

TDK dolgozat

Készítette:

Bogenfürst Bence

Szegedi Tudományegyetem Természettudományi és Informatikai Kar

Informatikai Intézet

Szoftverfejlesztés Tanszék

A GPT-4 automatikus programjavító képességének vizsgálata valós sérülékenységek felhasználásával

Készítette:

Bogenfürst Bence

mérnökinformatikus

III. évf. BSc hallgató

Témavezetők:

Dr. Antal Gábor

adjunktus

Ságodi Zoltán

PhD hallgató

Szeged

2023

Absztrakt

A szoftverhibák és sérülékenységek felismerése, azok javítása kihívást jelentő feladat. A bonyolult rendszerekben számos hiba rejtőzhet, amelyek észrevétlenek maradhatnak mindaddig, amíg valaki ki nem használja őket. A hibák forrása legtöbbször egyszerű, más környezetben ártalmatlan kódrészletek, így a tapasztalt szem is könnyedén elsiklik felettük. A nagy nyelvi modellek megjelenése új távlatokat nyitott az adatok feldolgozásában, új lehetőségeket ígérve többek között a szoftverfejlesztés, ezen belül pedig az automatikus programjavítás terén is. Jelenleg azonban nem ismerjük pontosan ezen rendszerek teljesítményét, robusztusságát, megbízhatóságát, hiszen eddig csupán kis méretű, mesterségesen összeállított adathalmazok segítségével folytak kutatások, amelyek nem tükrözik a valós rendszereket.

A dolgozatban szisztematikusan vizsgáljuk meg egy népszerű nagy nyelvi modell, a GPT-4 automatikus programjavító képességét. Ehhez a valós sérülékenységeket tartalmazó VUL4J adatbázist használjuk, amely számos nyílt forráskódú ipari Java rendszerekből tartalmaz sérülékenységet. A modell csupán az eredeti hibás forráskódot, az azt tartalmazó projekt rövid leírását, valamint a sérülékenység típusát kapta bemenetként. Az eredményeket manuálisan és a VUL4J adatbázisban található egységtesztek segítségével értékeljük ki. A GPT-4 modell hibátlan, módosítás nélkül alkalmazható megoldást 12 alkalommal nyújtott a megvizsgált 46 esetből. További 5 esetben a szöveges magyarázat nagy segítséget nyújthat egy fejlesztő számára a sérülékenység kijavításában (annak ellenére, hogy a visszaadott kód hibás volt).

Tartalomjegyzék

1. Bevezetés	2
2. Kapcsolódó munkák	5
3. Módszertan	8
3.1. Adatgyűjtés	8
3.2. Metódusok kigyűjtése	9
3.3. Promptok írása és fejlesztése	10
3.4. Kiértékelés	14
4. Eredmények	16
4.1. A generált kód elemzése	16
4.2. A szöveges válasz elemzése	19
4.3. A megbízhatóság elemzése	21
5. Diskusszió	23
5.1. Az eredményekről	23
5.1.1. Manuális értékelés során elfogadott esetek, amelyeket az egységtesztek elutasítottak	23
5.1.2. Esetek, ahol az előzetes tesztelés (sanity check) elbukott	24
5.1.3. Egyéb érdekes esetek	24
5.2. Keresztvalidáció	25
6. Eredményeinket veszélyeztető tényezők	26
7. Összegzés	27
Függelék	28
Irodalomjegyzék	34

1. fejezet

Bevezetés

Az elmúlt években a pszichológiától [1] a gyógyszerészetén át [2] a szoftverfejlesztésig számos területen felfigyeltek a nagy nyelvi modellek nyújtotta lehetőségekre. Ezek a modellek elsősorban szöveges nyelvi elemek feldolgozására specializálódtak, így a szemantikai szabályok alapján felépülő programkódok elemzésére is alkalmazhatók. A legtöbb nagy nyelvi modell alapja a transformer architektúra [3], amely encoder és egy decoder részekből áll.

A BERT [4] egy encoder alapú modell, ennek egy kifejezetten forráskód feldolgozására továbbfejlesztett változata a CodeBERT [5]. A BERT egy maszkolt nyelvi modell, vagyis az ismeretlen (maszkolt) szót az azt megelőző, illetve utána következő szövegrész alapján próbálja megválasztani, így ezek a modellek nem a legalkalmasabbak szöveggenerálásra.

A Generative Pre-trained Transformer (GPT) egy dekóder alapú architektúra. Ez képezi az OpenAI cég GPT-3 modelljének alapját is, amelynek egy speciális, forráskódon tanított változata a Codex [6]. A legújabb GPT alapú modell a GPT-4.

A szoftverfejlesztés egyik legígéretesebb területe az automatikus programjavítás, hiszen a feladat új "szöveg", a kijavított forráskód generálása. A modellek valódi képességéről azonban keveset tudunk, csupán néhány olyan tanulmány van, amely valós rendszerek valós sérülékenységeit használja.

A szoftveres sebezhetőség kifejezésnek nincsen formális definíciója. Potenciális sérülékenységek szinte bármilyen rendszerben megtalálhatók, de célunk dokumentált, a gyakorlatban már felismert biztonsági hibák használata. A kiértékelésekhez ezért a VUL4J [7] adatbázist választottuk, amely a 79 darab valós Java sebezhetőség mellett azok kijavításáról is tartalmaz információt. Az adathalmaz egyik nagy előnye, hogy egységteszteket (*unit test*) is biztosít az egyes példákhoz, segítve a modell által generált eredmények automatikus és objektív értékelé-

sét. A VUL4J minden eleme rendelkezik egy CVE (*Common Vulnerabilities and Exposures*) [8] azonosítóval is.

A CVE egy publikusan elérhető adatbázis: valós rendszerek biztonsági problémáiról tartalmaz információt. Minden bejegyzett elem rendelkezik egy azonosítóval, amely egy adott sérülékenységet ír le. Az egyes sérülékenységek CWE (*Common Weakness Enumeration*) [9] kategóriákba sorolhatók. A CWE egy általánosabb absztrakciója a szoftver- és hardversebezhetőségeknek. Mindkét adatbázis karbantartásáról a MITRE vállalat gondoskodik.

A VUL4J segítségével részletes kutatást végeztünk az OpenAI legújabb GPT alapú modelljéről, a GPT-4-ről. A bemenetként adott szöveg (prompt) optimalizálására különböző stratégiákat alkalmaztunk (prompt engineering), hogy egy megfelelő promptot találjunk az automatikus programjavítási feladatra. Az előkészített bemeneteket két alkalommal küldtük el a GPT-4-nek a nyilvános API segítségével, a válaszokat pedig lementettük. A két egymástól független eredményhalmazt manuálisan és az adatbázis egységtesztjeivel vizsgáltuk meg. A kutatásban a következő kérdésekre keressük a választ:

- Milyen hatékonyan képes a GPT-4 automatikusan javítani a sebezhetőségeket egy valós környezetben?
- Vannak-e olyan sebezhetőségek, amelyekre a GPT-4 nem képes megfelelő forráskódot generálni, de a mellékelt szöveges magyarázat hasznosnak bizonyul?
- Mennyire konzisztensek a GPT-4 javításai az egyes futások és CWE kategóriák között?

Az eredmények azt mutatják, hogy a GPT-4 a két futtatás során átlagosan az esetek 35,14%-ában generált helyes, egységtesztek által elfogadott javításokat. A manuális kiértékelés magasabb arányt mutat, hiszen ott a forráskódhoz mellékelt szöveges magyarázatokat is figyelembe vettük, illetve néhány esetben az apró finomításokat igénylő kódrészleteket is elfogadtuk. Bár a modell válaszai bizonyos szintű véletlenszerűséget mutatnak a különböző futtatások között, többnyire képes volt a GPT-4 ugyanazon hibát mindkét alkalommal sikeresen kijavítani. Ezek az eredmények azonban jelentősen elmaradnak más tanulmányok értékeihez képest, amelyekben mesterségesen összeállított adathalmazokon [10] akár 77,5%-os javítási arányt is elértek. Biztató viszont, hogy Wu és munkatársai [11] ugyancsak a VUL4J adathalmazzal csupán 20,4%-os arányt értek el annak ellenére, hogy a hiba pontos helyét (sorát) is megadták a modellnek. Ez mutatja, hogy a nagy nyelvi modellek folyamatosan fejlődnek, hamarosan elér-

ve azt a szintet, amikor már a gyakorlatban is alkalmazhatjuk őket szoftveres sebezhetőségek automatikus kijavítására.

A dolgozat felépítése a következők szerint szerveződik: a 2. fejezetben a témához kapcsolódó egyéb munkákat soroljuk fel, a 3. fejezetben mutatjuk be tanulmányunk módszertanát. A kutatás eredményeit a 4. fejezet részletezi, az 5. fejezetben a speciális eseteket tárgyaljuk. A 6. fejezetben a kutatásunkat veszélyeztető tényezőket vizsgáljuk, végül a 7. fejezet zárja a tanulmányt.

2. fejezet

Kapcsolódó munkák

A nagy nyelvi modellek feladatok széles skáláját képesek végrehajtani forráskódon, ezek közül a legjelentősebbek a hibák, sebezhetőségek automatikus kijavítása [12]. A GPT-2 megjelenése óta számos tanulmány foglalkozik a témával, például Patrik és Wang [13] GPT-2 segítségével generált programkódot. Lajkó és kutatótársai [14] egy finomhangolt GPT-2 modell hibajavító képességét vizsgálták a JavaScript hibákat tartalmazó BugsJS [15] adathalmazon. Egy másik munka Lajkó és munkatársai [16] részéről különböző finomhangolt és készen elérhető GPT-2 modelleket hasonlított össze.

Prenner és munkatársai [17] a gondosan összeállított, szintetikus példákat tartalmazó QuixBugs [18] adathalmazt használták fel. Az adatkészlet 40 egyszerűbb, Python és Java nyelven írt, szándékosan hibás kódrészletet tartalmaz. Az OpenAI Codex modelljével az adathalmazban található példák 57,5%-át sikerült kijavítaniuk, mindössze a hibás kódrészletet és a hozzá tartozó Python dokumentációt szolgáltatva bemenetként. Továbbá a modell képes volt a hibák 45%-át úgy kijavítani, hogy a bemenet csupán a Python dokumentációt tartalmazta. Kísérleteik során a modell jobban teljesített a Python nyelvű kódokon, a Java nyelvű kódok eredményeihez viszonyítva. Sobania és társai [10] ugyancsak a QuixBugs adathalmazt használták az OpenAI ChatGPT modelljének kiértékelésére, amellyel 77,5%-os javítási arányt tudtak elérni. Bár ezek a munkák lenyűgöző eredményeket mutatnak, fontos megjegyezni, hogy mindegyikük egyszerű, mesterségesen előállított hibákat tartalmazó, kis méretű adathalmazokkal dolgozott, amelyek nem feltétlenül tükrözik a valós forráskódok kihívásait. A modellek teljesítményének pontosabb megismeréséhez realisztikusabb programozási példákon végzett vizsgálatok szükségesek.

