérülékenységek általánosabb kategorizálása a CWE (*Common Weakness Enumeration*) [2] azonosítókkal történik.

A mára már szinte bárkinek elérhető nagy nyelvi modellek széleskörűen elterjedtek produktivitásnövelő technológiaként és a modellek által nyújtott lehetőségek miatt a szoftverfejlesztés számos területén szintén használják. Számos kutatás számol be a feltörekvő technológia automatikus programjavító képességeiről, viszont az ezt próbára tevő egységtesztek generálásáról kevés információt tudunk.

A dolgozat célja a gyakorlatban felismert sebezhetőségekhez való egységteszt generálás, ezért a kiértékelés során a VUL4J [3] projektgyűjtemény részhalmazát használtuk,

amely számos Java nyelven íródott valós rendszer sérülékenységeiről tartalmaz információkat, többek között a sérülékenység javítását, a sérülékenységhez tartozó CVE és többnyire CWE azonosítót. További előnyei az adathalmaznak, hogy a biztosított környezettel könnyedén lehet alternálni a projekt sérülékeny és javított változatai között, elérhetők a sérülékenység javításával alkalmazott változtatások és a projektben eredetileg használt egységtesztek is.

A kiértékeléshez használt részhalmazunk 20 olyan példát tartalmaz, amelyet igyekeztünk úgy kiválasztani, hogy minél több sérülékenység fajtát lefedjen. Minden példát fokális kontextusokra bontottunk [4], amely négy szintből állt (0-3). A kutatás során az egyik legismertebbek nyelvi modellt használtuk, az OpenAI cég generatív előtanított transzformer modelljét, a GPT-t, ezen belül is a legfrissebb GPT-4 Turbo változatát. A bemenetként adott szöveghez (prompthoz) kizárólag egy szerepet (ami lehetővé teszi az LLM számára, hogy pontosabb, kontextusnak megfelelő összhangban álló válaszokat adjon, itt most a modell egy senior szoftvertesztelő) és a rendelkezésre álló kódot adtuk át. Az eredmények osztályozása két részre különíthető. Először egy automatizált környezetben végeztük el a kontextus legyártását, promptolást, a kapott válasz feldolgozását és a kiértékelését, ezt követően kézzel is felülbíráltuk a generált kódok szubjektív használhatóságát.

A kutatás a következő kérdésekre keresi a választ:

- **RQ1**: Hány százalékban kaptunk szintaktikailag helyes kódot?
- **RQ2**: Hány százalékban kaptunk szemantikailag is helyes kódot?
- **RQ3**: Milyen a szubjektív használhatósága a generált teszteknek?
- **RQ4**: Hogyan befolyásolja a kontextus a generálás pontosságát?

Eredményeink azt mutatják, hogy a GPT domain-specifikus előtanítás nélkül is szintaktikailag korrekt teszteseteket generál az esetek 50%-ában, amely bíztató eredmény annak tudatában, hogy semmilyen technikai háttérinformációt nem adtunk át a promptolás során. A tesztesetek szemantikai helyessége 15%-ban volt automatikusan validálható, ennek ellenére a szubjektív, kézi kiértékelésünk során a GPT az esetek többségében ez sokkal nagyobb arányban generált hasznos sablont a fejlesztők számára. Ez mutatja, hogy

a nagy nyelvi modellek kellő finomhangolással nagyon hasznos részét képezhetik ezen fontos munkafolyamatnak is.

A dolgozat felépítése a következőképpen alakul : a 2. fejezetben a témához kapcsolódó munkákat tárgyaljuk meg, majd a 3. fejezetben ismertetjük a kutatás módszertanát. A 4. fejezetben részletesen bemutatjuk a kutatás eredményeit, majd a speciális esetekről ejtünk szót. A 5. fejezetben a limitációkról és a jövőbeli lehetőségekről számolunk be, végül a 6. fejezet zárja le a dolgozatot.

2. fejezet

Kapcsolódó munkák

2.1. Nagy nyelvi modellek a sérülékenység analízisben

A nagy nyelvi modellek széleskörűen használtak a sérülékenység analízisben. A legjelentősebb iránya az APR (Automated Program Repair), amiről számos tanulmány szól. Quanjun Zhang és társai összevetették a VRepair transzfer-tanuló neurális hálózati modell APR képességeit számos nagy nyelvi modellel [5], köztük a CodeBERT-tel, UniXcoderrel és CodeGPT-vel. Fő eredményként fény derült a tényre, hogy a nagy nyelvi modellek 10.21% és 22.23% közti értékkel pontosabban hajtották végre a javításokat.

Egy másik tanulmány keretein belül Michael Fu és társai kifejezetten a ChatGPT-t használták a modell APR képességeinek felmérésére [6] a Big-Vul nevezetű adathalmazon, amely C++ nyelvben íródott projektek függvényeinek sérülékeny és javított változatait egyaránt tartalmazza, CVE, CWE azonosítókkal együtt, további egyéb technikai információkkal. A ChatGPT finomhangolások nélkül hátramarad a kódspecifikus modellekkel szemben.

A GPT adottságain túl Kamel Alrashedy és társa a CodeLlama nevezetű modellt is megvizsgálta [8] egy Python sebezhetőségi adatbázissal, visszajelzés vezérelt hibajavítással, miszerint a modell egy másik példánya véleményezi az eredeti példány válaszát, majd ezen vélemény alapján történik az eredeti példány válaszának finomhangolása. Ezzel a megközelítéssel 5-10%-kal jobb eredményeket értek el.

A témánkhoz releváns ötletként megemlíthető Chunqiu Steven Xia csapatának ötlete [15], amelyben a generált hibajavítás helyességét megadott tesztekkel validálják.

A nagy nyelvi modellek ilyen módú alkalmazhatóságának fényében mi a sérülékenységek témakörét vesszük célba, és mi is a ChatGPT segítségével, de mi javítások automatizálása helyett már létező javítások helyességét szeretnénk ellenőrizni a hozzájuk tartozó tesztesetek generálása által.

