VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

TESTOVACÍ DVOJNÍK PRO ZAŘÍZENÍ A SOUBORY

DEVICE AND FILE TEST DOUBLE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **DANIEL ZÁLEŽÁK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. ALEŠ SMRČKA, Ph.D.**
SUPERVISOR

BRNO 2025

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

160695

Ústav:	Ústav inteligentních systémů (UITS)
Student:	**Záležák Daniel**
Program:	Informační technologie
Název:	**Testovací dvojník pro zařízení a soubory**
Kategorie:	Analýza a testování softwaru
Akademický rok:	2024/25

Zadání:

1. Nastudujte techniky testovacích dvojníků. Nastudujte principy operačního systému GNU/Linux a uživatelských procesů pro práci se speciálními soubory.
2. Navrhněte přístup testování programů pracující se speciálními soubory. Testování by mělo využívat techniku mocking těchto souborů.
3. Implementujte testovací rámec využívající techniku mocking pro speciální soubory.
4. Vyhodnoťte správnost implementace na sadě umělých testů.

Literatura:

- Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: reducing software overhead in file systems for persistent memory. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19). ACM, NY, USA, 494–508. https://doi.org/10.1145/3341301.3359631
- Dynamic linker/loader, manuálová stránka. https://linux.die.net/man/8/ld-linux

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Smrčka Aleš, Ing., Ph.D.**
Konzultant:	Pavel Moravec
Vedoucí ústavu:	Kočí Radek, Ing., Ph.D.
Datum zadání:	1.11.2024
Termín pro odevzdání:	14.5.2025
Datum schválení:	31.10.2024

Abstrakt

Táto práca rieši problematiku testovania softvéru. V istých prípadoch nie je možné zhotoviť vhodné prostredie na otestovanie daného softvéru. Cieľom tejto práce je vytvoriť testovacieho dvojníka, ktorý dokáže nasimulovať vhodné prostredie. Hlavným zameraním tohto testovacieho dvojníka je práca so špeciálnymi a bežnými súbormi. Zvolený problém je riešený vytvorením vlastnej zdieľanej knižnice a mechanizmu LD_PRELOAD, ktorý umožňuje nahradiť volania knižnice GNU C na systémoch GNU/Linux. Testovací dvojník je navrhnutý na testovanie diagnostického softvéru sos od Red Hat. Je sa však využiteľný aj na testovanie iných aplikácií.

Abstract

This work addresses the issue of software testing. In some cases, it is not possible to create a suitable environment for testing the given software. The goal of this work is to create a test double that can simulate an appropriate environment. The main focus of this test double is working with special and common files. The chosen problem is solved by creating a custom shared library and the LD_PRELOAD mechanism, which allows overriding GNU C library calls on GNU/Linux systems. The test double is designed for testing the diagnostic software sos from Red Hat, but it can also be used for testing other applications.

Kľúčové slová

Testovací dvojník, LD_PRELOAD, zdieľaná knižnica, testovanie softvéru, špeciálne súbory

Keywords

Test double, LD_PRELOAD, shared library, software testing, special files

Citácia

ZÁLEŽÁK, Daniel. *Testovací dvojník pro zařízení a soubory.* Brno, 2025. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Testovací dvojník pro zařízení a soubory

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pana Ing. Aleša Smrčku, Ph.D. Ďalšie informácie mi poskytol konzultant Mgr. Pavel Moravec, Ph.D. Uviedol som všetky literárne premene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Daniel Záležák
13. mája 2025

Poďakovanie

Týmto by som sa rád poďakoval vedúcemu mojej práce Ing. Alešovi Smrčkovi, Ph.D. za odborné vedenie, rady a pomoc pri písaní tejto práce. Taktiež by som sa chcel poďakovať konzultantovi Mgr. Pavlovi Moravcovi, Ph.D. za poskytnutie všetkých potrebných informácií na úspešné dokončenie práce, spoločnosti Red Hat za možnosť pracovať na tejto téme a mojej rodine za neustálu podporu počas celého štúdia.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Kapitola 1

Úvod

Jednou z najdôležitejších častí procesu vydania softvéru je jeho dôkladné otestovanie. Avšak nie vždy je tester schopný vytvoriť vhodné prostredie pre správne otestovanie softvéru. Správny chod softvéru môže byť závislý na rôznych faktoroch, ako je napríklad prítomnosť určitých súborov, zariadení alebo ďalších softvérov v systéme. Vtedy je vhodné použiť nástroj, ktorý je schopný nasimulovať vyhovujúce prostredie, takzvaný testovací dvojník. Ako sa v práci spomína, existuje viacero druhov testovacích dvojníkov. Každý je vhodný na konkrétny druh problému. Táto práca sa zameriava konkrétne na vytvorenie zdieľanej knižnice, ktorá je pomocou mechanizmu prednačítania dynamických knižníc pripojená k programom pred ostatné knižnice a vďaka tomu nahrádza funkcie štandardnej knižnice **GNU C Library** programovacieho jazyka C. Funkcie knižnice **GNU C Library** slúžia ako obaly pre systémové volania do jadra, aby zjednodušili ich používanie pre procesy. Konkrétne riešenie problému bolo v tejto práci vybrané na základe porovnania a vyhodnotenia výhod a nevýhod vybraných existujúcich riešení.

V druhej kapitole tejto práce je spracovaná problematika testovania softvéru a predstavené problémy, na ktoré môže tester pri testovaní softvéru naraziť. V tejto kapitole sú taktiež dôkladne opísané typy a techniky testovania softvéru, rôzne druhy testovacích dvojníkov a vysvetlené rôzne pojmy, ktoré sa týkajú tejto práce.

Tretia kapitola sa zameriava na návrh testovacieho dvojníka. Obsahuje analýzu základných požiadaviek, ktoré by mal testovací dvojník spĺňať na správne fungovanie v praxi. Taktiež obsahuje dôkladné spracovanie a opis existujúcich nástrojov, ktoré by mohli slúžiť ako konkrétny testovací dvojník v tejto práci, a nakoniec riešenie, ktoré bolo zvolené na túto prácu.

Štvrtá kapitola podrobne opisuje postup pri získavaní všetkých potrebných údajov, nástroje, ktoré boli pri tom použité, a samotnú implementáciu zdieľanej knižnice a jej funkcií a vlastností.

Testovanie funkcionality testovacieho dvojníka je spracované v piatej kapitole tejto práce. Na testovacom dvojníkovi boli vykonané rôzne druhy testov. Následne boli získané výsledky spracované do tabuliek a porovnané s očakávanými výsledkami.

Kapitola 2

Problematika testovania softwaru

Táto kapitola sa zaoberá testovaním softvéru a problémami, s ktorými sa môže tester stretnúť, ale taktiež aj konkrétnymi riešeniami týchto problémov.

Softvérové systémy sú neoddeliteľnou súčasťou nášho každodenného života. Softvér, ktorý nefunguje správne, môže spôsobiť množstvo problémov, vrátane straty peňazí, času alebo obchodnej reputácie a v extrémnych prípadoch aj zranenia alebo úmrtia. Testovanie softvéru hodnotí kvalitu softvéru a pomáha znižovať riziko zlyhania softvéru v prevádzke. [29]

2.1 Testovanie softwaru

Testovanie softvéru je proces, alebo séria procesov, navrhnutých tak, aby sa zabezpečilo, že počítačový kód robí to, na čo bol navrhnutý, a naopak, že nerobí nič neúmyselné. Softvér by mal byť predvídateľný a konzistentný, bez prekvapení pre používateľov. [20] Softvérové testovanie je dôležitou súčasťou vývoja softvéru. Nie je to jednorázová činnosť, ale skôr dlhodobý proces, ktorý začína v skorej fáze vývoja a pokračuje cez celý životný cyklus daného softvéru. Tento proces zahŕňa plánovanie, navrhovanie, spúšťanie a následné vyhodnocovanie testov, aby sa identifikovali a opravili všetky chyby a zlepšila kvalita daného softvéru. [16]

Jednou z najbežnejších mylných predstáv o testovaní softvéru je, že je to iba o hľadaní chýb v softvéri. Aj keď hľadanie chýb je dôležitou súčasťou testovania, nie je to jediný cieľ. Testovanie je taktiež o overovaní, že softvér spĺňa požiadavky užívateľov, jeho používanie nie je zložité a že pracuje podľa očakávaní. Taktiež, že je bezpečný, stabilný a efektívny. [16]

Typy testovania

Existuje viacero typov testovania softvéru. Každý s jedinečným zameraním a cieľom. Viď obrázok 2.2. Niektoré typy zahŕňajú jednotkové testovanie, integračné testovanie, systémové testovanie, akceptačné testovanie, výkonnostné testovanie, regresné testovanie alebo bezpečnostné testovanie. Každý typ má svoj vlastný špecifický účel a je zameraný na rôzne fázy vývojového cyklu softvéru. Základné rozdelenie testovania: [16]