Több tanulmány is foglalkozik GPT-modellek automatikus programjavító képességének ki-

értékelésével, ahol a hibás kódrészletet a modell generálta. Charalambous és munkatársai [19] GPT modellekkel generáltak hibás kódrészleteket, majd ezeket megkísérelték automatikusan kijavítani. Hasonlóképpen írtak a modellel hibás C kódot Hammond és kutatótársai [20], hogy aztán javítási feladatokra használják fel őket.

Bár vannak munkák, amelyek valós példákra támaszkodnak az automatikus programjavítás kiértékeléséhez, legtöbbjük szintaktikai problémákat old meg valódi sérülékenységek helyett. Ilyen például Nashid és munkatársai [21] tanulmánya, ahol a kiértékelő adathalmaz 52%-át sikerült javítani. Xia és kutatótársai [22] szintén valós projekteket tartalmazó adathalmazokat használtak (Defects4J [23] és ManyBugs [24]), azonban ezek általános szoftverhibákat tartalmaznak, nem kifejezetten sérülékenységeket.

Chunqiu és társai [25] több nyelvi modell teljesítményét hasonlították össze automatikus programjavítási feladatokban. A kiértékelés során a lehetséges javításokat egységtesztekkel validálták, majd amennyiben szükség volt rá, az előző kimenetet felhasználva új megoldást generáltattak. Chunqiu és Lingming munkájához kapcsolódik az AlphaRepair [26] automatizált javítóeszköz. A bemenet kódrészleteiben megjelölték a hibás sorokat, majd az így kapott eredményeket interaktív módon értékelték a unit tesztek segítségével. Az iteratív generálásra és tesztelésre támaszkodó módszerek nem alkalmasak gyakorlatban alkalmazható automatikus programjavításra, hiszen az adott hibát felismerő egységtesztek általában nem állnak rendelkezésre.

Ahhoz, hogy mélyebb betekintést nyerjünk a GPT-alapú automatikus programjavítás gyakorlati alkalmazhatóságába, valós példákon kell vizsgálnunk a modell teljesítményét. Ez azonban azt is jelenti, hogy a generálási folyamathoz nem használhatunk fel egységteszteket, azok csak a javított példák automatikus, objektív ellenőrzésében segíthetnek. Az ilyen tanulmányok ritka példája közé tartozik Wu és társai [11] munkája, amely a VUL4J [7] és VJBench adathalmazok gyakorlati példáit használta. A megvizsgált nagy nyelvi modellek az adatbázisokból válogatott 42 példa 20,4%-ában tudták kijavítani a sebezhetőségeket. A Codex modell esetében a szerzők a forráskódban kommentekkel jelezték a hibát tartalmazó sort. Az eredmények értékelésére egységteszteket használtak, így választották ki a generált 10 javítás közül a legjobbat.

A mi munkánkban az OpenAI legújabb modelljét, a GPT-4-et teszteltük a VUL4J adathalmaz 46 példáján, és más tanulmányokkal ellentétben csupán a hibás metódust és az azt tartalmazó osztályt adtuk át bemenetként, semmilyen információt nem szolgáltatva a hiba hollétéről. Ez a megközelítés sokkal közelebb áll a gyakorlati esetekhez, így pontosabb információt

kaphattunk a GPT-4 valódi teljesítményéről. A bemenetet egyedül a sebezhetőség CWE besorolásával egészítettük ki. Bár ez az információ nem mindig áll rendelkezésre a gyakorlatban, sokkal könnyebben meghatározható, mint a probléma pontos helye a forráskódban. Emellett nem támaszkodtunk az egységtesztekre sem a javítások generálása során; törekedtünk a lehető legpontosabban követni egy gyakorlati alkalmazás körülményeit. A tanulmány empirikus eredményeket ad a GPT-4 automatikus szoftversebezhetőség-javító képességével kapcsolatban, amelyek közelebb állnak a valós alkalmazáshoz, mint eddig bármely korábbi munkában.

3. fejezet

Módszertan

A folyamat négy fő részre bontható. Először adatot gyűjtöttünk a GPT kiértékeléséhez, majd az összeállított adatbázisunkból kiválogattuk a sérülékeny metódusok forráskódját. Következő lépésként megalkottuk a megfelelő promptokat, amelyeket aztán a kódrészletekkel együtt adtunk át a GPT-nek. Végül lefuttattuk a modellen az összeállított bemeneteket, majd manuálisan és egységtesztekkel értékeltük az eredményeket.

3.1. Adatgyűjtés

Az értékelés elvégzéséhez valós sebezhetőségeket kellett szereznünk. Annak érdekében, hogy objektív és (félig) automatizált módon tudjunk értékelni (a manuális vizsgálat mellett), csak olyan adathalmazok jöhettek számításba, amelyek egységtesztet is biztosítanak. Az egyik legfrissebb, de máris széles körben elterjedt adathalmaz, a VUL4J [7] minden követelményünknek megfelelt, hiszen valós sérülékenységeket tartalmaz unit tesztekkel együtt. A gyűjtemény minden példájához tartozik hibás forráskód, fejlesztő által kijavított forráskód, a problémás részt felismerő tesztelés és információ a sérülékenység reprodukálhatóságáról. Ezen kívül egyéb releváns adatokat is tartalmaz, mint például a CVE azonosítót, CWE és OWASP kategóriákat.

Sokféle sérülékenység található a VUL4J-ben, vannak példák, amelyek csupán egy metódusban igényelnek módosítást (ezek legtöbbször *egysoros (single-line)*, vagy *egyrészes (single-hunk)* javítások), míg más példákban sokkal bonyolultabb beavatkozás szükséges. Ilyenkor több helyen kell változtatást végezni egy fájlban belül (*többrészes (multi-hunk)* javítás), a legbonyolultabb esetekben viszont több fájlban kell módosítani. Bármilyen sérülékenység javítása nagy kihívást jelentő feladat, amit jól tükröz az is, hogy a legmodernebb technológiák [27, 28] is

egyrészes, vagy maximum többrészes változtatásokat céloznak meg egy adott fájlban belül. A nagy nyelvi modelles javítások esetében többfájlos, többrészes sérülékenységek átadása nehézkes, tapasztalataink alapján¹ a modell legtöbbször kifut a kontextusból. Éppen ezért döntöttünk úgy, hogy mi is csak egyfájlos javítási feladatokat adunk át a GPT-nek. Ez lehetővé teszi továbbá azt is, hogy a kutatók hagyományos programjavító módszerekkel hasonlítsák össze a mi eredményeinket.

Ebből kifolyólag megszűrtük az adatbázist, és a VUL4J csupán egy olyan részhalmazát alkalmazzuk, amelyben a sérülékenységek egy fájlban igényelnek módosítást (a legtöbbjük egyrészes, de van néhány többrészes is). Ez a megoldás összhangban van a legmodernebb megközelítésekkel, és ha a GPT nem képes egyfájlos problémák megoldására, akkor valószínűtlen, hogy képes lenne többfájlos javításokra. Ha azonban a modell hatékonyan képes egyfájlos hibákat javítani, akkor később kiterjeszthetjük a kutatást többfájlos esetekre is.

Technikai szempontból a sebezhetőségeket javító GitHub commitokat használtuk, hogy a GitHub API segítségével meghatározzuk a módosított metódusok és fájlok számát. Az adatbázis szűrésével 33 esetet kellett kivennünk az eredeti halmazból, így továbbiakban a fennmaradó 46 sérülékeny példával dolgoztunk.

3.2. Metódusok kigyűjtése

A neurális hálózatok (például a nagy nyelvi modellek, a mi esetünkben a GPT) alkalmazása az automatikus programjavításban neurális gépi fordítási feladatnak tekinthető. Ez azt jelenti, hogy a modellnek hibás forráskódot adunk bemenetként, és annak kijavított változatát várjuk kimenetként. Ez a megközelítés nyilvánvalóan megköveteli a hibás kódrészlethez tartozó kontextus meglétét. A metódus szintű kontextus a legelterjedtebb a szakirodalomban [29, 30, 31, 32], hiszen általában elégséges mennyiségű információt tartalmaz a modellek megfelelő működéséhez, különösen egyrészes hibák esetében. Teljes fájlok átadása gyakran technikailag nem kivitelezhető, emellett ilyen nagy mennyiségű adat elterelheti a modell figyelmét, ami az eredmények romlásához vezethet. Ezek alapján mi is metódus formájában adtuk meg a javításhoz szükséges kontextust.

¹A hivatalos dokumentáció szerint (<https://platform.openai.com/docs/models/gpt-4>) a GPT-4 egy kérés alkalmával maximum 32,768 tokenet képes feldolgozni (ez magában foglalja a bemeneti és kimeneti tokeneket is).

Annak érdekében, hogy egyszerűen kigyűjthessük a sérülékeny metódusokat a forrásfájlból, készítettünk egy egyszerű parancssori alkalmazást, amely a VUL4J adatbázis alapján kigyűjti a szükséges adatokat. A GitHub API és JavaParser² könyvtár segítségével összegyűjtöttük a sérülékeny példákat tartalmazó osztályok fejlécét, adattagjait, illetve hibát tartalmazó metódusok kódrészleteit.

3.3. Promptok írása és fejlesztése

A GPT API lekérdezéséhez szükségünk van a megfelelő prompt(ok)ra. Először áttekintettük a vonatkozó szakirodalmat [33, 34], majd manuálisan kipróbáltunk több promptot.