2.2. Nagy nyelvi modellek a teszteset generálásban

Egy tanulmány statikus metrikákat alkalmazva finomhangolta a modelleket, a magas minőségű egységtesztek generálására [7], miszerint egy alap modell 17%-ban szintaktikailag hibás, 31%-ban állítás (assertion) nélküli tesztet generál és az esetek 37%-ában nem hívja meg a tesztelendő metódust. A megerősítéses tanítás eredményeként felügyelt finomhangolással és az RLSQM (Reinforcement Learning from Static Quality Metrics) alkalmazása után ezek a számok lecsökkentek 1%, 5% és 10% környéki értékekre. A finomhangolás során, hozzánk hasonlóan dinamikus kontextushosszt használtak, ami akár az egész fájl szövege volt, egy egész osztály, csak a metódus fejlécek, vagy csak a metódus maga.

Michele Tufano és csapata szintén különböző fokális kontextusszintekkel kísérleteztek az egységteszt generálás [4] promptolása során. Ez a megközelítés sokkal hasonlóbb ahhoz az átadott dinamikus kontextushoz, amit mi választottunk. Kezdve kizárólag a fokális metódustól, hozzáadva szintenként az osztály nevét, a konstruktor fejléceit, a metódusok fejléceit és végül az osztály mezőit. A használt modell tanítása progresszíven történt. Történt egy angol nyelvű előtanítás, utána egy kódbázisú, végül egy teszteset generáláshoz kapcsolódó finomhangolás. A generálás egy fordítási folyamatnak van értelmezve, fokális kontextusról tesztesetre.

Egy empirikus tanulmányban szintén új kutatási dimenziót képezett a kontextus hatása a teszt generálására vonatkozóan, habár itt a kontextus máshogy játszik szerepet. A különböző szinteket a Java dokumentáció jelenléte és specifikussága definiálta [9]. Három szintje a következőképp épült fel : Javadoc nélküli kontextus, részleges Javadoc (használati példák nélkül) és teljes Javadoc implementáció nélkül. Az osztály, a javítást megelőző és a javított metódus mindig szerepel a bemenetben. A generált tesztek 40-70%-a nem okoz fordítási hibát, szubjektív kiértékelés során ez az arány sokkal nagyobb, 75-100%. 50-80%-ban helyes generálás történt és 70-90% volt a kódlefedettség. Egyéb érdekesség,

hogy előzetes munkálatok kimutatják, hogy a nagy nyelvi modellek jobban teljesítenek gyengén típusos nyelvek használata során.

Egy Meta cég által kiállított tanulmányban [10] nem generálnak konkrét "nyers" teszteket, hanem a már létező ember által írt teszteket próbálták feljavítani. 75%-ban szintaktikailag helyes javítás történt, 57%-ban sikeres tesztet generált, a kód lefedettsége pedig 25%-kal javult. Összesítésben a bemenetkét megadott teszt osztályok 10%-át javította fel és a modell ajánlásai később 73%-ban elfogadottnak minősültek a fejlesztők által.

Alok Mathur és csapata olyan megközelítést alkalmazott a T5 és GPT-3 modellekkel, hogy kizárólag teszt bemeneteket generálnak és/vagy releváns természetes szöveges leírást [14]. A fejlesztők leírnak néhány peremfeltételt vagy követelményeket és a modell kimenete releváns tesztesetek lesznek, amik kiértékelhetők a rendszer megfelelő végrehajtási útvonalaihoz.

Egy nagy nyelvi modellekkel való szoftvertesztelésre specializált kutatási felmérés [11] alapján senki se gondolkodott még az általunk használt irányban, bemeneti formátum szempontjából. Kizárólag a hibás függvényt adják át, elő- vagy utótagot, vagy leírást a javítás, vagy teszt generálásához, de senki sem próbálkozott mind a hibás és a javított kód átadásával a tesztgeneráláshoz. A Sugmin Kang és társai által kiállított tanulmányban [12] szereplő bemeneti formátum a hiba leírását adja meg a teszteset generáláshoz.

Habár mi is teszt eseteket generálunk, a fentiekkel szemben a mi kutatásunk nem csak egy adott funkcionalitás tesztelését célozza a nagyobb lefedettség érdekében, hanem egy adott sérülékenység előtte/utána állapotai alapján próbál kimondottan a sérülékenységet bizonyító tesztesetet előállítani, ami az sérülékeny állapoton hibát jelez, míg a javított változaton sikeresen lefut.

2.3. Sérülékenységekhez tartozó tesztesetek

A mi vizsgálatunkhoz leghasonlóbb irodalom a sérülékenységek és a tesztesetek metszetét tekinti. [13] az egységteszt generálás szintén úgy történik, hogy a bemenet tartalmazza a régi és új változatát a kódnak, de a tényleges generálás instrumentálás és útvonallefedettségen alapul.

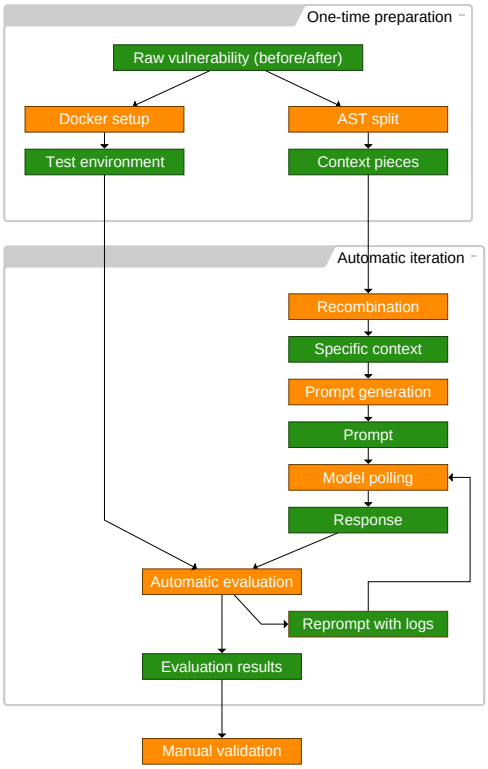
A mi kutatásunk is ebben a metszetben helyezkedik el, de a fentiekkel szemben mi a

napjainkban kibontakozó, és egyre biztatóbb eredményeket elérő nagy nyelvi modelljeit használjuk a sérülékenység bizonyítékául szolgáló teszteset generálására az instrumentáció és/vagy genetikus algoritmusok alkalmazása helyett.