- Funkcionálne testovanie
- Nefunkcionálnené testovanie

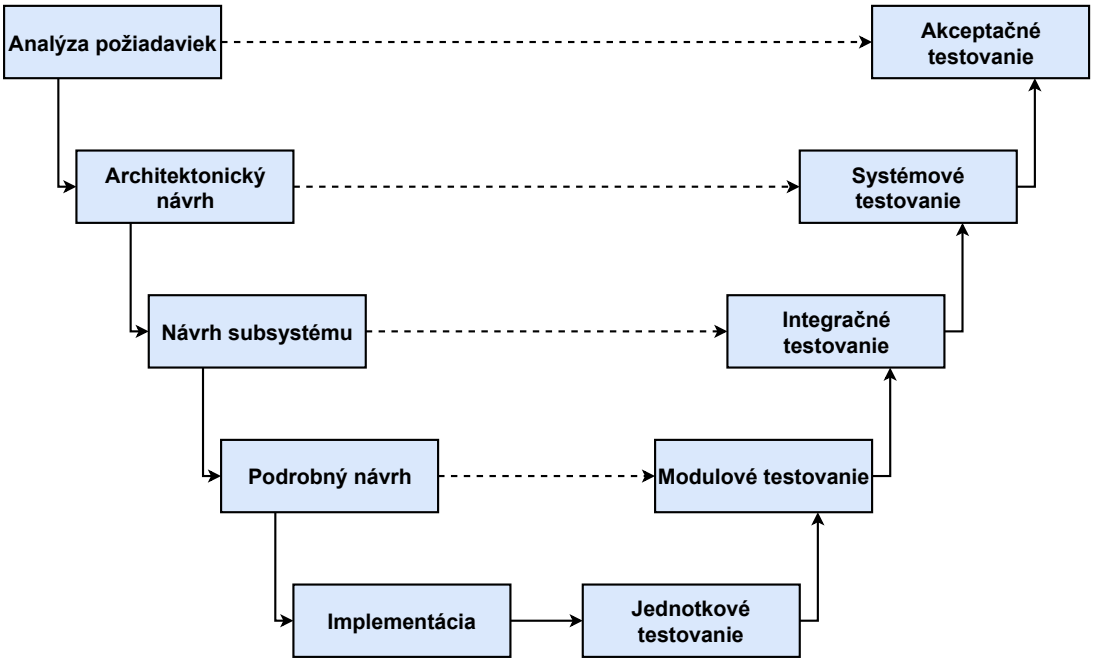
Funkcionálne testovanie

Tento typ testovania sa používa na overenie, že softvér pracuje podľa očakávaní a spĺňa všetky požiadavky užívateľa. Hlavným zameraním je funkčnosť softvéru a zahŕňa rôzne testovacie úrovne. [16]

Úrovne funkcionálneho testovania

Testy môžu byť odvodené od požiadaviek a špecifikácií, návrhových artefaktov alebo zdrojového kódu. Každá odlišná úroveň testovania sprevádza rôznu aktivitu vývoja softvéru [1], ako je to možné vidieť na obrázku 2.1.

- **Akceptačné testovanie** – hodnotí softvér vzhľadom na požiadavky.
- **Systémové testovanie** – hodnotí softvér vzhľadom na architektonický návrh.
- **Integračné testovanie** – hodnotí softvér vzhľadom na návrh subsystémov.
- **Modulové testovanie** – hodnotí softvér vzhľadom na podrobný návrh.
- **Jednotkové testovanie** – hodnotí softvér vzhľadom na implementáciu



Obr. 2.1: V-model úrovní funkcionálneho testovania

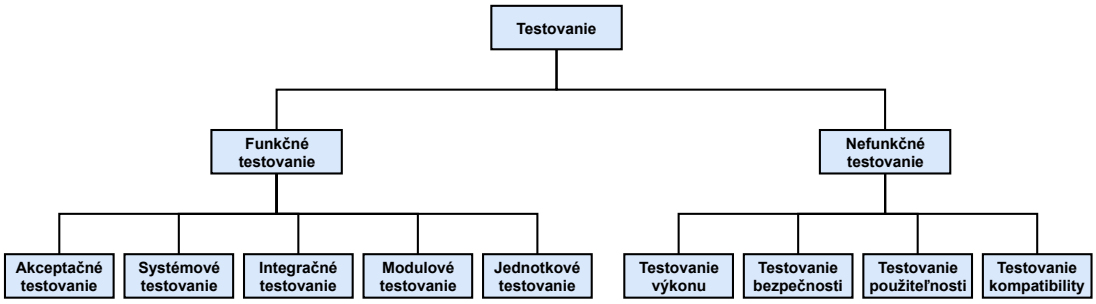
Nefunkcionálne testovanie

Nefunkcionálne testovanie sa zameriava na nefunkcionálne aspekty softvéru, ako sú výkon, bezpečnosť a použiteľnosť. Cieľom je zaistiť, že softvér spĺňa jeho nefunkcionálne požiadavky a poskytuje pozitívny dojem na užívateľa. [16]

Typy nefunkcionálneho testovania

Nefunkcionálne testovanie je možné rozdeliť na tieto typy: [16]

- **Testovanie výkonu** – zahŕňa testovanie výkonu softvéru v rôznych podmienkach
- **Testovanie bezpečnosti** – zahŕňa testovanie bezpečnostných vlastností softvéru voči rôznym útokom
- **Testovanie použiteľnosti** – zahŕňa testovanie užívateľského rozhrania a dojmu užívateľov aby sa zabezpečilo, že sa dá softvér ľahko používať
- **Testovanie kompatibility** – zahŕňa testovanie systému a jeho kompatibility s rôznymi softvérmi, hardvérmi a operačnými systémami

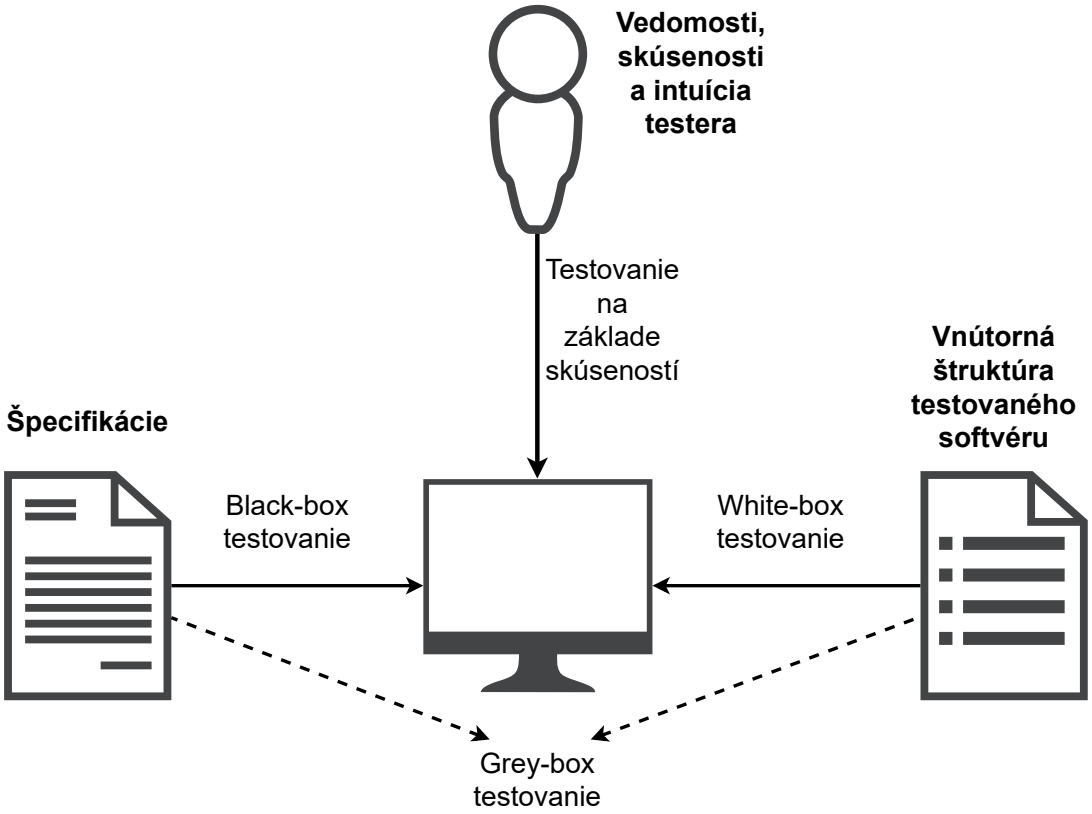


Obr. 2.2: Typy testovania

Techniky testovania

Testovacia technika je aktivita založená na používaní určitých metód na vytvorenie testovacích podmienok, testovacích prípadov a testovacích dát na základe softvérových vedomostí. [29] Viď obrázok 2.3.