A valódi alkalmazási körülmények tükrözéséhez zero-shot promptolás mellett döntöttünk, vagyis nem adtunk meg explicit példákat az elvárt kimenetről. A promptkísérletek korai fázisában észrevettük, hogy a GPT gyakran elveszíti a fókuszot. A sérülékenység kijavítása helyett apró, a hiba szempontjából lényegtelen módosításokat hajtott végre a kódon. Ezért döntöttünk úgy, hogy további segítséget nyújtunk a modellnek a hiba CWE kategóriájának megadásával. Mások is találkoztak ezzel a problémával [11], és explicit módon megjelölték a hibát tartalmazó sorokat a modell számára. Úgy véljük, ez a való életben jelentősen korlátozná a módszer alkalmazhatóságát, hiszen a fejlesztők nem tudhatják, hol van pontosan a sérülékeny rész. Ellenkező esetben nem írnának hibás kódot. Ugyanakkor projekt- vagy területspecifikus ismeretekkel rendelkező személyek, például fejlesztők vagy tesztelők képesek lennének megállapítani a lehetséges sebezhetőségek típusait, így ez nem limitálja nagy mértékben az alkalmazhatóságot.

A négy legígéretesebb promptunkat aztán tovább finomítottuk és az adathalmazunkból tíz véletlenszerűen kiválasztott példán teszteltük, ügyelve, hogy a példák változatos sebezhetőségeket tartalmazzanak. Ez volt egyúttal a keretrendszerünk próbája is, amely segítségével automatikusan tudunk lekérdezéseket indítani a GPT API felé a meghatározott prompt sablonokat használva. A négy leghatékonyabb prompt:

- **DIFF3** - A prompt három javító kódrészletet kér a modelltől, amelyek mindegyikéhez szöveges magyarázatot is kell biztosítani. A három lehetőség közül ki kell választania a legjobb megoldást. A prompt különlegessége, hogy csupán a megváltoztatott kódsorokat kéri a válaszban diff³ formátumban.

²A JavaParser könyvtár Java forráskód absztrakt szintaxisfáját írja le különféle elemzési célokra. Bővebb információ a <https://javaparser.org/> oldalon érhető el.

³<https://en.wikipedia.org/wiki/Diff>

- **FULL** - Ez a prompt szolgáltatja a legtöbb információt a sérülékenységről. A forráskód mellett megadjuk a kódot tartalmazó projekt és a sebezhetőség CWE kategóriájának rövid leírását.
- **IMPROVE** - A prompt a sebezhetőség kijavítása mellett megengedi a modellnek, hogy hatékonyabbá tegye a kódot, szükség szerint akár törölhet is belőle részeket az eredeti funkcionalitás megtartásával.
- **EXPLOIT** - A prompt megkéri a modellt, hogy sorolja fel a kódban található sérülékenységeket. Ezután olyan kódot kér, amely a CWE kategóriába leginkább illő sebezhetőséget használja ki. Végül legeneráltatja vele a sebezhető metódus javított változatát.

Mindegyik prompt a javított forráskódon kívül szöveges leírást is kér a módosítás lépéseiről, illetve mindegyik promptban megadtuk a CWE kategóriát.

A promptok értékeléséhez a példákat kontrollált véletlenszerű módon jelöltük ki. Miután a kiválasztott sebezhetőségekből kigyűjtöttük a szükséges adatokat, lefuttattuk őket a GPT-3.5⁴ és GPT-4⁵ modelleken. A GPT-4-en kétszer futtattuk a promptokat, hogy csökkentsük a véletlen eredmények esélyét.

A 10 véletlenszerűen választott példa eredményének értékelését a 3.4 szakaszban leírtak szerint végeztük. Annak érdekében, hogy nagyobb rálátást kapjunk a válaszok minőségéről, több kritériumot határoztunk meg az eredményül kapott kód és szövegre vonatkozóan: (1) teljes-e a forráskód (*code_comp*), (2) helyes-e a kódjavítás (*code_corr*), (3) releváns-e a kód javítása (*code_rel*), (4) helyes-e a magyarázat (*text_corr*), (5) releváns-e a magyarázat (*text_rel*). A válaszokat manuálisan vizsgáltuk meg ezen kritériumok szerint. Ha az adott szempont nem teljesült, 0 pontot adtunk, 1 pontot, ha igen. Végül az alábbi képlet szerint számítottuk ki a promptok végső pontszámát:

$$score_code_i = code_comp_i + code_corr_i + code_rel_i \quad (3.1)$$

$$score_text_i = text_corr_i + text_rel_i \quad (3.2)$$

$$score_i = score_code_i + score_text_i \quad (3.3)$$

$$score_{prompt} = \sum_{i=1}^{10} score_i \quad (3.4)$$

⁴A pontos verzió: *gpt-3.5-turbo-16k-0613*.

⁵A pontos verzió: *gpt-4-0613*.

A kapott pontszámok határozzák meg a válaszok minőségét, ezek alapján választottuk ki a leghatékonyabb promptot. Az 3.1. és 3.2. egyenletekben számítottuk ki a kód és a szöveges javítások pontszámát, amelyeket a 3.3. egyenletben összegeztünk. A prompt teljesítményét a 10 példa eredményének összegeként kaptuk meg a 3.4. egyenlet alapján. A manuális értékelést legalább két kutatótárs végezte, majd egy harmadik is ellenőrizte az eredményeket, így legalább 3 különböző személy vett részt az egyes javítások kiértékelésében. Az ellenőrzés után mindegyik javítást beillesztettük az eredeti környezetébe, majd lefuttattuk az egységteszteket.

Példa #	GPT-3.5								GPT-4 (Futtatás #1)								GPT-4 (Futtatás #2)							
	DIFF3		FULL		IMPROVE		EXPLOIT		DIFF3		FULL		IMPROVE		EXPLOIT		DIFF3		FULL		IMPROVE		EXPLOIT	
	Kézi	Teszt	Kézi	Teszt	Kézi	Teszt	Kézi	Teszt	Kézi	Teszt	Kézi	Teszt	Kézi	Teszt	Kézi	Teszt	Kézi	Teszt	Kézi	Teszt	Kézi	Teszt	Kézi	Teszt
VUL4J-4	0	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0
VUL4J-6	0	0	0	0	1	0	0	0	1	0	1	0	1	0	1	0	0	0	3	0	3	0	0	0
VUL4J-16	1	0	3	0	0	0	1	0	1	0	1	0	1	0	2	0	1	0	1	0	1	0	0	0
VUL4J-18	3	0	2	0	1	0	2	0	3	0	3	0	3	0	3	0	3	0	3	0	3	0	3	0
VUL4J-24	2	0	4	0	4	0	4	0	3	0	5	1	5	0	5	1	5	0	5	0	5	1	5	0
VUL4J-40	1	0	2	0	2	0	2	0	2	0	3	0	3	0	3	0	1	0	1	0	1	0	1	0
VUL4J-44	1	0	4	0	1	0	0	0	1	0	3	0	1	0	1	0	1	0	3	0	1	0	1	0
VUL4J-48	0	1	3	0	3	0	3	0	3	0	3	0	3	0	3	0	3	0	3	0	3	0	3	0
VUL4J-54	3	0	2	0	1	0	3	0	2	0	3	0	3	0	3	0	3	0	3	0	3	0	2	0
VUL4J-65	0	0	5	1	5	0	5	0	5	0	5	0	5	0	5	1	5	0	5	0	5	1	5	0
Σ	11	1	25	1	19	0	21	0	22	0	28	1	26	0	27	2	22	0	28	0	26	2	21	0

3.1. táblázat. A prompt engineering eredményei

Az eredmények rövid összefoglalóját a 3.1. táblázatban mutatjuk be. Amint látható, a GPT-4 általánosságban sokkal jobban teljesített, mint a GPT-3.5. A promptok közül, a FULL prompt volt a legkövetkezetesebb és legjobban teljesítő a kísérletek során, 25 (GPT-3.5), 27 és 28 (GPT-4) pontokat elérve. A leggyengébben teljesítő a DIFF3, míg a legkevésbé konzisztens az EXPLOIT prompt volt, 21 (GPT-3.5), 27 és 21 (GPT-4) pontokat elérve. Fontos megjegyezni, hogy a generált javítások csupán néhány esetben mentek át az egységteszteken, jellemzően 10-ből csak egy, ami megközelítőleg 10%-os elfogadási arányt mutat. A meghatározott kritériumok alapján végzett részletes kézi kiértékelés azonban mélyebb betekintést nyújtott a promptok minőségébe. A promptok részletes eredményei, illetve az egyes promptszövegek a függelékben olvashatók.

A végső promptsablon megalkotásához végül a FULL promptot használtuk, hiszen ez volt a legjobban teljesítő a 4 közül. Mind a 46 példát az alábbi sablonnal futtattuk le a GPT API segítségével:

```
You are a senior cyber security specialist and Java developer.
You will be given the source code of a Java class where you will find only one ↵
vulnerable method. Your task is to patch any vulnerabilities in this method. You will ↵
also have access to the description of the Java project and the issue.

Project description: {The description of the project}

Issue description: {The description of the CWE}

{source code}

It is very important for me, please patch the vulnerable code based on your best ↵
knowledge based on the given descriptions.
Provide the patched source code and write detailed comments about your fix explaining ↵
your way of thinking.
```

Kódrészlet 3.1. A végleges prompt sablonja.

A promptunk egy szerepkör (*role*) megadásával kezdődik. A hivatalos dokumentáció szerint⁶ ez segít a modellnek megérteni a kontextust, amelyben dolgoznia kell. Ezután elmagyarázzuk, milyen bemenetre számíthat a modell, milyen választ várunk és mi a pontos feladat. Ezeken felül megadjuk a projekt GitHub-on közzétett leírását (például: *An Implementation of the Jakarta RESTful Web Services Specification*). Ez az információ segít a modellnek megérteni a kapott kódrészlet funkcióját és az ahhoz kapcsolódó lehetséges sérülékenységeket. Ezután megadjuk a sebezhetőség CWE kategóriáját és annak rövid leírását, hogy a modell az általános kódhibák helyett a fő sebezhetőségre koncentráljon. A következő blokkban adjuk át a sérülékeny forráskódot. Végül tisztázzuk a modell számára, hogy ez a feladat nagyon fontos számunkra, hiszen az érzelmi stimulációk segíthetnek jobb eredmények generálásában [33]. A pontosság javítása érdekében megkérjük a modellt, hogy magyarázza el lépésről lépésre a gondolatmenetét [34].

Kísérleteztünk magyar nyelvű promptok futtatásával is, így azonban kevésbé használható válaszokat kaptunk és a generálási idő is megnőtt, ezért ezt a megközelítést elvetettük.