3. fejezet

Módszertan

A kutatásunk folyamata öt lépésre osztható. Elsősorban kigyűjtöttünk 20 példát a kiértékeléshez, ezután a sérülékeny metódussal rendelkező osztályokat felbontottuk fokális kontextusszeletekre. Létrehoztunk egy promptot, amelyekbe a kiértékelés során beleágyazódott a megfelelő szintű kódkontextus. Lefuttattuk a többlépcsős kiértékelő szkriptet, végül manuálisan átvizsgáltuk a kiértékelés eredményeit.



3.1. ábra. A módszertan ábrázolása

3.1. Adatok kigyűjtése

A kiértékeléshez valós rendszerekben valaha fennálló sérülékenységeket gyűjtöttünk ki. Amire mindenképpen szükségünk volt az adathalmaz kiválasztása során, azok a sérülékeny metódusok, javításuk és egy környezet, amiben ezeket a metódusokat tesztelni tudjuk.

Ezeknek az igényeknek eleget tett az APR-rel kapcsolatos kutatásokhoz széleskörben használt VUL4J [3] adatbázis, amely számos tanúsított Java sérülékenységekhez tartalmaz valós példákat. Rengeteg hasznos információ áll rendelkezésünkre ezzel az adatbázissal, köztük az egyes projektekben szereplő sérülékenységi azonosítók, javítás előtti és utáni állapotok, GitHub commit hivatkozások és viszonyítási alapnak a PoV (proof of vulnerability) egységtesztek az egyes sérülékenységekhez. Társul hozzá egy Docker környezet is, amely nagyban megkönnyíti a projektek lefuttatását és ellenőrzését.

Az adatbázisban szerepelnek olyan sebezhetőségek, amely kijavításához egy, vagy több metódus javítása szükséges, ami akár több osztály módosítását is igényelheti. A modell kontextusból való kifutás veszélyének csökkentéseként úgy döntöttünk, hogy csak olyan példákat használunk a kiértékelés során, amelyekben a sérülékenység javítása csak egyetlen osztályt, metódust érint.

Hogy minél szélesebb spektrumban mérhessük a GPT egységteszt generálási képességeit, igyekeztünk minél több, különböző sérülékenységgel rendelkező példát használni.

3.2. Fokális kontextusszintek és a promptolás

Az egységteszt generálásához, a megfelelő mértékű kontextus definiálása elengedhetetlen, a természetes szöveg mellé szükségünk van a tesztelendő metódus mellékelésére. Ehhez a kigyűjtött példák összes sebezhető osztályából kontextusdarabokat [4] gyártottunk, amelyek segítségével az automatizált kiértékelés során könnyedén szabályozhattuk, hogy mekkora kontextussal szeretnénk az adott mérést elvégezni.

Kisebb adathalmazokon való tesztelések után a szintek elosztásában a következőképp döntöttünk:

- **L0**: tartalmazza az osztály csomagdeklarációját, az osztály deklarációját és a sebezhető metódust.
- **L1**: tartalmazza a **L0**-ban felsorolt elemeket és az osztály konstruktorainak fejlécét, ha van.
- **L2**: tartalmazza a **L1**-ben felsorolt elemeket és az osztályban deklarált metódusainak fejlécét.
- **L3**: tartalmazza a **L2**-ben felsorolt elemeket és az osztályban deklarált mezőket.

A kiértékelés során minden szintből lezajlott egy futtatás.

Ahogy az előbb is említettem, a generáláshoz megfelelő méretű és elegendően specifikus kontextusra van szükségünk. Első lépésként promptolási referenciákat gyűjtöttünk előzetes releváns szakirodalmakból [11, 13], majd a promptolással való kísérletezés következett.

Igyekeztünk minél optimálisabb és lényegre törőbb bemeneti szöveget definiálni, ezért a kísérleti futtatásokból kinyert tapasztalatok alapján formáltuk meg a végleges promptot.

A hivatalos OpenAI dokumentációja szerint[1] egy válasz generálása során, ha specifikálunk a GPT-nek saját szerepkört, akkor könnyebben megérti a hozzáadott kontextust, amivel szemben áll. Ellenkező esetben a generikus "segítőkész asszisztens" szerepet kapja meg. Ez a mi promptunkban az átadott bemenet elejét képezi. Ezek után ismertetjük a feldolgozandó bemenet jellemzőit, majd a feladatot, amin a modell dolgozni fog.

A leggyakrabban előforduló hiba a GPT által generált egységtesztekben, az általa írt kódban használt függvények, osztályok importálásának hiánya.

A kiértékelésre szánt példák jelentős része elavult Java verzióval kerültek fordításra, ezért többször is előfordult, hogy olyan kódot generált a modell, ami nem volt kompatibilis ezekkel a változatokkal. Ezért úgy döntöttünk, hogy külön figyelmeztetjük a modellt a promptolás során ezekre a részletekre. Az osztály nevek egységessége is problémát okozott az automatikus kiértékelés során, ezért a kiértékelés során ezeknek a formátumát is definiáltuk a promptban.

[1] `https://platform.openai.com/docs/guides/gpt/chat-completions-api`

Ezután következett maga a feldolgozandó bemenet, a forráskód az előző fejezetben említett fokális kontextusként megadva. Először a sérülékenységgel rendelkező metódus, körítve a kontextusszintnek megfelelő hozzáadott adatokkal, ezt követően pedig a javítást tartalmazó metódus, de ez már a kontextus többi része nélkül, annak érdekében, hogy a feldolgozandó prompt mérete ne legyen feleslegesen túl nagy.