- **Black-box testovanie** je testovacia technika skúmajúca funkcionalitu softvéru bez znalosti vnútornej štruktúry kódu daného softvéru. Cieľom techniky black-box je identifikovať defekty vo funkcionalite, používateľnosti a výkonnosti testovaného softvéru na základe vopred známych špecifikácií. [16]
- **White-box testovanie** je naopak testovacia technika, ktorá skúma práve vnútornú štruktúru testovaného softvéru. Cieľom white-box techniky je identifikovať defekty v štruktúre kódu a tak zabezpečiť správnu funkčnosť testovaného softvéru. [16]
- **Grey-box testovanie** je kombinácia techník black-box a white-box. Používa sa ak tester má nejaké znalosti o vnútornej štruktúre testovaného softvéru, ale nemá k nej plný prístup. Cieľom je taktiež zabezpečiť správnu funkčnosť testovaného softvéru. [16]
- **Testovanie na základe skúseností** je technika, pri ktorej záleží na skúsenostiach, vedomostiach a intuícii testera. Často je kombinovaná s ďalšími technikami, čo umožňuje testerom identifikovať chyby, ktoré môžu byť často prehliadnuté. [29]



Obr. 2.3: Typy testovacích techník. Prevzaté z [29]

2.2 Problémy pri testovaní

Táto podkapitola čerpá z článku [36], ktorý sa zaoberá problémami v testovaní.

Na naplánovanie a vykonanie testov musia softvéroví testeri zohľadniť softvér a funkciu, ktorú vykonáva, vstupy a spôsob, akým ich možno kombinovať, ako aj prostredie, v ktorom bude softvér nakoniec fungovať. Tento náročný a časovo vyčerpávajúci proces si vyžaduje technickú vyspelosť a správne plánovanie. Testeri musia mať nielen dobré vývojárske schopnosti, testovanie často vyžaduje veľké množstvo kódovania, ale musia byť tiež znalí vo formálnych jazykoch, teórii grafov a algoritmoch. Tvoriví testeri skutočne dokázali uplatniť mnohé príbuzné výpočtové disciplíny na riešenie testovacích problémov, často s pôsobivými výsledkami.

Aj jednoduchý softvér predstavuje pre testerov prekážky. Aby sme získali jasnejší pohľad na niektoré z prirodzených ťažkostí testovania softvéru, môžeme sa na testovanie pozrieť v štyroch fázach:

- Modelovanie prostredia softvéru
- Výber testovacích scenárov
- Spustenie a vyhodnotenie testovacích scenárov
- Meranie pokroku v testovaní

Modelovanie prostredia softvéru

Úlohou testera je simulovať interakciu medzi softvérom a jeho prostredím. Testeri musia identifikovať a simulovať rozhrania, ktoré softvérový systém využíva, a určiť vstupy, ktoré môžu cez každé rozhranie prechádzať. Toto môže byť jeden z najzákladnejších problémov, ktorým testeri čelia, a môže byť náročný vzhľadom na rôzne formáty súborov, komunikačné protokoly a dostupné rozhrania tretích strán (API – aplikačné programové rozhrania). Štyri najbežnejšie rozhrania sú: [36]

- **Užívateľské rozhrnie** zahŕňa všetky bežné metódy komunikácie medzi užívateľom a softvérom. Najvýraznejším príkladom je grafické užívateľské rozhranie (GUI), no stále sa používajú aj staršie dizajny ako príkazový riadok či menu-ovládanie. Medzi možné mechanizmy vstupu patria kliknutia myšou, stlačenia klávesnice a vstupy z iných zariadení. Tester potom rozhoduje, ako tieto údaje zorganizovať, aby pochopil, ako ich zostaviť do efektívneho testu.

- **Softvérové rozhranie** (API) je spôsob akým softvér využíva operačný systém, databázu alebo `runtime` knižnicu. Služby, ktoré tieto aplikácie poskytujú, sú modelované ako testovacie vstupy. Výzvou pre testerov je otestovať nielen očakávané, ale aj neočakávané služby. Napríklad, všetci vývojári očakávajú, že operačný systém uloží súbory za nich. No často zanedbávajú službu, ktorou operačný systém informuje o plnom úložisku. Aj chybové hlásenia musia byť testované.

- **Súborové rozhranie** vznikne vždy, keď softvér číta alebo zapisuje dáta do externých súborov. Vývojári musia napísať veľa kontrolného kódu na overenie, či súbor obsahuje správne údaje a formátovanie. Preto musia testeri vytvárať alebo generovať súbory s legálnym ale aj nelegálnym obsahom a súbory, ktoré obsahujú rôzny text a formátovanie.

- **Komunikačné rozhranie** umožňuje priamy prístup k fyzickým zariadenia, ktoré vyžadujú komunikačný protokol. Na otestovanie takýchto softvérov, testeri musia byť schopný generovať platné a neplatné protokolové toky. Musia zostaviť a poslať testovanému softvéru množstvo rôznych kombinácií príkazov a údajov v správnom formáte.

Testeri sa môžu tiež stretnúť s podmienkami, ktoré je ťažké vytvoriť v testovacom prostredí. Nastavenie týchto závislostí môže vyžadovať značné množstvo času a energie, čím sa môžu zmariť výhody, ktoré automatizované testy prinášajú. [24]

Aby sa prekonali tieto výzvy, odborníci vyvinuli mechanizmus nazývaný *mockovanie* (napodobňovanie), ktorý nahrádza testovacie závislosti testovanej jednotky (FUT – Function Under Test) vytvorením *mock* objektov. To znamená, že vývojári vytvoria falošný objekt a riadia jeho správanie tak, aby napodobňoval správanie skutočnej závislosti na účely testovania. [37]

Výber testovacích scenárov

Mnohé doménové modely a rozdelenia premenných predstavujú nekonečný počet testovacích scenárov, pričom každý z nich stojí čas a peniaze. Preto sa testeri usilujú o čo najefektívnejšie pokrytie pri testovaní.

- Pokrytie kódu – spustenie každého riadku kódu aspoň raz.
- Pokrytie vstupov – aplikovanie každej externej udalosti.

Toto je minimálne kritérium, podľa ktorého testeri posudzujú úplnosť svojej práce a vyberajú testovaciu sadu, aby splnili svoj gól pokrytia. Ale ak by pokrytie kódu a vstupov bolo skutočne dostačujúce, softvér by sa vydával takmer bez chýb. [36]

Spustenie a vyhodnotenie testovacích scenárov

Keď testeri identifikujú vhodné testy, prevedú ich do spustiteľnej podoby, často ako kód, aby výsledné testovacie scenáre simulovali typické používateľské akcie. Pretože manuálne vykonávanie testovacích scenárov je pracné a náchylné na chyby, snažia sa testeri tieto scenáre čo najviac automatizovať.

V mnohých prostrediach je možné vstupy automaticky aplikovať pomocou kódu, ktorý simuluje používateľov, a existujú nástroje, ktoré s tým pomáhajú. Úplná automatizácia si vyžaduje simuláciu každého vstupného zdroja a výstupného cieľa celého prevádzkového prostredia.

Vyhodnotenie scenárov, druhá časť tejto fázy, sa síce dá jednoducho formulovať, no je náročné ju vykonať. Vyhodnotenie znamená porovnanie skutočného výstupu softvéru, ktorý vznikol vykonaním testovacieho scenára, s očakávaným výstupom, ako ho popisuje špecifikácia. Predpokladá sa, že špecifikácia je správna a ak sa skutočný výstup od nej líši, jedná sa o zlyhanie. [36]

Meranie pokroku v testovaní

Definovať pokrok v testovaní je zložité. V súčasnej praxi sa meranie pokroku v testovaní obmedzuje najmä na počítanie vecí. Pokrok v testovaní je možné vyjadriť napríklad ako: [36]

- Počet aplikovaných vstupov
- Percento pokrytého kódu
- Počet spustení aplikácie
- Počet úspešných ukončení aplikácie
- Počet zistených chýb

Interpretácia týchto počtov je však zložitá, pretože veľký počet chýb sa dá interpretovať ako dobrá správa, ale aj ako zlá správa. Vysoký počet chýb môže znamenať, že testovanie bolo dôkladné a už zostáva len málo chýb na odhalenie. Alebo to môže znamenať, že softvér obsahuje veľa chýb a aj keď už boli mnohé odhalené, ešte mnoho ďalších zostáva. [36]