⁶<https://platform.openai.com/docs/guides/gpt/chat-completions-api>

3.4. Kiértékelés

A teljes adathalmaz kiértékeléséhez a legkorszerűbb GPT modellt, a GPT-4-et használtuk. A promptok tesztelése során is a GPT-4 eredményeit találtuk ígéretesebbnek. A VUL4J sebezhetőségek értékeléséhez a leghatékonyabb promptot alkalmaztuk (3.1. kódrészlet). Minden példát kétszer futtattunk le, hogy minimalizáljuk a véletlenszerű válaszok esélyét, az eredményeket pedig egymástól függetlenül kezeltük. A válaszokat az alábbiak szerint vizsgáltuk:

Formátum

Megvizsgáltuk, hogy az adott válasz megfelel-e a kért formátumnak, vagyis tartalmazza mind a kódot, mind a természetes szöveggel írt magyarázatot. Szerencsére a formátum minden esetben megfelelt az elvárásoknak, így a továbbiakban ezzel a szemponttal nem foglalkozunk.

- **Lehetséges értékek:** igen/nem

Kód

A javított forráskóddal kapcsolatos követelmények.

- **Teljes:** a kód teljesnek minősül, ha módosítás nélkül, vagy legfeljebb apró finomításokkal (például hiányzó importok pótlásával) alkalmazható. Ha a forráskód nem létező metódusra hivatkozik, vagy nagyobb változtatások szükségesek a futtatáshoz, akkor nem minősül teljesnek.

- **Lehetséges értékek:** igen/nem

- **Helyes:** a javítás helyesnek minősül, ha közvetlen megoldást nyújt a sebezhetőségre. Ha a modellnek nem sikerült teljes mértékben kijavítania a hibát, de lépéseket tesz annak irányába, akkor a megoldás részben helyes.

- **Lehetséges értékek:** igen/nem/részben jó

- **Releváns:** akkor minősül relevánsnak a javítás, ha a modell a problémával kapcsolatos módosításokat végez.

- **Lehetséges értékek:** igen/nem

Szöveg

A modellt megkértük a gondolatmenetének leírására is, így a szöveges válaszokat is értékeltük.

- **Helyes:** a szöveges válasz egyértelműen leírja az adott sebezhetőség javításához szükséges lépéseket. Ha a modell csak rövid, kevésbé pontos, de helyes leírást biztosít, akkor a válasz részben jó.

– **Lehetséges értékek:** igen/nem/részben jó

- **Releváns:** a szöveges válasz a megadott hibára összpontosít, és nem javasol lényegtelen módosításokat.

– **Lehetséges értékek:** igen/nem

4. fejezet

Eredmények

A 3.4. szakaszban tárgyaltak alapján, a példákat két alkalommal futtattuk a GPT-4-en, és a TOP-1 eredményeket használtuk. A kiértékelés részletes eredményeit a 4.1. táblázatban mutatjuk be. A generált eredményeket manuálisan és az egységtesztekkel értékeltük. Sajnos nem tudtunk minden tesztet lefuttatni a VUL4J Docker környezetében a projektben, a függőségek elérhetőségében vagy az adathalmazban történt változások miatt. Ezen felül két esetben az eredeti hibás kód és a fejlesztő által javított megoldás is átment az egységteszteken, így ezeket eredményeket nem tudtuk számításba venni. A kivételes eseteket részletesen tárgyaljuk a 5.1.2. szakaszban. Végül 46 példát ellenőriztünk manuálisan, de csak 37 esetben futtattunk egységteszteket is. A továbbiakban az arányokat manuális esetben 46, a unit tesztek esetében pedig 37 példához képest számítjuk.

4.1. A generált kód elemzése

A generált szöveg és forráskód helyessége 3 lehetséges értéket vehetett fel (igen, nem, részben jó), de végül döntenünk kellett, hogy az adott javítást elfogadjuk-e vagy sem. Ez szubjektív véleményt vitt az értékelésbe, amely hatással lehet a végső értékekre.

Az első futtatás alkalmával 15 (32,61%) megoldást fogadtunk el a kézi értékelés során, az egységteszteken pedig 12 (32,43%) modell által generált javítás ment át. A második futtatás hasonló értékeket mutat, a manuális értékelés 16 (34,78%), a tesztesetek 14 (37,84%) megoldást hagytak jóvá. Bizonyos esetekben az egységtesztek túl specifikusnak bizonyultak (pontos egyezést követelve a probléma szempontjából lényegtelen pontokon), ekkor a forráskódok minimális módosításon estek át.

Példa #	Futtatás #1							Futtatás #2						
	Kód			Szöveg		Javítás		Kód			Szöveg		Javítás	
	Teljes	Helyes	Releváns	Helyes	Releváns	Kézi	Teszt	Teljes	Helyes	Releváns	Helyes	Releváns	Kézi	Teszt
VUL4J-1	x	x	x	x	x	x	x	x	x	x	x	x	x	x
VUL4J-2	✓	✓	✓	✓	✓	✓	-	✓	x	x	x	x	x	-
VUL4J-3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
VUL4J-4	✓	x	x	x	x	x	x	✓	x	x	x	x	x	x
VUL4J-5	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
VUL4J-6	✓	✓	✓	✓	✓	x	-	✓	x	x	x	x	x	-
VUL4J-7	✓	✓	✓	✓	✓	x	x	✓	x	x	x	x	x	x
VUL4J-8	✓	x	x	x	x	x	x	✓	x	x	x	x	x	x
VUL4J-10	✓	x	x	x	x	x	x	✓	x	x	x	x	x	x
VUL4J-12	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
VUL4J-13	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
VUL4J-15	✓	x	x	x	x	x	x	✓	x	x	x	x	x	x
VUL4J-16	✓	x	x	x	x	x	x	✓	✓	x	✓	x	x	x
VUL4J-17	✓	x	x	x	x	x	x	✓	x	x	x	x	x	x
VUL4J-18	✓	x	x	x	x	x	x	✓	x	x	x	x	x	x
VUL4J-19	✓	x	x	x	x	x	-	✓	✓	✓	✓	✓	x	-
VUL4J-20	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
VUL4J-22	✓	x	x	x	x	x	x	x	x	x	x	x	x	x
VUL4J-23	✓	✓	✓	✓	✓	x	x	✓	x	x	x	x	x	x
VUL4J-24	✓	✓	✓	✓	✓	x	x	✓	✓	✓	✓	✓	✓	✓
VUL4J-25	✓	x	✓	✓	✓	x	x	✓	x	x	✓	x	x	x
VUL4J-26	✓	x	x	x	x	x	x	✓	x	x	x	x	x	x
VUL4J-30	✓	x	x	x	x	x	-	✓	x	x	x	x	x	-
VUL4J-31	✓	x	x	x	x	x	x	✓	x	x	x	x	x	x
VUL4J-34	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
VUL4J-39	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	x
VUL4J-40	✓	x	x	x	x	x	x	✓	x	x	x	x	x	x
VUL4J-41	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
VUL4J-44	✓	x	x	x	x	x	x	✓	x	x	x	x	x	x
VUL4J-45	✓	✓	✓	✓	✓	x	x	x	x	✓	✓	✓	x	x
VUL4J-48	✓	✓	✓	✓	✓	x	x	✓	✓	✓	✓	✓	x	x
VUL4J-50	✓	✓	✓	✓	✓	x	x	✓	✓	✓	✓	✓	✓	✓
VUL4J-51	✓	x	x	✓	x	x	-	✓	x	x	x	x	x	-
VUL4J-53	x	x	x	x	x	x	x	x	x	x	x	x	x	x
VUL4J-54	✓	✓	✓	✓	x	x	x	x	x	x	x	x	x	x
VUL4J-55	✓	✓	✓	✓	✓	x	x	x	✓	✓	✓	✓	x	x
VUL4J-57	✓	x	x	x	x	x	x	✓	x	x	x	x	x	x
VUL4J-61	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
VUL4J-62	✓	x	x	x	x	x	-	✓	x	x	x	x	x	-
VUL4J-63	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
VUL4J-64	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
VUL4J-65	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
VUL4J-66	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
VUL4J-69	✓	✓	✓	✓	✓	✓	-	✓	✓	✓	✓	✓	✓	-
VUL4J-74	✓	x	✓	x	✓	x	-	✓	x	✓	x	✓	x	-
VUL4J-79	x	✓	✓	✓	✓	x	-	✓	✓	✓	✓	✓	x	-
Helyes	42	13	27	18	26	15	12	40	15	22	19	22	16	14
Helytelen	4	21	19	19	20	31	25	6	25	24	23	24	30	23

4.1. táblázat. A két futtatás részletes eredményei.

Az első futtatásból:

- VUL4J-5: A GPT megoldása nem az elvárt típusú kivételt dobta hiba esetén.
- VUL4J-41: A GPT megoldása nem az elvárt hibaüzenettel dobott kivételt.
- VUL4J-63: A GPT megoldása hiba esetén -1 értékkel tért vissza, azonban az egységtesztek kivételdobást vártak.
- VUL4J-65: A GPT megoldása a hibás input felismerésekor nem dobott hibát, hanem a sérülékenységet okozó értéket kijavította, így a függvény tovább futhatott.

A második futtatásból:

- VUL4J-5: A GPT megoldása nem az elvárt típusú kivételt dobta hiba esetén.
- VUL4J-41: A GPT megoldása a hibás input felismerésekor nem dobott hibát, hanem a sérülékenységet okozó értéket kijavította, így a függvény tovább futhatott.
- VUL4J-63: A GPT megoldása hiba esetén -1 értékkel tért vissza, azonban az egységtesztek kivételdobást vártak.
- VUL4J-65: A GPT megoldása a hibás input felismerésekor nem dobott hibát, hanem a sérülékenységet okozó értéket kijavította, így a függvény tovább futhatott.

Ezeket az javításokat jóváhagytuk, hiszen az apró módosítások nem befolyásolták a megoldás funkcionalitását. Azokban az esetekben, amikor az eredeti (hibás) forráskódban a hibakezelés visszatérési érték és kivételdobás formájában is megjelent, nem fogadtuk el, ha a megoldás értékkel tért vissza kivétel helyett, hiszen az eredeti forráskódból lehetett következtetni a logikára.

Összehasonlítva az eredményeinket a kapcsolódó munkákkal, arra a következtetésre jutotunk, hogy a valós példákkal kapott eredmények jelentősen elmaradnak a mesterséges adathalmazokkal (QuixBugs) elért, 77,5%-os [10] és 57,5%-os [17] javítási arányokhoz képest. Wu és munkatársai [11] a miénkhez hasonló eredményeket, 20,4%-os javítási arányt mutattak, bár ők sorszintű információt is csatoltak a bemenethez.

Ez a GPT modellek javítási képességeinek folyamatos fejlődését mutatja; az automatikusan generált sebezhetőség-javítások egyre több segítséget nyújtanak a fejlesztők számára.