Végezetül érzelmi stimulációt alkalmazunk a modellen, amiben közöljük, hogy a feladat megoldása nagyon fontos számunkra. Ez az eljárás jobb eredményekhez vezet. [17]

A végső prompt, amellyel a kiértékelés összes példája lezajlott a következőképp állt össze:

```
You are a senior software tester and a cyber security specialist.
You will be given the source code of a Java class where you will find the context of↩
   a vulnerable method before and after the patch.
Your task is to create a unit test that triggers the vulnerability and fails before ↩
  the patch and passes after it. The class' name should be the name of the class ↩
  appended with the string "Test".
Use simple Java language features in the generated test!

{focal context of the vulnerable code}

The method after patching the vulnerability:

{patched method of vulnerable code}

It is very important for me, please create the unittest based on your best knowledge↩
   in the given context.
```

Kódrészlet 3.1. A végleges prompt

Az automatikus kiértékelés során felvetettük az ötletet, hogy érdemes lenne több iterációt végrehajtani a véletlenszerű eredmények elkerülése, a generálás konzisztenciájának vizsgálata érdekében. Az iterációk helyett korlátozott számú visszakérdezéses finomhangolás mellett döntöttünk, abban az esetben, ha a generált válasz nem felel meg a megfelelő szempontok alapján kiállított kritériumoknak. Ezekhez a visszakérdezésekhez különböző promptok társulnak. Mivel a visszakérdezés ugyanabban az üzenetváltási környezetben történik, ezért a modell nem felejti el az előzőlegesen átadott információkat, így ezeknél a promptoknál nem szükséges a szerepkör és egyéb technikai információ ismertetése.

Ebből kiindulva csupán egy egyszerű visszajelzést adtunk bemenetként a GPT-nek,

az éppen fennálló kritérium megsértéséről a tesztelés futtatása során keletkezett naplófájl társításával.

Három elfogadást megtagadó helyzet állt fent, amelyekhez társult prompt:

- **BEFORE_SUCCESS** - A sérülékeny metódus sikeres tesztet produkál.
- **AFTER_FAILURE** - A javított metódus sikertelen tesztet produkál.
- **ERROR** - A teszt futtatása fordítási hibát okoz.

3.3. Kiértékelő környezet

A kutatás során a legnagyobb szerepet az automatizált kiértékelőnk játszotta. A generálás a GPT legújabb elérhető modelljével történt, a legfrissebb GPT-4 Turbo-t használtuk. Nagyban megkönnyítette az adathalmazunkon végzett mérések folyamatát, hiszen a projektenként szétbontott kontextusszeletek és a projektek elérési útvonalának segítségével, a GPT API-val való kommunikáción át, a generált válasz lefuttatása és az eredmény kiértékelése teljesen automatikus volt. Ezen felmérés egy néhány lépcsős folyamatba foglalható.

Kontextus legyártása

Elsősorban az előre definiált fokális kontextusszint megadásával megépítette azt a Java fájlt, amely szintaktikailag is helyesen strukturálva tartalmazta a szintnek megfelelő mennyiségű adatot az adott sérülékenységről.

Kommunikáció a GPT-vel

A legyártott Java fájl került átadásra a GPT API-val kommunikáló komponensnek, amely a fentebb kifejtett promptot tette egésszé. A modellel való kommunikáció során visszakapott válaszból kinyeri a Java kódot, amelyet elment egy fájlba és elhelyezi a projekt tesztkörnyezetébe.

Futtatások

A futtatás a VUL4J adatbázishoz járó Docker környezetet inicializálva, mindenféle fordításhoz szükséges kompatibilitási igényt kielégítve történik. A csökkentett futtatási idő érdekében kizárólag azt a tesztet futtatja, amelyet a GPT generált.

A futtatás először a projekt sérülékenységgel rendelkező változatára fut le, és a tesztelés naplófájlját a szkript eltárolja és felhasználja az adott generálás kiértékelésére. A sérülékeny változat tesztelésének befejezése után, ugyanez a folyamat lezajlott a javítással rendelkező változat esetén is.

Megvizsgálja, hogy az adott teszt sikeresen lefutott-e és amennyiben az aktuális állapotban (sérülékeny vagy javított) nem elvárt értéket kapunk, elkezdődik a visszakérdezéses generálás.

A visszakérdezés többször is megtörténhet, egészen addig amíg nem történik három egymást követő fordítási hiba (ekkor feltételezzük, hogy a modell nem fogja tudni megoldani a szintaktikailag helyes tesztgenerálást), vagy öt olyan generálás, ahol nem történt fordítási hiba, de nem az elvárt eredményt kaptuk. Azaz a sérülékeny állapot átment a generált teszten, vagy a javított változat bukott meg.

Fontos megjegyezni, hogy a kiértékelés alatt minden példához kapcsolódó generálás esetében a legjobb generációt vettük figyelembe. Ez csupán annyit jelent, hogy ha egyszer is történt nem fordítási hibás generálás, akkor a kiértékelő azt a naplófájlt külön megtartotta és a végső eredményekbe ez az eset került bejegyzésre.

Futási eredmények

Tehát a bejegyzett eredmények példánként két részre oszlanak. Ahogy korábban említettük, a futtatás és tesztelés először a sérülékeny változattal történik meg (before patch), majd utána a javítottal (after patch). Mind a két esethez három lehetséges érték társul.

- **ERROR**: A generált teszt fordítási hibát tartalmaz
- **SUCCESS**: A generált teszt az adott állapotban sikeresen lefutott és átment
- **FAILURE**: A generált teszt az adott állapotban rossz várt értékkel tért vissza

A 4. fejezetben tárgyalni fogunk néhány olyan esetről, ahol felmerült olyan helyzet, ahol ezek a visszajelzések önmagukban nem tartalmazzák a teljes tényszerű eredményt, emiatt is volt fontos a következő szekcióban tárgyalt emberi felmérés.