2.3 Testovacie závislosti

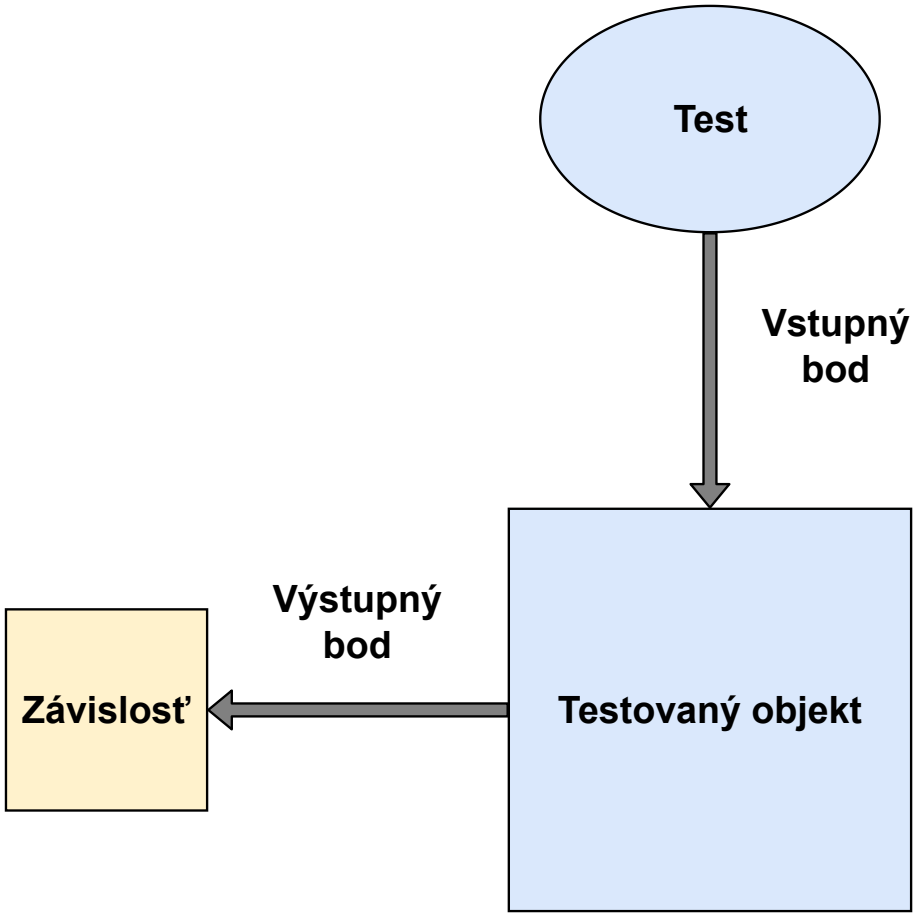
Testovacie závislosti sa delia na:

- Vstupné závislosti
- Výstupné závislosti

Avšak niektoré testovacie závislosti môžu byť súčasne aj vstupné, aj výstupné. V niektorých testoch môžu predstavovať výstupné body a v iných testoch zasa vstupné dáta, ktoré prichádzajú do testovanej aplikácie. [21]

Výstupné závislosti

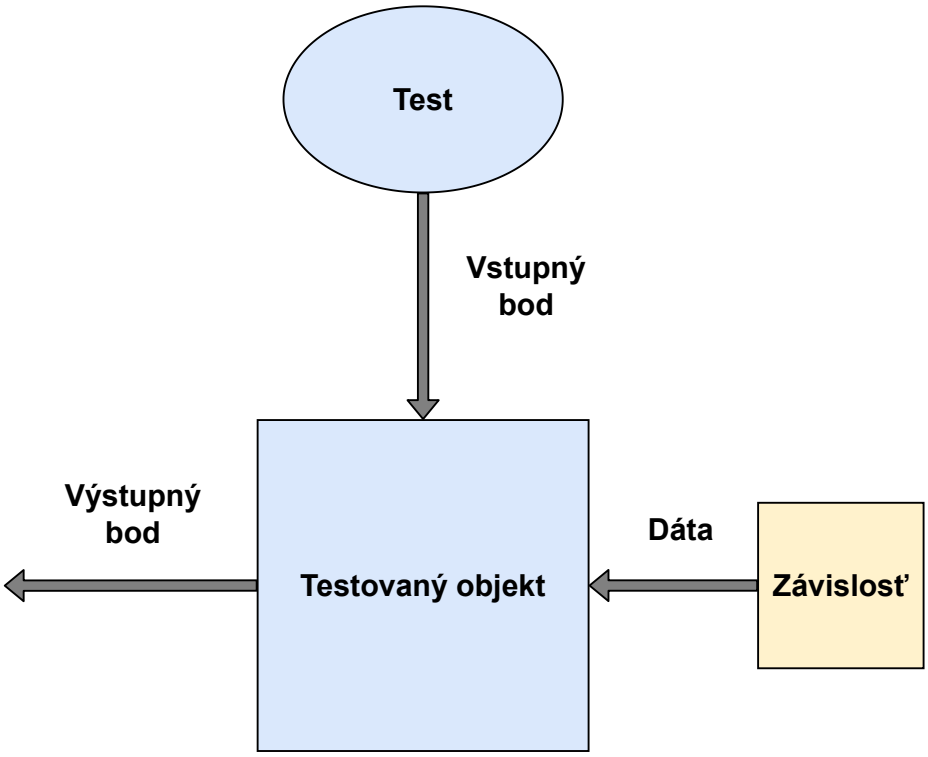
Závislosti, ktoré predstavujú výstupné body testovanej jednotky, ako napríklad ukladanie dát do databázy, odosielanie e-mailov alebo upozorňovanie API. Reprezentujú logický tok dát, ktorý vystupuje z testovanej jednotky. [21] Viď obrázok 2.4.



Obr. 2.4: Výstupné testovacie závislosti. Prevzaté z [21]

Vstupné závislosti

Závislosti, ktoré nepredstavujú výstupné body. Nereprezentujú podmienky na koncové správanie testovanej jednotky, ale naopak reprezentujú podmienky pre správny chod testovanej jednotky alebo vstupné dáta, ako napríklad dáta z databázy, súbory v súborovom systéme alebo sieťové odpovede. Reprezentujú logický tok dát, ktorý vstupuje do testovanej jednotky ako výsledok predošlých operácií. [21] Viď obrázok 2.5.



Obr. 2.5: Vstupné testovacie závislosti. Prevzaté z [21]

2.4 Testovací dvojník

Niekedy je veľmi ťažké otestovať testovaný systém (SUT – System Under Test), pretože závisí od iných komponentov, ktoré nie je možné použiť v testovacom prostredí. Môže to byť z rôznych dôvodov – tieto komponenty nemusia byť dostupné, nemusia vracať očakávané výsledky potrebné pre test, alebo ich spustenie by mohlo mať nežiaduce vedľajšie účinky. V iných prípadoch môže naša testovacia stratégia vyžadovať väčšiu kontrolu alebo viditeľnosť nad vnútorným správaním testovaného systému (SUT). Keď píšeme test, v ktorom nemôžeme použiť skutočnú závislú komponentu, môžeme ju nahradiť testovacím dvojníkom. Táto náhrada nemusí fungovať úplne rovnako ako reálny komponent, stačí, aby poskytovala rovnaké API, aby si SUT „myslel“, že komunikuje so skutočným komponentom. [18]

Typy testovacích dvojníkov

Testovací dvojník je všeobecný pojem pre akýkoľvek prípad, keď nahradíme produkčný objekt na účely testovania. [10] Existuje viacero typov testovacích dvojníkov, ako je možné vidieť v tabuľke 2.1.

Testovací dvojník	Podvrhy (Stubs)	Dummy objects	Odovzdávajú sa medzi metódami, ale v skutočnosti sa nikdy nepoužívajú. Zvyčajne slúžia len na vyplnenie zoznamov parametrov. [10]
		Fake objects	Majú funkčné implementácie, ale často používajú skratky, ktoré nie sú vhodné pre produkčné použitie. [10]
		Test stubs	Poskytujú vopred pripravené odpovede na volania vykonané počas testu a zvyčajne nereagujú na nič mimo toho, čo je pre test naprogramované. [10]
	Napodobeniny (Mocks)	Test spies	Sú rozšírené stubs, ktoré navyše zaznamenávajú informácie o tom, ako boli volané. Jednou z foriem môže byť napríklad e-mailová služba, ktorá zaznamenáva, koľko správ jej bolo odoslaných. [10]
		Mock objects	sú naprogramované s očakávaniami, ktoré predstavujú špecifikáciu volaní, ktoré by mali prijať. Ak dostanú neočakávané volanie, môžu vyhodiť výnimku, a pri overovaní sa kontroluje, či dostali všetky volania, ktoré sa od nich očakávali. [10]

Tabuľka 2.1: Typy testovacích dvojníkov

Termíny napodobeniny a podvrhy sú často nesprávne používané, preto ak si tester nie je istý, ktorý termín je správny, je lepšie použiť termín testovací dvojník, ktorý zahŕňa obe pomenovania. [21]

Podvrhy (Stubs)

Podvrhy riešia problém vstupných závislostí. Sú to falošné moduly, objekty alebo funkcie, ktoré poskytujú falošné správanie alebo dáta testovanej jednotke. [21]

Napodobeniny (Mocks)