Milyen hatékonyan képes a GPT-4 automatikusan javítani a sebezhetőségeket egy valós környezetben?

Csupán a sebezhetőség típusának ismeretében a GPT-4 12 és 14 helyes megoldást generált, 32,43% és 37,84%-os javítási arányt elérve az első és második futtatás alkalmával (35,14%-os átlaggal). Ezek az eredmények messze alulteljesítenek a szintetikus adathalmazokon elért eredményekhez képest. A valós sebezhetőségeket használó tanulmányokhoz viszonyítva viszont enyhén jobb értékeket kaptunk annak ellenére, hogy nem jelöltük meg a hiba pontos helyét a forráskódban.

4.2. A szöveges válasz elemzése

A korábban bemutatott eredmények alapján nyilvánvaló, hogy nem minden generált javítás elfogadható. A prompt azonban szöveges magyarázatot is kért a kijavított forráskód mellé.

Azokban az példákban, ahol a generált kód átment a teszteken, nem meglepő, hogy hasznos szöveges leírásokat kaptunk. Voltak azonban esetek, amikor a válaszban kapott kódrészlet nem javította ki az adott sebezhetőséget, az értékelők mégis hasznosnak vagy legalább elfogadhatónak tartották a szöveges magyarázatot. Egy ilyen példa a VUL4J-7, ahol a GPT megoldását pirossal, a hivatalos helyes megoldást zölddel jelöltük a 4.1 kódrészletben.

```
// ...
final int l = in.read(buf.array());
-   if (l == -1 || l == 0) {
-       return -1;
+   if (l == -1) {
+       buf.limit(0);
+       throw new IOException("Truncated ZIP file");
// ...
```

Kódrészlet 4.1. A VUL4J-7 példához generált GPT-4 kód az első futtatásból

Látható, hogy a fejlesztők megoldásában az `l` változó `-1`-es értéke jelenti a hibát, ekkor kivételdobás szükséges. Ebben az esetben nem fogadhattuk el a kivétel helyett értékkel történő visszatérést, hiszen a `buf.limit` beállítása kulcsfontosságú a hiba szempontjából. A GPT kihagyta ezt a lépést, így az egységteszten akkor sem ment át a kód, ha a visszatérési értéket kivételdobásra cseréltük.

A forráskód nem helyes, de a mellékelt leírás hasznos lehet a fejlesztők számára. A 4.2. kódrészletben a modell egy potenciális végtelen ciklust említ, valamint megjegyzi, hogy a buffer használata összefüggésben áll a beállított limittel, ami ebben az esetben éppen a megoldás kulcsa. A szöveg nem fogalmazza meg pontosan a hiba megoldásának lépéseit, de tesz olyan megjegyzéseket, amelyek a probléma megoldása szempontjából lényegesek lehetnek.

```
The vulnerability in the code is that it can potentially enter an infinite loop if the  
InputStream 'in' returns 0 when the 'read' method is called. This can happen if the  
InputStream is blocked or if it is reading from an empty source. In such a case, the  
'readStored' method will keep looping indefinitely because the condition 'buf.position()  
>= buf.limit()' will always be true and 'in.read(buf.array())' will always return 0,  
which is not equal to -1.  
To fix this, we need to add a check to ensure that 'in.read(buf.array())' does not  
return 0. If it does, we should break the loop and return -1 to indicate that the end of  
the stream has been reached.  
<FAILING CODE>
```

Kódrészlet 4.2. A VUL4J-7 példához generált GPT-4 szöveg az első futtatásból

Második kutatási kérdésünk megválaszolásához azt vizsgáljuk, hogy gyakoriak-e a hasonló esetek, és hogy a generált szöveget fel lehet-e használni még akkor is, ha a mellékelt kód helytelen. Azokat az eseteket vettük figyelembe, ahol a manuális értékelés a szöveget helyesnek vagy részben helyesnek ítélte, a forráskódot azonban sem a fejlesztők, sem az egységtesztek nem fogadták el.

Az első futtatásban 27 (58,70%) esetben találtuk a szöveges utasításokat hasznosnak, és 9 (19,57%) olyan eset volt, amikor a forráskódot sem a manuális, sem az egységtesztes ellenőrzés nem fogadta le, mégis hasznosnak találtuk a magyarázatot. A második futtatás esetében ezek a számok 23 (50,00%) és 5 (10,87%). Fontos megjegyeznünk, hogy a második kör értékelésekor több kódrészletet fogadtunk el, így az arányok is kisebbek.

Vannak-e olyan sebezhetőségek, amelyekre a GPT-4 nem képes megfelelő forráskódot generálni, de a mellékelt szöveges magyarázat hasznosnak bizonyul?

A szöveges javításokat az első és második futtatásban az esetek 57,70% és 50,00%-ában találtuk hasznosnak, valamint 19,57% és 10,87%-ban kizárólag a magyarázat volt helyes. A szöveges javítások ígéretesnek bizonyulnak a félig automatikus programjavításban és forráskód-ellenőrzésben.

4.3. A megbízhatóság elemzése

Megvizsgáltuk, hogy mennyire hatékonyan lehet a generált forráskódot vagy szöveges magyarázatot automatikus programjavítási feladatokra használni. Az alkalmazhatóság egyik fontos aspektusa a megbízhatóság, vagyis mennyire következetesen képes a modell helyes javításokat generálni ugyanazokra a problémákra. Már megmutattuk, hogy a modell megoldásait 12 (32,43%) és 14 (37,84%) esetben fogadtuk el a 37-ből, de még nem vizsgáltuk a két eredményhalmaz metszetét. Ha a metszet üres, vagy csupán néhány elemet tartalmaz, azt bizonyítaná, hogy a modell nem képes következetes, megbízható javításokat nyújtani. Esetünkben 12 olyan példát számoltunk, amelyet a modell mind a két futtatás alkalmával sikeresen kijavított, ez a 37 példához viszonyítva 31,58%-os arány. Ha ezt az értéket összevetjük a két futás során legalább egyszer kijavított esetek számával (16 ilyen eset volt), akkor 75%-os konzisztenciát kapunk. Ismét hasonló eredmények születtek, mint Wu és kutatótársai [11] munkájában.

CWE	Σ	Futtatás #1		Futtatás #2	
		Helyes	Helytelen	Helyes	Helytelen
CWE-20	5	1 (20%)	4 (80%)	1 (20%)	4 (80%)
CWE-22	3	2 (66.7%)	1 (33.3%)	2 (66.7%)	1 (33.3%)
CWE-611	4	2 (50%)	2 (50%)	3 (75%)	1 (25%)
CWE-79	4	1 (25%)	3 (75%)	2 (50%)	2 (50%)
CWE-835	6	2 (33.3%)	4 (66.7%)	2 (33.3%)	4 (66.7%)

4.2. táblázat. A konzisztensen javított CWE kategóriák javítási arányai

A következetesség egyik lehetséges oka a sebezhetőség típusa, vagyis a CWE kategória. A metszet 12 példája a következők egyikébe sorolható: CWE-20, CWE-22, CWE-79, CWE-611, CWE-835. A kategóriák többsége a bemeneti értékek nem megfelelő kezelésével kapcsolatos, így megvizsgáltuk, hajlamosabb-e a modell az ilyen típusú sebezhetőségeket helyesen javítani. A 4.2. táblázatban láthatjuk, hogy az említett CWE kategóriák nagy számmal jelennek meg az elutasított példákban is, tehát a sérülékenység típusa nem lehet a következetesség fő oka.

A konzisztencia másik lehetséges magyarázatát a hibák jellegében kerestük. Megvizsgálva a két futás során generált 12-12 forráskódot úgy találtuk, hogy a javítások leginkább Java és programozói praktikákhoz kapcsolódnak. A módosításokat a következőképpen csoportosítottuk:

- **konfiguráció szigorítása:** a modell további beállításokat végez a használt objektumokon, például kikapcsolja XML beolvasása esetén a DTD-t.
 - ebbe a kategóriába tartozó példák: VUL4J-3, VUL4J-61 és VUL4J-64

- **bemeneti paraméterek szigorúbb ellenőrzése:** a modell több lépésben ellenőrzi a bemenetként adott paraméterek helyességét.
 - ebbe a kategóriába tartozó példák: VUL4J-5, VUL4J-13, VUL4J-34, VUL4J-41, VUL4J-63, VUL4J-65, VUL4J-66
- **apró módosítások a biztonságosabb kód érdekében:** a VUL4J-20 esetében a modell típusellenőrzést végez, mielőtt végrehajtaná a típuskonverziót, a VUL4J-12 javítása pedig kis változtatással elkerülte a végtelen ciklusok lehetőségét.

Mennyire konzisztensek a GPT-4 javításai az egyes futások és CWE kategóriák között?

Az eredmények alapján a GPT-4 javításai konzisztensek az egyes futtatások között, helyes megoldást nyújtva mindkét alkalommal az esetek 31,58%-ában. Ezek alapján elmondható, hogy a modell válaszai többnyire konzisztensek, de a következetesség nem a sérülékenység típusához, hanem a javítás jellegéhez kapcsolódik.

5. fejezet

Diszkusszió

5.1. Az eredményekről

5.1.1. Manuális értékelés során elfogadott esetek, amelyeket az egységtesztek elutasítottak

Bár a legtöbb példa egyező eredményt hozott mind a manuális, mind az egységtesztes értékelés során, előfordultak esetek, ahol a unit tesztek elutasították a javítást. Igaz, csupán néhány ilyen példa volt, mégis fontosnak tartjuk kivizsgálni, hogy az eltérést emberi hiba okozta, vagy a tesztek voltak túl specifikusak.

A VUL4J-39 mindkét futása alkalmával a manuális értékelés elfogadta a javítást, a egységtesztek azonban nem. A sebezhetőség a `toString()` metódusban rejlik, ahol a visszaadott string a következő értékeket tartalmazza: *remoteAddress*, *clientId*, *sessionId*. A projekt fejlesztői jogosan tekintették a *sessionId*-t érzékeny adatnak, ahogyan azt a CWE-200 besorolás is tükrözi, így kivették a `toString()` metódusból. Mindkét futás során a GPT nemcsak a *sessionId*-t, hanem a *clientId*-t is eltávolította. A prompt kérdésének megfelelően magyarázattal is szolgált a modell: *"The clientId and sessionId are sensitive because they can be used to impersonate a user or a client."*. Egyetértettünk az indoklással, így elfogadtuk a javítást.