3.4. Manuális kiértékelés

A teljesen automatikus és szigorú validálás mellett gondoskodtunk a szubjektív, emberi kiértékeléséről is a generált teszteknek. A kézi kiértékelés során több fejlesztő is dolgozott a validáción, az emberi hibák csökkentésének érdekében. Az egységtesztek elkészítéséhez egy fejlesztőnek is rengeteg releváns háttérinformációval kell rendelkezzen már ahhoz is, hogy egyáltalán olyan tesztet készítsen egy adott programhoz, ami futtatható. Ezért döntöttünk úgy, hogy a tesztek hasznosságának felmérése érdekében megtekintjük az egyes tesztek felépítését és kézzel is elbíráljuk szemantikai helyességét.

Két esetre bontottuk a kézi kiértékelés véleményezését:

– Szemantikailag rossz

– Szemantikailag jó

Természetesen a szigorú validáláskor helyes kimenetet produkáló eseteket is ellenőriztük kézzel.

4. fejezet

Eredmények

A 3.3. szakaszban említettük, hogy minden példára, akár több mint négyszer, történt generálás a modell által és mindig az utolsó legjobb eredményt vettük figyelembe. A kiértékelés részletes eredményeit a 4.1. táblázat ábrázolja. Szintén említésre került, hogy az eredmények automatikus és manuális, kézi validáláson is átestek. Összesen 80 darab eredménypár született az automata kiértékelés során, ami magába foglal változó mennyiségű visszakérdezéses generálást is.

	L0			L1			L2			L3		
Példa	Before	After	Manuális	Before	After	Manuális	Before	After	Manuális	Before	After	Manuális
VUL4J-5	ER	ER	OK	ER	ER	NO	FA	FA	OK	ER	ER	OK
VUL4J-10	ER	ER	NO	FA	FA	OK	ER	ER	NO	FA	FA	OK
VUL4J-12	SU	SU	OK	SU	SU	NO	FA	FA	NO	ER	ER	NO
VUL4J-17	FA	SU	OK	FA	SU	OK	FA	SU	OK	FA	SU	OK
VUL4J-22	FA	FA	OK	FA	FA	NO	FA	FA	OK	FA	FA	OK
VUL4J-24	FA	FA	NO	ER	ER	NO	SU	SU	OK	FA	FA	OK
VUL4J-34	SU	SU	OK	SU	SU	OK	SU	SU	OK	SU	SU	OK
VUL4J-41	FA	FA	OK	ER	ER	OK	ER	ER	OK	FA	FA	OK
VUL4J-43	SU	SU	OK	SU	SU	OK	SU	SU	OK	SU	SU	OK
VUL4J-44	FA	FA	OK	FA	SU	OK	SU	SU	OK	FA	SU	OK
VUL4J-45	FA	FA	NO	FA	FA	NO	SU	SU	NO	FA	FA	NO
VUL4J-47	ER	ER	NO	SU	SU	OK	SU	FA	OK	ER	ER	NO
VUL4J-48	SU	SU	NO	ER	ER	OK	SU	SU	OK	ER	ER	NO
VUL4J-50	ER	ER	NO	SU	SU	OK	SU	SU	OK	SU	SU	OK
VUL4J-52	ER	ER	OK	ER	ER	OK	ER	ER	OK	ER	ER	OK
VUL4J-53	ER	ER	NO	ER	ER	NO	ER	ER	NO	ER	ER	NO
VUL4J-54	ER	ER	OK	ER	ER	OK	ER	ER	OK	ER	ER	OK
VUL4J-57	ER	ER	NO	ER	ER	NO	ER	ER	OK	ER	ER	NO
VUL4J-62	SU	SU	NO	SU	SU	NO	SU	SU	NO	SU	SU	OK
VUL4J-77	FA	SU	OK	FA	SU	OK	FA	SU	OK	ER	ER	NO

4.1. táblázat. A futtatások eredményei

4.1. Szintaktikai elemzés

A szintaktikai elemzés során az volt a kérdés, hogy a GPT képes-e mindenféle emberi beavatkozás nélkül olyan egységtesztet generálni, amely univerzálisan alkalmazható egy tesztkörnyezetben, feltéve, hogy a modellnek az utasítás részeként egyszerű Java nyelvi elemek használatát kértük a válaszában. Előfordultak olyan esetek, ahol a futtató környezet extra jelölést igényelt a teszt lefuttatása érdekében. Ezen kontextus hiánya miatt voltak olyan generálások, ahol a tesztet a futtató környezet nem érzékelte egységtesztként, így az nem futott le, de a projekt fordítási folyamatába beletartozott. Az utóbb említett tényező alapján az ilyen eseteket is elfogadtuk szintaktikailag helyes eredménynek.

Néhány futtatásnál, ahol a sérülékeny és javított változat egyaránt **FAILURE** választ produkált, nem feltétlen csak a teszt eredménye volt hibás. Fennállt az esete annak a jelenségnek is, hogy maga a fordítás nem hiúsult meg, de a tesztben használt szimbólumok

nem voltak elérhetők a megfelelő importálások nélkül, amiket a modell esetleg elfelejtett, vagy egyszerűen nem is léteztek. Ezeket a manuális kiértékelés során ellenőriztük és felülbíráltuk. Az ilyen eset a 4.1. eredmények táblázatban piros színnel van jelölve.

RQ1: Hány százalékban kaptunk szintaktikailag helyes kódot?
A felülbírált eseteket kivéve a statisztikából, **L0** és **L1** fokális kontextussal 11 esetben, azaz 55%-ban szintaktikailag helyes kódot generált a GPT. Továbbá az **L2**-vel futó generálás 13, azaz 65%-os és az **L3** 10, tehát 50%-os arányban produkált pozitív eredményt.