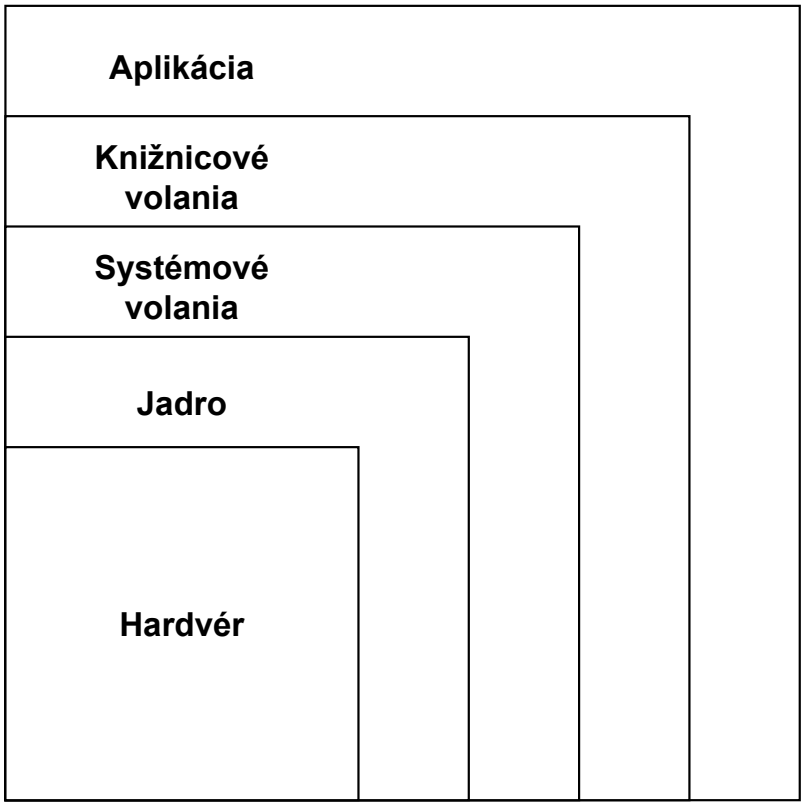
Napodobeniny riešia problém výstupných závislostí. Sú to takisto falošné moduly, objekty alebo funkcie, podobne ako podvrhy (stubs). Avšak namiesto predávania dát, reprezentujú výstupné body testovanej jednotky. [21]

2.5 Systémové volania a knižnicové volania

Systémové volania, tiež nazývané volania jadra, sú základnou a najpoužívanejšou metódou komunikácie medzi aplikačnými procesmi a jadrom. [13]

Väčšinou však procesy nekomunikujú s jadrom priamo pomocou systémových volaní, ale používajú obaľovacie funkcie knižníc, takzvané knižnicové volania. Ako je to možné vidieť na hierarchii komunikácie na obrázku 2.6. Knižnicové volania zväčša nesú rovnaký názov ako systémové volania, ale nie je to pravidlo. Obaľovacie funkcie sú často jednoduché funkcie, ktoré len kopírujú argumenty do správnych registrov predtým, ako zavolajú skutočné systémové volanie a následne nastavia vhodné chybové hlásenie po návrate systémového volania. [33]

Ak neexistuje knižnicové volanie pre dané systémové volanie, je možné použiť obaľovaciu funkciu `syscall()` so správnymi argumentami v podobe špecifického čísla daného systémového volania a špecifických argumentov, ktoré dané systémové volanie potrebuje. [32]



Obr. 2.6: Hierarchia komunikácie v operačných systémoch

Princíp a fungovanie systémových volaní

Táto časť čerpá z knihy [13].

Užívateľský proces beží vo svojom vlastnom adresnom priestore a má presne vymedzené rozhranie. K jadru nemá bežne prístup. Avšak často nastáva situácia, kedy proces od jadra potrebuje nejakú službu. Vtedy proces využije systémové volanie, ktoré mu umožní komu-

nikovať s jadrom. Systémové volanie predá riadenie jadru a následne po vykonaní operácie sa systém prepne naspäť a procesu je predaný výsledok.

Predanie riadenia jadru sa líši od architektúry procesoru. V zásade sa ale využívajú dve metódy:

- Prerušenie
- Špeciálna inštrukcia

Metóda prerušenia

Prerušenie je staršia a pôvodná metóda v systémoch GNU/Linux. Má široké využitie, avšak je pomalšia. Metóda funguje na základe softvérového prerušenia, ktoré spôsobí proces zavolaním systémového volania. Prerušenie sa začne obsluhovať a procesor sa prepne do režimu jadra a začne sa vykonávať kód jadra.

Pred obsluhou volania sa však ešte musia vykonať viaceré operácie. Medzi najdôležitejšie patrí uloženie stavu registrov a kontrola platnosti čísla volania.

Po skončení obsluhy sa obnoví pôvodný stav procesora, prepne sa späť do užívateľského režimu a riadenie sa vráti procesu. Výsledok uložený v registri sa uloží do pamäte a obnoví sa pôvodný stav registrov. [13]

Metóda špeciálnej inštrukcie

Metóda špeciálnej inštrukcie je novšia, nie veľmi rozdielna od metódy prerušenia. Rozdiel je iba v tom, že nepoužíva prerušenie, ale inštrukcie. Prepnutie do režimu jadra zabezpečuje špeciálna inštrukcia `sysenter`, ktorá zároveň aj nastaví registre. Na konci obsluhy volania sa vykoná špeciálna inštrukcia `sysexit`, ktorá obnoví registre a prepne stav procesora späť do užívateľského režimu. [13]

2.6 Prednačítanie pri dynamickom načítaní knižníc

`LD_PRELOAD` je premenná prostredia v Linuxe, ktorá umožňuje načítať určené zdieľané knižnice (objekty) pred všetkými ostatnými pri spustení aplikácie. [3] Táto funkcionalita je podporovaná dynamickým linkerom na väčšine UNIX/Linux systémoch. Pomocou prednačítania môžeme ovplyvniť prepojenie zdieľaných knižníc a rozlíšenie symbolov/funkcií počas **behu programu (runtime)**. Tento mechanizmus je možné využiť na prepísanie určitých funkcií vlastnými implementáciami alebo na injektovanie dodatočnej funkcionality do procesu počas jeho behu. [15]

Ak je spustiteľný súbor dynamicky linkovaný, je možné načítať knižnicu, ktorá prepíše akékoľvek funkcie alebo symboly prednačítané z iných knižníc. To je obzvlášť užitočné pri odchytávaní funkcií, teda pri pozorovaní alebo úprave správania aplikácií tretích strán, ku ktorým je nemožný prístup k zdrojovému kódu. Napríklad, ak je potrebné prepísať alebo rozšíriť funkciu malloc zo štandardnej knižnice libc, je možné pomocou `LD_PRELOAD` načítať vlastnú knižnicu pred libc a tak zachytiť volania malloc, čím je možné ju efektívne prepísať. Táto funkcionalita sa často používa na **ladenie (debugging)** alebo na prepísanie funkcií systémových knižníc bez potreby meniť samotný binárny súbor alebo zdrojový kód. [15]

Zdieľané knižnice

Moderné linuxové systémy používajú zdieľané knižnice (dynamické knižnice), ktoré sú pripojené k programu až počas behu programu, na rozdiel od statických knižníc, ktoré sú pripojované už v čase prekladu. Zdieľané knižnice sú uložené v súboroch s príponou `.so` **(shared object)**, ako napríklad štandardná knižnica programovacieho jazyka C `glibc.so`. Väčšinou sa tieto knižnice nachádzajú v adresári `/usr/lib`. [27]

Protipólom dynamických knižníc sú už spomenuté statické knižnice. Programy so statickými knižnicami sú pri prvom spustení rýchlejšie ako tie s dynamickými knižnicami, avšak veľkosť daných programov je viditeľne väčšia, ako keby používali zdieľané knižnice. [27]

Úspora pamäte je hlavným dôvodom, prečo sa v dnešnej dobe v systémoch využívajú prevažne zdieľané knižnice. Ďalším dôležitým dôvodom je údržba systému. Pri zmene dynamickej knižnice je potrebné preložiť len samotnú knižnicu, zatiaľ čo v prípade statickej knižnice je nutné preložiť aj knižnicu, aj všetky programy, ktoré danú knižnicu používajú. [27]

2.7 Zachytávanie knižnicových volaní

Rozlišujeme dva spôsoby obalenia, založené na tom, kedy sa zachytávanie nastaví:

- počas pripojenia (link time)
- počas behu (runtime)

Tieto dva prístupy sa líšia aj v typoch funkcií, ktoré je možné zachytiť.