Az egységtesztek azonban megkövetelték a *clientId* meglétét a metódus visszatérési értékében, így elutasították a megoldást. A projekt mélyebb ismerete hiányában nem találtuk az okát, hogy miért szükséges a *clientId* a `toString()` metódusban. Ez az eset hangsúlyozza az emberi értelmezés és az automatizált tesztelési kritériumok összehangolásának fontosságát.

5.1.2. Esetek, ahol az előzetes tesztelés (sanity check) elbukott

A generált eredmények egységtesztes kiértékelése előtt ellenőriztük, hogy valóban megbízhatóak-e a tesztek, vagyis sanity check-et végeztünk.

A VUL4J-30 esetében mind a hibás, mind a fejlesztők által kijavított részt elfogadta a vonatkozó tesztet. Ennek tudatában nem meglepő, miért különbözik a manuális (elutasított) eredménytől a teszt döntése. Bármely megoldás átment, amely érintetlenül hagyta kódrészlet fő funkcionalitását, így ez az eset megbízhatatlannak minősül.

A problémás metódus a `validate()`, feladata a megadott URL validálása. A CWE-20 besorolás arra utal, hogy nem ellenőrzi megfelelően a bemenetként kapott adatokat. A fejlesztők az alkalmazott reguláris kifejezés által elfogadott stringek elejéről és végéről eltávolították a whitespace karakterket. A GPT azonban részben újraírta a metódust és kisebb egységekre bontotta az eredeti `if` blokkot. Az egyetlen újítás egy kivételkezelés, amelyre a validálás helyett alkalmazott URL értelmező függvény miatt volt szükség. Az egyes futtatások megoldásai nem foglalkoznak a whitespace karakterek eltávolításával, így elutasítottuk a javításokat, de a hiányos egységtesztek miatt mégis átmentek a gépi ellenőrzésen.

Hasonló eset a VUL4J-19, ahol szintén az új tesztek hiánya miatt hagytuk figyelmen kívül az eredményeket. A sebezhető metódus a `prepareForDecryption()`, amelyben elmarad a dokumentumot csak olvasásra engedélyező flag beállítása. A fejlesztők a javításban beállítják a flaget a megfelelő objektumon. A tesztesetek hiánya miatt ismét minden olyan kód, amelyben nem történik lényeges változás, elfogadhatónak minősül, így ezt a példát sem tudtuk számításba venni.

5.1.3. Egyéb érdekes esetek

Bár elutasítottuk a GPT megoldását, a VUL4J-25 példáját érdekesnek tartottuk. Mind a manuális, mind az egységtesztek elutasították a javítást. A háttérben a visszaadott stringek eltérése áll:

```
+ <test[?discount=]25%25> // Expected
- <test[%3Fdiscount%3D]25%25> // RUN1 fix
- <test[&#x3f;discount&#x3d;25&#x25;]> // RUN2 fix
```

Kódrészlet 5.1. A VUL4J-25 elvárt és generált kimenetei

A `getValidHref()` metódus feladata egy adott URL biztonságossá tétele, hogy `href` vagy `src` HTML attribútumokban lehessen használni (CWE-79). A GPT megoldása egy speciális függvényt alkalmazott, amely minden különleges karaktert kódolt, ezzel elrontva a linket. Az eredeti javítás a `replaceAll()` függvényt használta csupán a szükséges szimbólumokat kódolva. Az `=` vagy `&` jelek fontos szerepet töltenek be az URL-ekben, de a támadók is felhasználhatják ezeket nem várt paraméterek átadására. A GPT javítása biztonságosabb, azonban a program többi része nincs felkészülve ilyen szintű kódolásra, így sérülnek az alapvető funkciók.

5.2. Keresztvalidáció

Szerettük volna biztosítani, hogy az elért eredmények nem annak köszönhetőek, hogy a GPT esetlegesen rátanult a megoldásokra. A GPT-4 hivatalos dokumentációja szerint 2021 szeptemberéig gyűjtött adatokon tanították a modellt, így ennél frissebb példákat tartalmazó adathalmazt kerestünk. Választásunk a CVEFixes [35]-re esett, egy automatikusan gyűjtött átfogó adathalmazra, amely CVE azonosítóval ellátott sebezhetőségeket tartalmaz. A CVEFixes eredetileg 2021-ben jelent meg, azonban legutóbb 2022. augusztus 28-án¹ frissítették. Olyan Java sebezhetőségekre szűrtünk, amelyeket 2021 szeptembere után javítottak ki, illetve megfeleltek a 3.1. szakaszban leírt kritériumoknak. Összesen 16 ilyen sérülékenységet találtunk, közülük a legrégebbit 2021 októberében javították. Ez biztosítja, hogy a GPT biztosan ne ismerhesse a példák megoldásait.

A CVEFixes egy automatikusan összeállított adathalmaz, így a VUL4J-jel ellentétben nem tartalmaz egységteszteket. Saját teszteseteket írni időigényes feladat lett volna, így a CVEFixes példáit csak manuálisan értékeltük a 3. fejezetben leírtak szerint.

Eredmények. A CVEFixes adathalmazt megállapításainkat igazolására használtuk, valamint arra, hogy megmutassuk, a GPT nem tanult rá az egyes példák megoldására. Amennyiben hasonló eredményeket érünk el az eddig GPT által nem látott adatokon, az megerősíti a tapasztalatainkat. A manuális kiértékelés 37,50%-os javítási arányt mutatott², ami összhangban van a VUL4J adatbázissal kapott eredményekkel. A generált szöveges utasításokat az esetek 81,25%-ában találtuk relevánsnak, a válaszok 43,75%-a pedig helyes magyarázatot adott, helytelen kódrészlettel. Összességében a CVEFixes eredményei arra utalnak, hogy a VUL4J adathalmazon kapott eredményeink nem annak köszönhetőek, hogy a GPT-4 rátanult volna a megoldásokra.

¹A legfrissebb verzió itt érhető el: [10.5281/zenodo.7029359](https://zenodo.org/record/7029359)

²Részletes eredmények a függelékben.

6. fejezet

Eredményeinket veszélyeztető tényezők

Az eredmények minőségét nagy mértékben befolyásolhatja a használt adatforrás. Választásunk a VUL4J [7] adathalmazra esett, sérülékenységek gondosan összeválogatott gyűjteményére, amelyet más munkák is alkalmaztak [36, 37, 38, 39]. Figyelembe kell azonban vennünk, hogy az adatbázist kézzel állították össze, így nem feltétlenül tartalmazza a sebezhetőségek teljes spektrumát. A változatosság növelésére egy másik adathalmaz, a CVEFixes egyes példáit is megvizsgáltuk.

A VUL4J legtöbb elemét 2021 előtt javították ki, így fennáll annak a lehetősége, hogy a GPT-4 megtanulta a sebezhető kódrészletek megoldását. Ennek a valószínűségének a csökkentésére, a CVEFixes azon példáit vizsgáltuk, amelyeket már nem tartalmazhat a modell tanítóhalmaza.

A modell nemdeterminisztikus természetű, így ugyanarra a bemenetre más és más kimenetet adhat. A promptok kétszeri futtatásával csökkentettük annak esélyét, hogy a kapott eredmények a véletlen művei.

Számításba kell vennünk a manuális kiértékelés során esetlegesen elkövetett emberi hibákat is. Ezek esélyét úgy próbáltuk minimalizálni, hogy előre meghatároztuk az értékelési szempontokat, és a lehető legpontosabban próbáltuk ezeket követni. Az eredményeket az értékelő személy mellett egy további kutatótárs is felülvizsgálta.

Figyelembe kell vennünk annak lehetőségét is, hogy a saját keretrendszerünk tartalmazott hibákat. Ennek megelőzése érdekében alapos forráskódelLENŐRZÉST (*code review*) végeztünk, továbbá legalább egy kutatótárs tesztelte a végleges szkripteket.

7. fejezet

Összegzés

Munkánk bemutatja a GPT-4 sérülékenység-javító képességét valóság-hű környezetben. Más munkákkal ellentétben törekedtünk a lehető legpontosabban követni a gyakorlati alkalmazások körülményeit, ipari nagyrendszerekből származó sérülékenységekkel. Értékelési módszerünk objektív egységteszteket (*unit test*) és manuális validációt is alkalmazott. Eredményeinkből kiderül, hogy a nagy nyelvi modellek képesek használható javításokkal szolgálni a fejlesztők számára.

Tanulmányunk fő megállapításai: 1) a GPT-4 automatikus programjavító képessége valós példákon messze nem tökéletes, de ígéretes eredményeket mutat. 2) Bár a GPT-4 számára kihívást jelenthet bonyolultabb kódjavítások generálása, a szöveges leírások az esetek nagy részében hasznosnak bizonyultak. 3) A GPT-4 javításai elsősorban egyszerű, jó praktikákon (*best practices*) alapulnak.

Összegezve: a GPT-4 jelentős előrelépést jelent az automatikus programjavítás területén, azonban további kutatások és fejlesztések szükségesek a teljesítmény javítására, hogy a modell bonyolultabb biztonsági sérülékenységeket is hatékonyan tudjon kezelni.

Az elért eredményekből egy nemzetközi konferenciacikk is készült az International Conference on Evaluation and Assessment in Software Engineering (EASE) konferenciára.

Függelék

Promptok

```
Act like a very senior Java developer who is especially great in fixing vulnerabilities.↵
Make sure to fix this vulnerable code. Delete any unnecessary code if necessary.
```

```
{source_code}
```

```
Generate 3 fixes as git diff-like format and explain what and how you fixed it. Choose ↵
the best and most optimized one. You need to know the following about the vulnerability:↵
{The description of the CWE}
```

```
It is very important for me, please do your best!
```

Kódrészlet 7.1. DIFF3 prompt

```
You are a senior cyber security specialist and Java developer. You will be given the ↵
source code of a Java class where you will find only one vulnerable method. Your task is↵
to patch any vulnerabilities in this method. You will also have access to the ↵
description of the Java project and the issue.
```

```
Project description: {The description of the project}
```

```
Issue description: {The description of the CWE}
```

```
{source code}
```

```
It is very important for me, please patch the vulnerable code based on your best ↵
knowledge based on the given descriptions. Provide the patched source code and write ↵
detailed comments about your fix explaining your way of thinking.
```

Kódrészlet 7.2. FULL prompt

You are an experienced ethical hacker and expert Java developer. You are given the ↵
following source code:

{source code}

You must find and list any possible vulnerabilities the code might contain. With the ↵
help of your exploit patch the vulnerable source code so that it would be safe to use. ↵
Provide a detailed report of your thinking and the patched source code. Find a way to ↵
exploit a vulnerability that best matches this description: {The description of the CWE}↵
With the help of your exploit patch the vulnerable source code so that it would be safe↵
to use. Provide a detailed report of your thinking and the patched source code. It is ↵
very important for me, please do your best!