4.2. Szemantikai elemzés

A szintaktikai elemzés önmagában nem elég ahhoz, hogy eldöntsük a generált tesztek helyességét. Ebben a szekcióban kizárólag azokat az eseteket vizsgáljuk, amelyek eleget tettek az automatikus validálás kritériumainak, azaz a javítás előtti állapotra a teszt elbukott, (**FAILURE**), a javítás utáni állapot során pedig sikeres futás történt (**SUCCESS**). Az összképet nézve ez összesen 9 alkalommal (15%) történt meg.

Ezek a generált tesztek, ha csak egy kicsit is, mindig eltérnek a hivatalos VUL4J adathalmazzal járó, példákban szereplő egységtesztektől. Ez arra enged következtetni, hogy a helyesen megoldott példák valószínűsíthetően nem szerepeltek ebben a formában a GPT tanítási halmazában, amely tovább erősíti a született eredmények ígéretességét.

Az eredmények védelme érdekében a kézi kiértékelés során ellenőriztük, hogy ezek a generálások valóban helyesek-e.

RQ2: Hány százalékban kaptunk szemantikailag is helyes kódot?
Összesen 9 (15%) generált teszt volt szemantikailag is helyes. Fokális kontextusszintek szerint az **L0**, **L2** és **L3** 2 (10%), az **L1** érdekes módon 3 (15%) pozitív eredményt produkált.

4.3. A generálás szubjektív hasznossága

A tesztek szubjektív használhatóságánál a főbb értékelési szempontok a következők voltak:

- A generált teszt releváns a példában levő sérülékeny kódhoz és annak javításához.
- Azt a metódust teszteli, ami a modell feladataként lett meghatározva.
- Legfeljebb minimális emberi finomhangolással (például a megfelelő csomagok importálásával) használható.
- Nem próbálja meg felülírni a tesztelendő metódust saját kóddal.

A szubjektív kiértékelés során nem elfogadott példa, az **L0** futtatás során a *VUL4J-24* példához olyan tesztet generált a modell, amelyben már saját maga próbálta meghatározni a tesztelendő osztály metódusának viselkedését, ahelyett, hogy a teszten javított volna a hiba és az előtte ismertetett kontextus alapján. Ebben a jelenségben a visszakérdezéses generálás is szerepet játszik, erről a 4.5. szakaszban fogunk tárgyalni.

Továbbá, néhány esetben a kód értelmezésével is akadtak problémái a modellnek. Az **L1**-es generálásban a *VUL4J-12* példát félreértette, a példában szereplő javítás egy végtelen ciklus kiküszöbölését tartalmazza, ellenben a teszt egy *long* típusú számértéket *integerre* való típuskasztolás segítségével akar összehasonlítani egy másik integerrel. Azonban ezek sem véletlenszerű utasítások, hiszen relevánsak a promptban specifikált kódhoz, viszont a sérülékenység teszteléséhez nem tesznek eleget.

```
@Test
public void testExtendMethodVulnerability() throws Exception {
    ...
    JpegDecoder jpegDecoder = new JpegDecoder();
    // Call the private extend method using reflection
    int result = (Integer) extendMethod.invoke(jpegDecoder, v, t);
    // Hypothetical patched logic that accounts for integer overflow
    long expected = ((long) v) + ((v < (1 << (t - 1))) ? ((-1L) << t) + 1 : 0);
    int expectedInt = (int) expected;
    assertEquals("The extend method should behave as expected after the patch", ↩
  expectedInt, result);
}
```

Kódrészlet 4.1. Az L1 VUL4J-12 példára generált teszteset részlete

```
  ...
  int vt = (1 << (t - 1));
-  while (v < vt) {
+  if (v < vt) {
       vt = (-1 << t) + 1;
       v += vt;
  }
  ...
```

Kódrészlet 4.2. A VUL4J-12 példához tartozó javítás

Rengeteg hasznos teszt került generálásra, erre egy példa az **L3**-ban található *VUL4J-5* generálása.

```
@Test
public void testExpandVulnerability() {
    ...
    // Vulnerable entry trying to write outside of target directory
    ArchiveEntry vulnerableEntry = new ArchiveEntry("../outsidetarget.txt", false);
    ...
    try {
        // This should fail before patch and pass after patch
        expander.expand(supplier, writer, targetDirectory);
        fail("The test should throw an IOException before the patch as it attempts ↩
  to expand outside of the target directory.");
    } catch (IOException e) {
        // Expected exception
        String expectedMessage = "expanding ../outsidetarget.txt would create file ↩
  outside of " + targetDirectory;
        assertEquals(expectedMessage, e.getMessage());
    }
}
```

Kódrészlet 4.3. Az L3 VUL4J-5 példára generált teszteset lényeges részei

Habár szintaktikailag nem megfelelő a teszt, ennek ellenére kifejezetten jól értelmezte a modell a feladatát és valóban letesztelte a javítás jelenlétét. A sérülékeny kódban lehetőség volt egy útvonal átjárás sérülékenység kihasználására, miszerint a "../fájl" bemenet átadása visszalép egy szintet a mappaszerkezetben, ami már nem biztos, hogy a working directory része, ezzel veszélyeztetve rengeteg olyan fájlt, amihez a szerver környezetén

kívül senkinek sem lenne külső hozzáférése.

Másik eset egy hasznos generálásra, az **L2**-ben szereplő *VUL4J-52* példához készült teszt. A kódban fennálló *helytelen jogosultságkezelés* sérülékenység javítása érdekében implementáltak egy metódust, ami a sandbox környezetbe bekerülő kódban megtiltja a *finalize* függvény felülírását egy *SecurityException* kivételdobás kíséretében.

```
@Test
public void ↩
  testForbidIfFinalizer_withUnsafeFinalizeMethod_ShouldThrowSecurityException() {
    thrown.expect(SecurityException.class);

    MethodNode unsafeFinalizeMethod = new MethodNode(
            "finalize",
            Opcodes.ACC_PUBLIC,
            ClassHelper.VOID_TYPE,
            Parameter.EMPTY_ARRAY,
            ClassNode.EMPTY_ARRAY,
            null
    );

    classNode.addMethod(unsafeFinalizeMethod);

    transformer.transform(sourceUnit, null, classNode);
}
```

Kódrészlet 4.4. Az L2 VUL4J-52 példára generált teszteset

A generált teszt valóban leteszteli azt az eshetőséget, hogy a környezetbe egy ilyen függvény kerül, és el is várja a megfelelő kivétel keletkezését, ami a sérülékeny kódban nem fog megtörténni, ellenben a javítottal.