Zachytávanie volaní počas pripojenia

Prvý prístup je založený na použití možnosti `--wrap` v GNU linkeri. Napríklad, ak chceme obaliť funkciu foo, musíme implementovať zodpovedajúcu obalovú funkciu `__wrap_foo`. Pôvodná funkcia je potom dostupná cez symbol `__real_foo`. Ak pri pripojovaní špecifikujeme `--wrap` foo, GNU linker zabezpečí správne pripojenie týchto symbolov. Tento prístup je však obmedzený na prípady, kedy môžeme upravovať krok pripojenia aplikácie, pretože požadované symboly musia byť špecifikované v čase pripojenia. Volania funkcií zo zdieľaných knižníc nie je možné takto obaliť, pretože tieto symboly GNU linker rieši až počas behu aplikácie. [4]

Zachytávanie volaní počas behu

Na začiatku spustenia aplikácie dynamický linker načíta a pripojí všetky závislé zdieľané knižnice. Druhý prístup mení poradie, v akom linker načítava tieto knižnice. Aby sme obalili funkciu, poskytneme náhradnú funkciu s rovnakým symbolickým názvom, aký má pôvodná funkcia. Linker potom musí túto náhradnú funkciu pripojiť pred pôvodnou knižnicou. [4]

Jedným zo spôsobov, ako to dosiahnuť, je úprava pripojovacieho kroku tak, aby knižnica s obalovými funkciami bola uvedená pred pôvodnou knižnicou. Alternatívne môžeme nastaviť premennú prostredia `LD_PRELOAD`, ktorá pred spustením aplikácie ukazuje na knižnicu s obalovými funkciami. Táto druhá metóda má výhodu v tom, že nie je potrebné zasahovať do pripojovania aplikácie. [4]

Keď sa funkcia zavolá, obalová funkcia načíta pôvodnú knižnicu pomocou `dlopen`, vyhľadá adresu pôvodného symbolu pomocou `dlsym` a následne presmeruje volanie na pôvodnú funkciu. Pri úprave pripojovacieho kroku môže tento prístup zachytiť všetky volania, ktoré

dokáže zachytiť **obalovanie pripojovanie počas pripojovania**, a navyše aj volania pochádzajúce zo zdieľaných knižníc. Naopak, verzia založená na `LD_PRELOAD` dokáže zachytiť iba volania do zdieľaných knižníc, nie do staticky pripojených. Oba mechanizmy vyžadujú obalové funkcie, ktoré sa tvária ako pôvodné funkcie. Pri každom volaní obalová funkcia informuje monitor výkonu pred aj po presmerovaní volania na pôvodnú funkciu. [4]

2.8 Súbory v systémoch Linux

Väčšina užívateľov linuxových alebo unixových systémov sa stretla s vetou "Všetko je súbor". [8] No nie je to tak úplne jednoznačné [14], pretože v linuxových alebo unixových systémoch sa súbory delia na viacero druhov: [7]

- Bežné súbory (Regular files)
- Adresárové súbory (Directory files)
- Blokové alebo znakové špeciálne súbory (Block or character special files)
- Odkazové súbory (Link files)
- Soketové súbory (Socket files)
- Súbory pomenovaných rúr (Named pipe files)

Každý z týchto druhov má spoločné, ale aj rozdielne vlastnosti. Existujú špeciálne súbory, ktoré by sa dali definovať ako niečo viac ako len súbor. [14] Všeobecne je možné definovať súbor ako základnú jednotku úložiska, ktorá obsahuje dáta a informácie. [25]

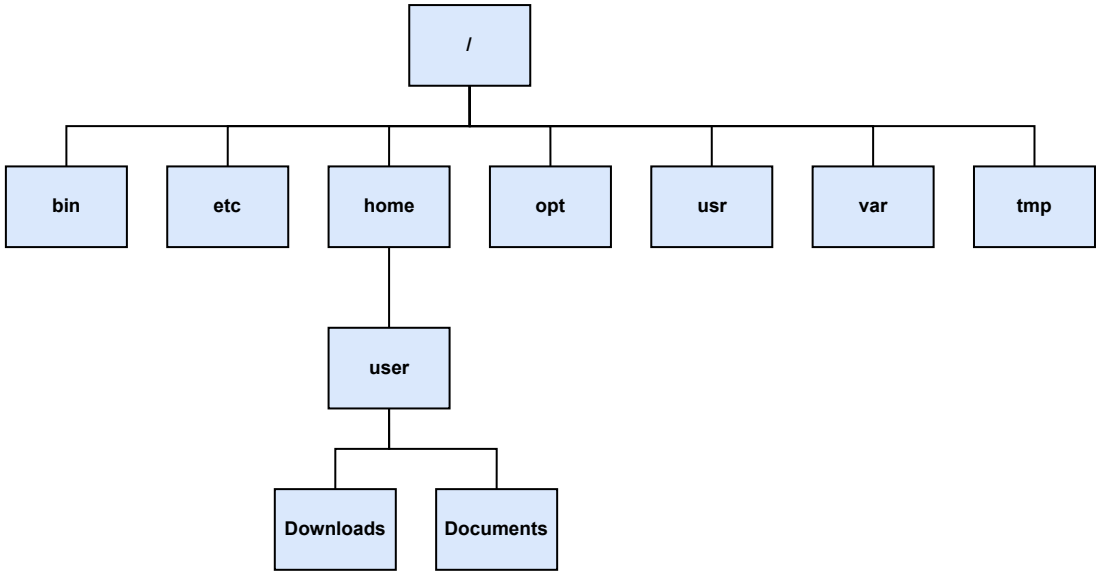
Bežné súbory (Regular files)

Súbory, ktoré obsahujú dáta s rôznym obsahom, ako napríklad text, audio, video alebo obrázky. [7] Väčšinou sú používané priamo užívateľom. [12] Môžu obsahovať rôzne prípony, ako napríklad `.txt` pre textový súbor alebo `.mp4` pre video. [7]

Adresárové súbory (Directory files)

Linuxový súborový systém používa adresáre na hierarchické usporiadanie súborov. Adresár je taktiež súbor, ale namiesto dát, ako pri bežných súboroch, obsahuje názvy a lokácie ostatných súborov v systéme. [7]

Linuxový súborový systém začína adresárom `root` alebo `/`, ktorý obsahuje názvy a lokácie ostatných súborov [7], ako je to možné vidieť na obrázku 2.7. Každý ďalší adresár, okrem `root` adresára, leží pod nejakým iným adresárom. Adresár, ktorý obsahuje aktuálny adresár, sa nazýva nadradený adresár. Ak sa v aktuálnom adresári nachádza ďalší adresár, nazýva sa podadresár. [35]



Obr. 2.7: Adresárový strom

Blokové alebo znakové špeciálne súbory (Special files)

Blokové a znakové špeciálne súbory sú podkategórie súborov zariadení, ktoré slúžia na komunikáciu medzi systémom a pripojenými zariadeniami, ako napríklad tlačiarne, externé pamäte alebo pevné disky. Súbory zariadení sú uložené v adresári `/dev` a v jeho podadresároch. Každý súbor zariadenia reprezentuje určité zariadenie v systéme. Čítanie a zápis z týchto súborov je v skutočnosti čítanie a zápis do reálneho fyzického zariadenia. [28] Znakové špeciálne súbory reprezentujú zariadenia, ktoré pracujú s dátami v bajtoch, ako napríklad tlačiarne alebo monitory. Blokové špeciálne súbory reprezentujú zariadenia, ktoré pracujú s dátami v blokoch, ako napríklad pevné disky. [7]

Odkazové súbory (Link files)

Odkazové súbory sú ukazovatele na iné súbory. Dovoľujú užívateľovi pracovať so súbormi, ktoré sa nachádzajú na rozličnej lokácii v systéme alebo pod iným názvom. Delia sa na dva typy: [7]

- **Pevné odkazy (Hard links)**, ktoré vytvoria zrkadlovú kópiu pôvodného súboru. Tento typ odkazu je možné použiť výhradne na súbory z toho istého súborového systému.
- **Jemné alebo symbolické odkazy (Soft or symbolic links)**, ktoré vytvoria ukazovateľ na pôvodný súbor. S použitím jemných odkazov je možné odkazovať aj na súbory z iných súborových systémov.

Soketové súbory (Socket files)

Soketové súbory slúžia na komunikáciu medzi aplikáciami alebo asynchrónnymi procesmi, ktoré nie sú potomkami rovnakého predchodcu. [28] Sú to komunikačné koncové body, ktoré sa používajú na výmenu dát. Aplikácie, ktoré poskytujú služby iným aplikáciám alebo vzdialeným klientom, využívajú sokety na nadviazanie spojenia. Pri spojení so vzdialeným klientom sa na potvrdenie spojenia využívajú IP adresy a čísla portu. Avšak pri lokálnej komunikácii, aby sa predišlo komplikovaným procesom, sa namiesto IP adresy a čísla portu používa názov súboru. [7]

Súbory pomenovaných rúr (Named pipe files)

Rúry slúžia na odoslanie výstupu z jedného procesu ako vstup do druhého procesu. [7] Rúry majú dva konce. Do jedného konca proces zapisuje dáta a z druhého konca proces dáta číta. [2] Existujú dva typy rúr: [7]

- Štandardné rúry
- Pomenované rúry

Pomenovaná rúra je komunikačný objekt, ktorý má svoje miesto v súborovom systéme. Nachádza sa vo svojom adresári aj v dobe, kedy ju žiaden proces nevyužíva. Keďže je to súbor, je možné ju dokonca uložiť do archívu a následne obnoviť. [13]

Kapitola 3

Návrh testovacieho dvojníka

Táto kapitola sa zaoberá návrhom a výberom vhodného testovacieho dvojníka pre testovacie účely softvéru **sos** od spoločnosti Red Hat. Taktiež budú v tejto kapitole predstavené už existujúce riešenia problémov, ktoré si vyžadujú prítomnosť testovacích dvojníkov.