Kódrészlet 7.3. EXPLOIT prompt

You are an expert cyber security specialist and Java developer. You are given the ↵
following source code:

{source code}

This code is suspected to contain {The title of the CWE} vulnerability. This task is ↵
very important for me, please patch the vulnerable code with your best knowledge. You ↵
may also improve or delete the existing code if necessary or fix additional ↵
vulnerabilities, but it must NOT change the original code functionality. Provide the ↵
patched source code and detailed comments of your changes.

Kódrészlet 7.4. IMPROVE prompt

Táblázatok

		VUL4J-4	VUL4J-6	VUL4J-16	VUL4J-18	VUL4J-24	VUL4J-40	VUL4J-44	VUL4J-48	VUL4J-54	VUL4J-65
DIFF3	code_comp	0	0	1	1	1	1	1	0	1	0
	code_corr	0	0	0.5	0.5	0	0	0	0	0.5	0
	code_rel	0	0	0	1	0	0	0	0	1	0
	text_corr	0	0	0.5	0	0	0	0	0	0.5	0
	text_rel	0	0	0	1	1	0	0	0	1	0
	VUL4J TEST	0	0	0	0	0	0	0	1	0	0
FULL	code_comp	0	0	1	1	1	0	1	1	0	1
	code_corr	0	0	0.5	0	0.5	0.5	1	0.5	0.5	1
	code_rel	0	0	1	0	1	1	1	1	1	1
	text_corr	0	0	0.5	0	1	0.5	0.5	0.5	0.5	1
	text_rel	0	0	1	1	1	1	1	1	1	1
	VUL4J TEST	0	0	0	0	0	0	0	0	0	1
IMPROVE	code_comp	1	1	0	1	1	0	1	1	1	1
	code_corr	0	0	0.5	0.5	0.5	0.5	0	0.5	0	1
	code_rel	0	0	0	0	1	1	0	1	0	1
	text_corr	0	0	0.5	0.5	1	0.5	0	0.5	0	1
	text_rel	0	0	0	0	1	1	0	1	0	1
	VUL4J TEST	0	0	0	0	0	0	0	0	0	0
EXPLOIT	code_comp	1	0	1	1	1	0	0	1	1	1
	code_corr	0	0	0.5	0.5	0.5	0.5	0	0.5	0.5	1
	code_rel	0	0	0	0	1	1	0	1	1	1
	text_corr	0	0	0.5	0.5	1	0.5	0	0.5	0.5	1
	text_rel	0	0	0	1	1	1	0	1	1	1
	VUL4J TEST	0	0	0	0	0	0	0	0	0	0

7.1. ábra. A promptok eredményei a GPT-3.5 modellen futtatva

		VUL4J-4	VUL4J-6	VUL4J-16	VUL4J-18	VUL4J-24	VUL4J-40	VUL4J-44	VUL4J-48	VUL4J-54	VUL4J-65
DIFF3	code_comp	1	1	1	1	1	1	1	1	1	1
	code_corr	0	0	0	0	0.5	0.5	0	0.5	0.5	1
	code_rel	0	0	0	1	1	0	0	1	0	1
	text_corr	0	0	0	0	0.5	0.5	0	0.5	0.5	1
	text_rel	0	0	0	1	1	1	0	1	1	1
	VUL4J TEST	0	0	0	0	0	0	0	0	0	0
FULL	code_comp	1	1	1	1	1	1	1	1	1	1
	code_corr	0	0	0	0.5	1	0.5	0	0.5	0.5	1
	code_rel	0	0	0	1	1	1	1	1	1	1
	text_corr	0	0	0	0.5	1	0.5	0	0.5	0.5	1
	text_rel	0	0	0	1	1	1	1	1	1	1
	VUL4J TEST	0	0	0	0	1	0	0	0	0	0
IMPROVE	code_comp	1	1	1	1	1	1	1	1	1	1
	code_corr	0	0	0	0.5	1	0.5	0	0.5	0.5	1
	code_rel	0	0	0	1	1	1	0	1	1	1
	text_corr	0	0	0	0.5	1	0.5	0	0.5	0.5	1
	text_rel	0	0	0	1	1	1	0	1	1	1
	VUL4J TEST	0	0	0	0	0	0	0	0	0	0
EXPLOIT	code_comp	1	1	0	1	1	1	1	1	1	1
	code_corr	0	0	0	0.5	1	0.5	0	0.5	0.5	1
	code_rel	0	0	1	1	1	1	0	1	1	1
	text_corr	0	0	0	0.5	1	0.5	0	0.5	0.5	1
	text_rel	0	0	1	1	1	1	0	1	1	1
	VUL4J TEST	0	0	0	0	1	0	0	0	0	1

7.2. ábra. A promptok eredményei a GPT-4 modellen futtatva (1)

		VUL4J-4	VUL4J-6	VUL4J-16	VUL4J-18	VUL4J-24	VUL4J-40	VUL4J-44	VUL4J-48	VUL4J-54	VUL4J-65
DIFF3	code_comp	0	0	1	1	1	1	1	1	1	1
	code_corr	0	0	0	0.5	1	0	0	0.5	0	1
	code_rel	0	0	0	1	1	0	0	1	1	1
	text_corr	0	0	0	0.5	1	0	0	0.5	0	1
	text_rel	0	0	0	1	1	0	0	1	1	1
	VUL4J TEST	0	0	0	0	0	0	0	0	0	0
FULL	code_comp	1	1	1	1	1	1	1	1	1	1
	code_corr	0	0	0	0.5	1	0	0	0.5	0.5	1
	code_rel	0	0	0	1	1	0	1	1	1	1
	text_corr	0	1	0	0.5	1	0	0	0.5	0.5	1
	text_rel	0	1	0	1	1	0	1	1	1	1
	VUL4J TEST	0	0	0	0	0	0	0	0	0	0
IMPROVE	code_comp	1	1	1	1	1	1	1	1	1	1
	code_corr	0	0	0	0.5	1	0	0	0.5	0.5	1
	code_rel	0	1	0	1	1	0	0	1	1	1
	text_corr	0	0	0	0.5	1	0	0	0.5	0.5	1
	text_rel	0	1	0	1	1	0	0	1	1	1
	VUL4J TEST	0	0	0	0	1	0	0	0	0	1
EXPLOIT	code_comp	1	0	0	1	1	1	1	1	0	1
	code_corr	0	0	0	0.5	1	0	0	0.5	0.5	1
	code_rel	0	0	0	1	1	0	0	1	1	1
	text_corr	0	0	0	0.5	1	0	0	0.5	0.5	1
	text_rel	0	0	0	1	1	0	0	1	1	1
	VUL4J TEST	0	0	0	0	0	0	0	0	0	0

7.3. ábra. A promptok eredményei a GPT-4 modellen futtatva (2)

PÉLDA	CWE	ÉV	code_comp	code_corr	code_rel	text_corr	text_rel	MANUÁLIS
CVEFIXES-1	CWE-79	2022	1	0	1	0	1	0
CVEFIXES-2	CWE-22	2022	1	0.5	1	0.5	1	0
CVEFIXES-5	CWE-378	2022	1	0.5	1	1	1	0
CVEFIXES-6	CWE-22	2022	1	1	1	1	1	1
CVEFIXES-10	CWE-918	2022	0	1	1	1	1	1
CVEFIXES-12	CWE-79	2022	1	0.5	1	0.5	1	0
CVEFIXES-22	CWE-611	2021	1	1	1	1	1	1
CVEFIXES-30	CWE-89	2022	1	0	0	0.5	1	0
CVEFIXES-33	CWE-835	2022	0	0.5	1	0.5	1	0
CVEFIXES-36	CWE-287	2022	0	0	1	1	1	0
CVEFIXES-39	CWE-79	2022	1	1	1	1	1	1
CVEFIXES-40	CWE-400	2022	0	0	1	0.5	1	0
CVEFIXES-43	CWE-918	2022	0	0	1	0	0	0
CVEFIXES-47	CWE-22	2022	1	1	1	1	1	1
CVEFIXES-51	CWE-918	2022	0	0	0	0	0	0
CVEFIXES-52	CWE-611	2022	1	1	1	1	1	1

7.4. ábra. A CVEFixes adathalmaz eredményei a GPT-4 modellen futtatva

PÉLDA	CWE	ÉV	code_comp	code_corr	code_rel	text_corr	text_rel	MANUÁLIS	EGYSÉGTESZT
VUL4J-1	CWE-20	2017	0	0	0	0	0	0	0
VUL4J-2	CWE-611	2016	1	1	1	1	1	1	-
VUL4J-3	Not Mapping	2015	1	1	1	1	1	1	1
VUL4J-4	Not Mapping	2015	1	0	0	0	0	0	0
VUL4J-5	Not Mapping	2018	1	1	1	1	1	1	1
VUL4J-6	CWE-835	2017	1	0.5	1	0.5	1	0	-
VUL4J-7	CWE-835	2018	1	0.5	1	0.5	1	0	0
VUL4J-8	CWE-835	2019	1	0	0	0	0	0	0
VUL4J-10	CWE-20	2013	1	0	0	0	0	0	0
VUL4J-12	CWE-835	2019	1	1	1	1	1	1	1
VUL4J-13	CWE-835	2019	1	1	1	1	1	1	1
VUL4J-15	CWE-611	2016	1	0	0	0	0	0	0
VUL4J-16	CWE-264	2015	1	0	0	0	0	0	0
VUL4J-17	Not Mapping	2017	1	0	0	0	0	0	0
VUL4J-18	CWE-22	2019	1	0	0	0	0	0	0
VUL4J-19	Not Mapping	2016	1	0	0	0	0	0	-
VUL4J-20	Not Mapping	2018	1	1	1	1	1	1	1
VUL4J-22	CWE-284	2016	1	0	0	0	0	0	0
VUL4J-23	CWE-79	2016	1	0.5	1	0.5	1	0	0
VUL4J-24	CWE-611	2016	1	0.5	1	0.5	1	0	0
VUL4J-25	CWE-79	2017	1	0	1	1	1	0	0
VUL4J-26	CWE-20	2016	1	0	0	0	0	0	0
VUL4J-30	CWE-20	2016	1	0	0	0	0	0	-
VUL4J-31	Not Mapping	2016	1	0	0	0	0	0	0
VUL4J-34	CWE-79	2016	1	1	1	1	1	1	1
VUL4J-39	CWE-200	2018	1	1	1	1	1	1	0
VUL4J-40	CWE-287	2019	1	0	0	0	0	0	0
VUL4J-41	CWE-22	2018	0	1	1	1	1	1	1
VUL4J-44	CWE-310	2016	1	0	0	0	0	0	0
VUL4J-45	CWE-74	2018	1	0.5	1	1	1	0	0
VUL4J-48	CWE-20	2018	1	0.5	1	0.5	1	0	0
VUL4J-50	CWE-79	2013	1	0.5	1	0.5	1	0	0
VUL4J-51	CWE-918	2018	1	0	0	0.5	0	0	-
VUL4J-53	CWE-835	2018	0	0	0	0	0	0	0
VUL4J-54	CWE-502	2017	1	0.5	1	0.5	0	0	0
VUL4J-55	CWE-835	2018	1	0.5	1	0.5	1	0	0
VUL4J-57	CWE-532	2017	1	0	0	0	0	0	0
VUL4J-61	CWE-611	2018	1	1	1	1	1	1	1
VUL4J-62	CWE-287	2018	1	0	0	0	0	0	-
VUL4J-63	Not Mapping	2018	1	1	1	1	1	1	1
VUL4J-64	CWE-611	2018	1	1	1	1	1	1	1
VUL4J-65	CWE-22	2018	1	0.5	1	1	1	1	1
VUL4J-66	CWE-20	2020	1	1	1	1	1	1	1
VUL4J-69	CWE-22	2016	1	0.5	1	1	1	1	-
VUL4J-74	CWE-332	2019	1	0	1	0	1	0	-
VUL4J-79	CWE-22	2018	0	0.5	1	1	1	0	-