RQ3: Milyen a szubjektív használhatósága a generált teszteknek?

A fentebb tárgyalt szempontok alapján a generált tesztek 50%-át hasznosnak ítéltük. Ezek egy fejlesztő számára akár egy használható vázat tud biztosítani egy sérülékenységet tanúsító egységteszt elkészítésében.

Eredményeink jól mutatják, hogy egy bizonyos szintig látványos javuláshoz vezet a hozzáadott kódkontextus.

4.4. Kontextus hatása a pontosságra

Habár a 4.2. szekcióban szereplő eredmények alapján a különböző szintű kontextusokra való futtatás nem mutatott különösebb elváltozást, néhány megállapítás született a kutatás során. Michele és munkatársai [4] az esetek közel 40%-ában pozitív eredményeket tudtak produkálni, viszont mindezt egy BART alapú modellen, kódbázisú előtanítással, amihez tartozott egy teszteset generálási finomhangolás.

A manuális kiértékelés alapján megállapítható, hogy a modell sokkal tudatosabb, kevesebb hibalehetőséggel rendelkező tesztesetet ír a konstruktorok felépítésének tudatában. Például **L0**-t használva a GPT összekeverte a publikusan elérhető Java *FileItem* osztályt a példában szereplő *DiskFileItem* osztállyal. A konstruktor ismertetésével az **L1**-ben már egy szintaktikailag megfelelő tesztet generált a modell.

```
...
  @Before
  public void setUp() {
    factory = new DiskFileItemFactory();
  }
  @Test(expected = IOException.class)
  public void testReadObjectWithInvalidRepositoryPath() throws Exception {
      ...
      File invalidRepository = new File("/some/repository\0");
      diskFileItem.setRepository(invalidRepository);
      ...
  }
```

Kódrészlet 4.5. Az L0-ban szereplő teszt kódja

```
  ...
  @Before
  public void setUp() {
      // Set up a valid repository directory for DiskFileItem
      repository = new File(TEMP_DIR, "diskFileItemTest");
      repository.mkdir();

      diskFileItem = new DiskFileItem(FIELD_NAME, CONTENT_TYPE, IS_FORM_FIELD, ↩
    FILE_NAME, SIZE_THRESHOLD, repository);
  }
  ...
```

Kódrészlet 4.6. Az L1-ben szereplő teszt kódja

Az 4.1. szakaszban tárgyalt eredmények alapján a legjobban teljesítő kontextusszint az **L2** volt, amely 13 szintaktikailag megfelelő generálást produkált.

RQ4: Hogyan befolyásolja a kontextus a generálás pontosságát?
A tesztek szubjektív kiértékelései alapján bizonyos mennyiségű extra kontextus hogyan segít a modellnek a feladat precízebb megoldásában. Az **L0** kontextusszint használatakor 11 esetben (55%) ítéltük szemantikailag hasznosnak a tesztet, **L1** esetén 12 esetben (60%), **L2** esetén 15 (75%), **L3**-ban pedig 13 alkalommal (65%).
Eredményeink jól mutatják, hogy egy bizonyos szintig látványos javuláshoz vezet a hozzáadott kódkontextus.

4.5. Visszakérdezéssel kapcsolatos tapasztalatok

Mint már említettük, a megközelítésünk egyik pillére a modelltől való visszakérdezés volt. Ez számos helyzetben segítette a generálás alakulását, hiszen a modell interaktívan visszanyerte az általa írt kódhoz tartozó naplófájl tartalmát is. Az **L2**-ben szereplő *VUL4J-77* példa, első generálásra **FAILURE** - **FAILURE** eredménypárral tért vissza, ezért megtörtént a visszakérdezés. A visszakapott kód a hiba alapján lett finomhangolva a GPT által, ez végül egy teljes, pozitív eredményt produkált.

```
It's important to note that with the SafeConstructor, the YAML parser should not ↩
  throw a SecurityException but rather a YAMLException, ConstructorException, or ↩
  another related exception when encountering malicious YAML content...
...
@Test
  public void testReadYamlTreeVulnerability() {
    ...
    assertTrue("Expected a YAML parser exception due to patched vulnerability",
                    e instanceof org.yaml.snakeyaml.error.YAMLException || // ↩
  change this if needed
                    e instanceof org.yaml.snakeyaml.constructor.↩
  ConstructorException); // change this if needed
  }
  ...
```

Kódrészlet 4.7. Az L2 VUL4J-77 példára generált válasz (részlet)

Néhány esetben azonban a modell teljesen "elvesztette a fonalat", és a tesztek finomhangolását olyan irányba vitte, ami nem volt releváns a kódban szereplő hiba kijavításához. Az **L1**-ben futtatott *VUL4J-62* példához először létrehozott teszt még nem rendelkezik feladatkörbe nem illő kód generálással. Fontos megjegyezni, hogy néhány példán való tesztfuttatáshoz szükséges lett volna, hogy a tesztosztály leszármazzon a TestCase osztályból, különben az ilyen projekteknél használt plugin nem ismerte fel a tesztet. Ennél a példánál is ez a szituáció állt fenn, ami azt jelenti, hogy a naplófájlok csupán annyi üzenetet tartalmaztak, hogy nem futott le egy teszt sem, így a buildelési folyamat sikeresen lezajlott. Ezek az esetek mindig **SUCCESS** - **SUCCESS** eredménypárt alkottak, ami azt jelenti, hogy megtörtént a **BEFORE_SUCCESS** prompt által történő visszakérdezés.