Softvér sos

Sos alebo konkrétne **sos report** je nástroj, ktorý zbiera konfiguračné detaily, systémové informácie a diagnostické informácie z Red Hat Enterprise Linux systémov. Ako sú napríklad systémové a konfiguračné súbory, verzia jadra a načítané moduly. Zozbierané informácie uloží do archívu, ktorý je následne možné nahrať na portál pre užívateľskú podporu. [23]

Sos je program písaný v jazyku Python, ktorý patrí medzi interpretované jazyky. To znamená, že kód programu je spracovaný interpretom za behu pri spustení programu.

3.1 Analýza požiadaviek

Testovací dvojník by mal spĺňať aspoň základné požiadavky ako práca so súbormi, použiteľnosť pri automatizovanom a manuálnom testovaní.

Práca so špeciálnymi a bežnými súbormi

Ako už bolo spomenuté, základom **sos** je práca so súbormi. Pre správny chod je nutné otestovať, že boli dané súbory zozbierané a že súbory, ktoré sa v archíve nemajú nachádzať, sa tam skutočne nenachádzajú. Nie vždy je však tester schopný nastaviť prostredie tak, aby boli dodržané všetky závislosti na správny chod softvéru.

Preto je kľúčový bod návrhu testovacieho dvojníka práca so špeciálnymi a bežnými súbormi. Testovací dvojník by mal byť schopný nahradiť existujúci, ale aj neexistujúci súbor bez toho, aby zasahoval do existujúcich adresárov daných súborov.

Automatizované testovanie

Ďalším bodom návrhu je schopnosť používať testovacieho dvojníka aj počas automatizovaného testovania. Testovací dvojník by mal byť ľahko nastaviteľný a použiteľný v automatizovaných testoch.

Automatizované testovanie je použitie špecializovaných softvérových nástrojov na vykonanie testovania a následné porovnanie získaných výsledkov s očakávanými výsledkami. [16]

Najväčšími výhodami automatizácie sú: [16]

- Úspora času a financií: Automatizácia testov dovoľuje otestovať softvér rýchlejšie ako pri použití manuálnych testov. Vďaka čomu šetrí čas aj náklady na testovanie.
- Vylepšenie testovacieho pokrytia: Nástroje na automatizáciu umožňujú spustiť veľké množstvo testov a taktiež opakované spustenie testov, čo môže zlepšiť celkové pokrytie.
- Zvýšenie presnosti testov: Nástroje na automatizáciu umožňujú testovanie s väčšou presnosťou ako ľudia, čo môže viesť k zredukovaniu chýb.
- Znovupoužiteľnosť: Automatizované testy môžu byť použité na rôzne verzie testovaného softvéru, čo šetrí čas a úsilie.
- Zvýšenie efektivity: Automatizácia umožňuje nepretržitý beh testov, čo zvyšuje efektivitu daného testovania.

Manuálne testovanie

Časť o manuálnom testovaní čerpá z [6].

Testovací dvojník by mal byť takisto použiteľný počas manuálneho testovania, pretože v praxi môže nastať prípad, kedy opravenie nejakej chyby nie je možné otestovať pomocou automatizovanej testovacej sady.

Na najzákladnejšej úrovni je manuálne testovanie proces testovania softvéru, pri ktorom sa testovacie prípady, teda konkrétne overenia funkcií, vlastností alebo výkonu, vykonávajú bez použitia akýchkoľvek automatizovaných nástrojov. Akákoľvek odchýlka od očakávaného správania alebo výstupu sa môže považovať za defekt.

Manuálni testeri môžu byť buď QA špecialisti, alebo zástupcovia potenciálnych koncových používateľov. Títo testeri pracujú s produktom z pohľadu používateľa. Buď na základe pôvodných návrhových požiadaviek, alebo podľa zavedených osvedčených postupov či štandardov. Funkcie, ktoré nezodpovedajú pôvodným požiadavkám alebo štandardom, sa zaznamenávajú ako defekty.

Najväčšími výhodami manuálneho testovania sú:

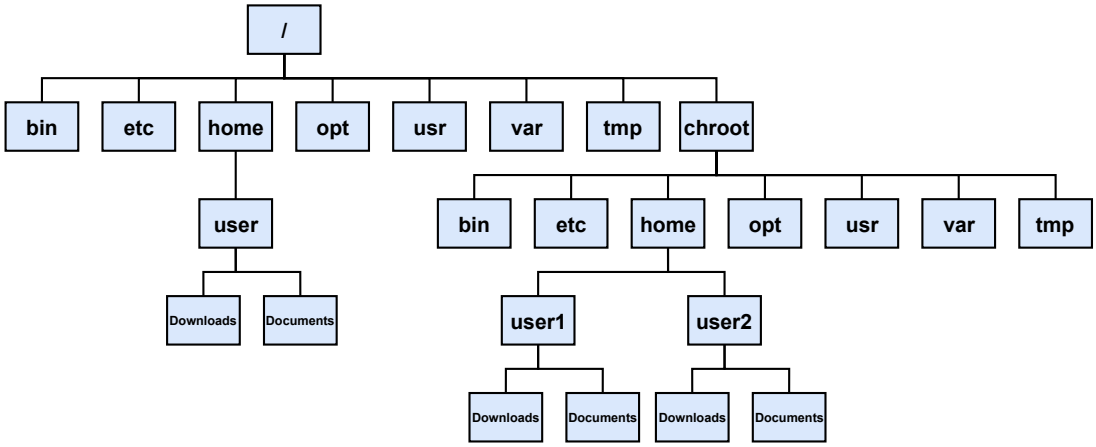
- **Komplexnosť** – Na rozdiel od automatizovaného testovania, manuálne testovanie môže pokrývať širšiu škálu oblastí, vrátane subjektívnych aspektov, ako je napríklad používateľská prívetivosť.
- **Prispôsobivosť** – Testeri môžu počas manuálneho testovania identifikovať potenciálne problémy, ktoré nie sú zdokumentované v testovacích prípadoch.
- **Dostupnosť** – Testeri nemusia ovládať programovanie alebo princípy automatizovaného testovania, aby vedeli testy nastaviť a vykonať.
- **Prevencia** – Vykonávanie testov v skorých fázach vývoja pomáha predchádzať drahým opravám alebo vydaniam s dodatočnými opravami.

3.2 Existujúce riešenia

V tejto sekcii je predstavených niekoľko existujúcich riešení, ktoré by mohli byť použité na vyriešenie daného problému.

Chroot

Názov `chroot` je skratka z **change root**, čo sa dá preložiť ako "zmena koreňového adresára". [22] Je to operačný príkaz v systémoch GNU/Linux, ktorý, ako už naznačuje názov tohto príkazu, umožňuje zmeniť koreňový adresár. Táto technika sa využíva na vytvorenie izolovaného prostredia pre aplikácie, aby sa zabránilo prípadnému negatívnemu ovplyvneniu ostatných častí systému. Zároveň je však možné tento mechanizmus využiť aj na vytvorenie vhodného prostredia pre testovacie účely danej aplikácie, vytvorením vlastného koreňového adresára so správnymi podmienkami na otestovanie aplikácie. [19] Viď obrázok 3.1.



Obr. 3.1: Nástroj chroot

Pre vytvorenie a správu `chroot` prostredia je nutné, aby mal užívateľ administrátorské práva. Prvým krokom je vytvorenie nového adresára, ktorý bude slúžiť ako nový koreňový adresár. Následne je možné do nového adresára pridať všetky potrebné súbory pre správne nastavenie prostredia. Avšak je nutné dodržať pôvodnú adresárovú štruktúru. Ďalším krokom je inštalácia všetkých potrebných knižníc pre správny chod aplikácií. Je potrebné nainštalovať minimálne základné systémové knižnice. Posledným krokom je zmena koreňového adresára na nový vlastný adresár a to pomocou príkazu `chroot`. [19]

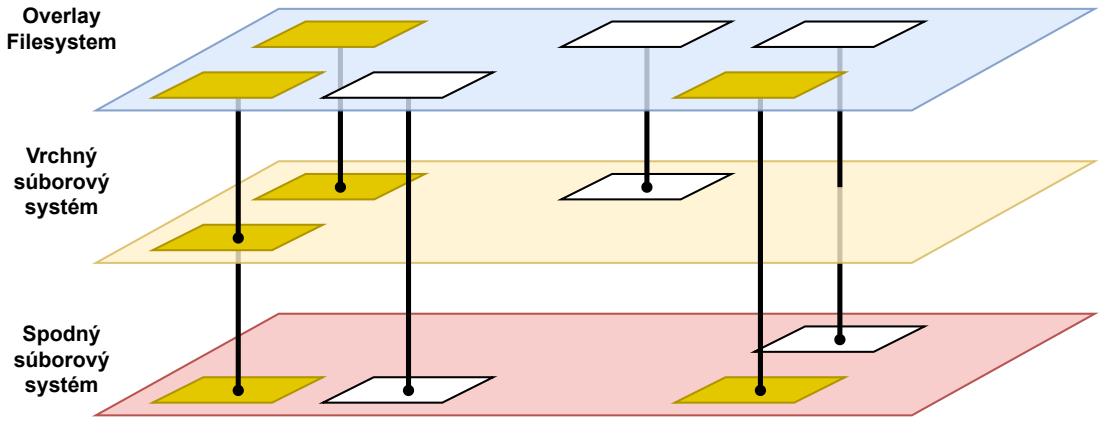
Overlay Filesystem

Časť o Overlay Filesystem čerpá z manuálu [5].