7.5. ábra. A VUL4J adathalmaz eredményei a GPT-4 modellen futtatva (1)

PÉLDA	CWE	ÉV	code_comp	code_corr	code_rel	text_corr	text_rel	MANUÁLIS	EGYSÉGTESZT
VUL4J-1	CWE-20	2017	0	0	0	0	0	0	0
VUL4J-2	CWE-611	2016	1	0	0	0	0	0	-
VUL4J-3	Not Mapping	2015	1	1	1	1	1	1	1
VUL4J-4	Not Mapping	2015	1	0	0	0	0	0	0
VUL4J-5	Not Mapping	2018	1	1	1	1	1	1	1
VUL4J-6	CWE-835	2017	1	0	0	0	0	0	-
VUL4J-7	CWE-835	2018	1	0	0	0	0	0	0
VUL4J-8	CWE-835	2019	1	0	0	0	0	0	0
VUL4J-10	CWE-20	2013	1	0	0	0	0	0	0
VUL4J-12	CWE-835	2019	1	1	1	1	1	1	1
VUL4J-13	CWE-835	2019	1	1	1	1	1	1	1
VUL4J-15	CWE-611	2016	1	0	0	0	0	0	0
VUL4J-16	CWE-264	2015	1	0.5	0	0.5	0	0	0
VUL4J-17	Not Mapping	2017	1	0	0	0	0	0	0
VUL4J-18	CWE-22	2019	1	0	0	0	0	0	0
VUL4J-19	Not Mapping	2016	1	0.5	1	0.5	1	0	-
VUL4J-20	Not Mapping	2018	1	1	1	1	1	1	1
VUL4J-22	CWE-284	2016	0	0	0	0	0	0	0
VUL4J-23	CWE-79	2016	1	0	0	0	0	0	0
VUL4J-24	CWE-611	2016	1	1	1	1	1	1	1
VUL4J-25	CWE-79	2017	1	0	0	0.5	0	0	0
VUL4J-26	CWE-20	2016	1	0	0	0	0	0	0
VUL4J-30	CWE-20	2016	1	0	0	0	0	0	-
VUL4J-31	Not Mapping	2016	1	0	0	0	0	0	0
VUL4J-34	CWE-79	2016	1	1	1	1	1	1	1
VUL4J-39	CWE-200	2018	1	1	1	1	1	1	0
VUL4J-40	CWE-287	2019	1	0	0	0	0	0	0
VUL4J-41	CWE-22	2018	1	1	1	1	1	1	1
VUL4J-44	CWE-310	2016	1	0	0	0	0	0	0
VUL4J-45	CWE-74	2018	0	0	1	1	1	0	0
VUL4J-48	CWE-20	2018	1	0.5	1	0.5	1	0	0
VUL4J-50	CWE-79	2013	1	1	1	1	1	1	1
VUL4J-51	CWE-918	2018	1	0	0	0	0	0	-
VUL4J-53	CWE-835	2018	0	0	0	0	0	0	0
VUL4J-54	CWE-502	2017	0	0	0	0	0	0	0
VUL4J-55	CWE-835	2018	0	0.5	1	1	1	0	0
VUL4J-57	CWE-532	2017	1	0	0	0	0	0	0
VUL4J-61	CWE-611	2018	1	1	1	1	1	1	1
VUL4J-62	CWE-287	2018	1	0	0	0	0	0	-
VUL4J-63	Not Mapping	2018	1	1	1	1	1	1	1
VUL4J-64	CWE-611	2018	1	1	1	1	1	1	1
VUL4J-65	CWE-22	2018	1	0.5	1	1	1	1	1
VUL4J-66	CWE-20	2020	1	1	1	1	1	1	1
VUL4J-69	CWE-22	2016	1	1	1	1	1	1	-
VUL4J-74	CWE-332	2019	1	0	1	0	1	0	-
VUL4J-79	CWE-22	2018	1	0.5	1	1	1	0	-

7.6. ábra. A VUL4J adathalmaz eredményei a GPT-4 modellen futtatva (2)

Irodalomjegyzék

- [1] Steve Rathje, Dan-Mircea Mirea, Ilia Sucholutsky, Raja Marjeh, Claire Robertson, and Jay J Van Bavel. Gpt is an effective tool for multilingual psychological text analysis. 2023.
- [2] Kunming Cheng, Qiang Guo, Yongbin He, Yanqiu Lu, Ruijie Xie, Cheng Li, and Haiyang Wu. Artificial intelligence in sports medicine: could gpt-4 make human doctors obsolete? *Annals of Biomedical Engineering*, pages 1–5, 2023.
- [3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS’17*, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.
- [4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.
- [5] 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 *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics.
- [6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code.(2021). *arXiv preprint arXiv:2107.03374*, 2021.

- [7] 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 *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 464–468, New York, NY, USA, 2022. Association for Computing Machinery.
- [8] Common Vulnerabilities and Exposures. <https://cve.mitre.org/>, 2023. Accessed: 2023-10-13.
- [9] Common Weaknesses Enumeration. <https://cwe.mitre.org/>, 2023. Accessed: 2023-10-13.
- [10] D. Sobania, M. Briesch, C. Hanna, and J. Petke. An analysis of the automatic bug fixing performance of chatgpt. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 23–30, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.
- [11] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. How effective are neural networks for fixing security vulnerabilities. *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023.
- [12] Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. Automatic program repair. *IEEE Software*, 38(4):22–27, 2021.
- [13] Incheon Paik and Jun-Wei Wang. Improving text-to-code generation with features of code graph on gpt-2. *Electronics*, 10:2706, 11 2021.
- [14] Márk Lajkó, Viktor Csuvik, and László Vidács. Towards javascript program repair with generative pre-trained transformer (gpt-2). In *Proceedings of the Third International Workshop on Automated Program Repair*, APR '22, page 61–68, New York, NY, USA, 2022. Association for Computing Machinery.
- [15] Pe’ter Gyimesi, Be’la Vancsics, Andrea Stocco, Davood Mazinanian, A’rpa’d Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: a benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101, 2019.

- [16] Márk Lajkó, Dániel Horváth, Viktor Csuvik, and László Vidács. Fine-tuning gpt-2 to patch programs, is it worth it? In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Ana Maria A. C. Rocha, and Chiara Garau, editors, *Computational Science and Its Applications – ICCSA 2022 Workshops*, pages 79–91, Cham, 2022. Springer International Publishing.
- [17] Julian Aron Prenner, Hlib Babii, and Romain Robbes. Can openai’s codex fix bugs?: An evaluation on quixbugs. In *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 69–75, 2022.
- [18] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH Companion 2017*, page 55–56, New York, NY, USA, 2017. Association for Computing Machinery.
- [19] Yiannis Charalambous, Norbert Tihanyi, Youcheng Sun, Mohamed Amine Ferrag, and Lucas Cordeiro. A new era in software security: Towards self-healing software via large language models and formal verification, 05 2023.
- [20] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356, 2023.
- [21] Noor Nashid, Mifta Sintaha, and Ali Mesbah. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2450–2462, 2023.
- [22] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*. Association for Computing Machinery, 2023.
- [23] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 07 2014.

- [24] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.
- [25] Chunqiu Steven Xia and Lingming Zhang. Conversational automated program repair, 2023.
- [26] Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: Revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 959–971, New York, NY, USA, 2022. Association for Computing Machinery.
- [27] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1469–1481. IEEE Press, 2023.
- [28] Seemanta Saha et al. Harnessing evolution for multi-hunk program repair. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 13–24. IEEE, 2019.
- [29] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. A survey of learning-based automated program repair. *arXiv preprint arXiv:2301.03270*, 2023.
- [30] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114, 2020.
- [31] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019.

- [32] Wei Yuan, Qunjun Zhang, Tiede He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. Circle: Continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 678–690, 2022.
- [33] 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.
- [34] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.
- [35] Guru Bhandari, Amara Naseer, and Leon Moonen. CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21)*, page 10. ACM, 2021.
- [36] Eduard Pinconschi, Quang-Cuong Bui, Rui Abreu, Pedro Adão, and Riccardo Scandariato. Maestro: A platform for benchmarking automatic program repair tools on software vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 789–792, 2022.
- [37] Aurora Papotti, Ranindya Paramitha, and Fabio Massacci. On the acceptance by code reviewers of candidate security patches suggested by automated program repair tools. *arXiv preprint arXiv:2209.07211*, 2022.
- [38] Jens Dietrich, Shawn Rasheed, and Alexander Jordan. On the security blind spots of software composition analysis. *arXiv preprint arXiv:2306.05534*, 2023.
- [39] Aayush Garg. Guiding quality assurance through context aware learning, 2023.