Ezeknél az eseteknél a GPT csak találgatott és sosem tudta megoldani a tesztek futtatását.

5. fejezet

Limitációk

A kutatás során fennálló limitációk függvényében elsősorban megjegyzendő, hogy csak egyetlen nagy nyelvi modell egységteszt generálási képességeit vizsgáltuk, a GPT-4-ét. A továbbiakban érdemes lehet más modelleket is vizsgálni, akár olyat is, ami kifejezetten kódspecifikus, például a CodeLlama. Nem alkalmaztunk előtanítást vagy érdemleges finomhangolást. A 4.5. szakaszban tárgyalt inkonzisztenciák elkerülése érdekében célszerű lehetne Michele és társai [4] mintájára egyes modelleket kód alapú, majd tesztgenerálási előtanításban részesíteni. A prompt szerkezetileg jól strukturált volt, ám nem tartalmazott extra adatot és információt a javítás elősegítéséhez. Érdemes lehet nagyobb hangsúlyt fektetni a prompt engineering részére is a jövőben, például a CWE információk promptba való integrálásával.

Habár az általunk használt adathalmaz, a VUL4J [3] egy gondosan összeválogatott sérülékenységek gyűjteménye, nem szabad eltekintenünk a ténytől, hogy kézzel lett összeállítva. Ez csupán annyit jelent, hogy a sebezhetőség teljes spektrumát nem feltétlen tartalmazza. A jövőbeli kutatás során érdemes lehet nagyobb, szélesebb választékú adathalmazt felhasználni, a széleskörű felmérés érdekében.

Figyelembe kell vennünk, hogy a szubjektív vélemény, a használhatóság megítélése függ az értékelést elvégző személyek tapasztalataitól. Ezen limitáció csökkentésére a kiértékelésben résztvevők egymás munkáját az egyes példákhoz kapcsolódó megjegyzéseket értelmezve és elemezve ellenőrizték.

A GPT nemdeterminisztikus természetű, ezért az eredmények gyártása során sosem kaptunk vissza kétszer ugyanolyan választ egy adott példára.

6. fejezet

Konklúzió

Kutatásunk bemutatja a GPT-4 egységteszt generáló képességeit a sérülékenységek kontextusában, valós környezetben. Előzetes irodalmakkal ellentétben, az általunk alkalmazott módszerben egymás mellé tesszük a sebezhető és a javított metódust, amely nagy nyelvi modellekkel történő munka esetén nem megszokott eljárás. Eredményeink alapján látható, hogy a nagy nyelvi modellekkel történő egységtesztek generálása tud hasznos segítséget adni a valós életbeli problémák megoldásához.

Felméréseinkből a következő megállapításokat tehetjük:

1. A GPT-4 egységteszt generálási képessége előtanítás nélkül közel sem tökéletes, de az eredmények alapján ígéretes.

2. Ha nem történik szintaktikailag helyes generálás, jó eséllyel minimális emberi finomhangolással helyes tesztesetet tudunk képezni.

3. A különböző mennyiségű információt hordozó kontextusszintek valóban hoznak változást a generálás minőségére.

4. A visszakérdezések, amelyek elegendő információt tartalmaznak a hiba kijavításához, inkább a helyes irányba terelik a generálást, mint a rossz irányba.

Összegzésként, a nagy nyelvi modellek által történő egységteszt generálás egyáltalán nem haszontalan irány, azonban a teljesítmény javításához további kutatások szükségesek (például az 5. fejezetben felsorolt célok megvalósításával).

Irodalomjegyzék

[1] Common Vulnerabilities and Exposures. `https://cve.mitre.org/`, 2023

[2] Common Weakness Enumeration. `https://cwe.mitre.org/`, 2023

[3] 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*, 2023

[4] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, Neel Sundaresan. *Unit Test Case Generation with Transformers and Focal Context*, 2021

[5] Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, Zhenyu Chen. *Pre-trained Model-based Automated Software Vulnerability Repair: How Far are We ?*, 2023

[6] Michael Fu, Chakkrit (Kla) Tantithamthavorn, Van Nguyen, Trung Le. *ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We ?*, 2023

[7] Benjamin Steenhoek, Michelle Tufano, Neel Sundaresan, Alexey Svyatkovskiy. *Reinforcement Learning from Automatic Feedback for High-Quality Unit Test Generation*, 2023

[8] Kamel Alrashedy, Abdullah Aljasser. *Can LLMs Patch Security Issues ?*, 2024

[9] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, Vinícius Carvalho Lopes. *An Empirical Study of Using Large Language Models for Unit Test Generation*, 2024

[10] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, Eddy Wang. *Automated Unit Test Improvement using Large Language Models at Meta*, 2024

[11] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, Qing Wang. *Software Testing with Large Language Models: Survey, Landscape, and Vision*, 2024

[12] Sungmin Kang, Juyeon Yoon, Shin Yoo. *Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction*, 2023

[13] Kunal Taneja, Tao Xie. *DiffGen: Automated Regression Unit-Test Generation*, 2008

[14] Alok Mathur, Shreyaan Pradhan, Prasoon Soni, Dhruvil Patel, Rajeshkannan Regunathan. *Automated Test Case Generation Using T5 and GPT-3*, 2023

[15] Chunqiu Steven Xia, Yuxiang Wei, Lingming Zhang. *Automated Program Repair in the Era of Large Pre-trained Language Models*, 2023

[16] Xing Hu, zirui Chen, Xin Xia, Yi Gao, Tongtong Xu, David Lo, Xiaohu Yang *Exploiting Library Vulnerability via Migration Based Automating Test Generation*, 2023

[17] Cheng Li, Jindong Wang, Yixuan Zhang, Kaijie Zhu, Wenxin Hou, Jianxun Lian, Fang Luo, Qiang Yang, and Xing Xie. *Large language models understand and can be enhanced by emotional stimuli*, 2023