Overlay Filesystem je súborový systém, ktorý sa nachádza nad pôvodným súborovým systémom a prekrýva ho. **Overlay Filesystem** spája dva súborové systémy, takzvaný "vrchný" a "spodný" súborový systém. Pokiaľ objekt s rovnakým názvom existuje v oboch systémoch, objekt vo "vrchnom" súborovom systéme je viditeľný, zatiaľ čo objekt v "spodnom" súborovom systéme je skrytý. Pokiaľ sú objektami adresáre, tak sú zlúčené dokopy vo "vrchnom" súborovom systéme. Viď obrázok 3.2.

Vrchný a spodný súborový systém môže byť taktiež nazývaný ako vrchný a spodný adresárový strom, pretože oba adresárové stromy môžu byť z rovnakého súborového systému a zároveň nie je podmienkou, že koreň súborového systému musí byť uvedený ako vrchný alebo spodný súborový systém.

Väčšina súborových systémov podporovaných systémom GNU/Linux môže slúžiť ako spodný súborový systém, ale nie všetky súborové systémy majú funkcie potrebné na to, aby **Overlay Filesystem** fungoval. Vrchný súborový systém môže mať možnosť zápisu, ale v tom prípade musí podporovať tvorbu dôveryhodných alebo užívateľských rozšírených atribútov a musí byť schopný poskytnúť platný `d_type`[1], zatiaľ čo spodnému súborovému systému stačí iba možnosť čítania. Pokiaľ sú oba súborové systémy iba s možnosťou čítania, je možné použiť ľubovoľný súborový systém. [5]



Obr. 3.2: Overlay Filesystem

Adresáre

Prekrývanie sa týka hlavne adresárov. Ako už bolo spomenuté, pokiaľ sa názvy adresárov v oboch súborových systémoch zhodujú, sú zlúčené do jedného. Následne, ak je požadované vyhľadávanie v zlúčenom adresári, vyhľadávanie sa prevedie v oboch adresároch a výsledok je uložený do `cache` pamäte **Overlay Filesystem**. Ak obe prevedené vyhľadávania nájdu adresáre v oboch súborových systémoch, oba sú uložené a je vytvorený zlúčený adresár. Ak je nájdený iba jeden adresár, tak je uložený iba ten. [5]

[1] Jedna z návratových hodnôt funkcie `readdir`, ktorá uvádza typ súboru

Tvorenie a mazanie adresárov

Aby bolo možné mazať adresáre a vytvárať nové adresáre bez toho, aby bol zmenený spodný súborový systém, **Overlay Filesystem** si zaznamenať v hornom súborovom systéme, že vznikli nové súbory (adresáre) alebo, že nejaké boli odstránené. To je možné vďaka `whiteout` a nepriehľadným súborom.

`Whiteout` súbor je vytvorený ako znakové zariadenie s číslom zariadenia 0/0 alebo ako bežný súbor s nulovou veľkosťou a s `xattr`[2] "trusted.overlay.whiteout". Pokiaľ je `whiteout` súbor nájdený v zlúčenom adresári horného súborového systému, všetky súbory so zhodným názvom z dolnej vrstvy sú ignorované a `whiteout` súbor je taktiež skrytý.

Adresár sa stane nepriehľadným nastavením `xattr`[2] "trusted.overlay.opaque" na hodnotu "y". Následne horný súborový systém obsahuje nepriehľadný adresár a adresár s rovnakým názvom zo spodného súborového systému je ignorovaný.

Nepriehľadný adresár by potom nemal obsahovať žiadne `whiteout` súbory, pretože nemajú žiaden význam. Ak zlúčený adresár obsahuje bežné súbory s `xattr`[2] "trusted.overlay.whiteout", mali by byť tieto súbory dodatočne označené nastavením `xattr`[2] "trusted.overlay.opaque" na hodnotu "x". Tento proces zaručí zbytočnému prechádzaniu všetkých záznamov. [5]

Systémové volanie readdir

Ak je zavolané systémové volanie `readdir` na zlúčený súbor, číta z adresárov z oboch vrstiev. Zoznam názvov súborov je potom takisto zlúčený. Najskôr číta z hornej vrstvy, potom zo spodnej. Už existujúce záznamy nie sú znova pridané na zoznam. Zoznam názvov súborov je uložený v `struct file`[3] a zostávajú uložené, dokým je súbor otvorený. Ak dva procesy súčasne čítajú z jedného adresára, oba budú mať vlastný zoznam vo vlastnej štruktúre. Ak je zavolané systémové volanie `readdir` na adresár, ktorý nie je zlúčený, jednoducho číta z adresára buď zo spodného alebo vrchného súborového systému. [5]

Súbory, ktoré nie sú adresáre

Súbory, ktoré nie sú adresáre, ako napríklad bežné súbory alebo rôzne špeciálne súbory, sú predávané buď z vrchnej alebo spodnej vrstvy. Ak je potrebný zápis do súboru zo spodného súborového systému, súbor je najprv nutné skopírovať do vrchného súborového systému, tento proces sa nazýva `copy_up`. Taktiež pri tvorbe pevného odkazu je nutné vykonať `copy_up`. Naopak pri tvorbe jemného odkazu tento proces nie je nutné vykonávať.

Proces `copy_up` sa najprv ubezpečí, že adresár, v ktorom sa súbor nachádza, existuje vo vrchnom súborovom systéme. Ak nie, vytvorí ho a taktiež všetky jeho rodičovské adresáre. Následne vytvorí objekt s rovnakými metadátami ako má objekt v spodnom súborovom systéme. Ak je objektom súbor, skopíruje všetky dáta zo súboru zo spodnej vrstvy. Nakoniec skopíruje všetky rozšírené atribúty. Po dokončení tohto procesu, **Overlay Filesystem** poskytne priamy prístup do novovytvoreného súboru vo vrchnom súborovom systéme. [5]

[2] xattr – rozšírený atribút
[3] struct file – štruktúra, v ktorej sú uložené informácie o súbore

Knižnica mock pre testovanie v Python

Knižnica `unittest.mock` umožňuje nahradiť časti testovaného systému za falošné objekty a overiť, ako boli použité. Knižnica poskytuje triedu `Mock`, čím odstraňuje potrebu vytvárania veľkého množstva podvrhov (stubs) v testovacej sade. Po ukončení akcie umožňuje overiť, aké metódy boli použité a s akými argumentmi boli volané. Taktiež umožňuje špecifikovať návratové hodnoty funkcií a ich atribúty. [9]

`Mock` knižnica je založená na princípe "akcia –> overenie". [9] Čo znamená, že najprv sa vykoná testovaná metóda, a následne sa overí, či výsledok zodpovedá očakávaniu. [11]

Nahradenie knižnicových volaní

Operačný systém GNU/Linux dovoľuje nahradiť knižnicové volania pomocou mechanizmu prednačítania pri dynamickom načítaní knižníc. Vďaka čomu je možné zmeniť správanie programu bez toho, aby musel byť upravovaný zdrojový kód. Ak program potrebuje vykonať určitú operáciu, zavolá funkciu zo zdieľanej knižnice, ktorá následne vyšle systémové volanie do jadra. Prepísanie týchto volaní dovoľuje zmeniť správanie testovaného programu v rôznych smeroch. Väčšina spustiteľných súborov nie je distribuovaná ako statické spustiteľné súbory, ale ako dynamické. To znamená, že namiesto toho, aby boli knižnice súčasťou spustiteľného súboru, je iba zaznamenaná v spustiteľnom súbore závislosť na danej knižnici. Knižnica je následne načítaná počas behu programu pomocou dynamického linkera. [34]

Vďaka týmto skutočnostiam je možné vytvoriť vlastnú knižnicu s vlastnými obaľovacími funkciami pre systémové volania. Následne ju načítať do testovaného programu a tak nahradiť existujúce knižnicové volania za vlastné a upraviť správanie testovaného programu.

3.3 Výhody a nevýhody existujúcich riešení

Hlavnou výhodou operačného príkazu `chroot` je možnosť vytvoriť nový koreňový adresár, ktorý umožňuje nastaviť vhodné prostredie pre testovanie daného programu. Dovoľuje vytvoriť všetky potrebné súbory, ktoré testovaný program vyžaduje pre správny chod, bez toho, aby to ovplyvnilo pôvodný koreňový adresár. Avšak najväčšou nevýhodou tejto metódy je zložitá konfigurácia a vytvorenie prostredia pri automatizovanom testovaní. Testovacia sada pre testovanie nástroja `sos` obsahuje množstvo testov, takže je náročné a neefektívne vytvárať nové koreňové adresáre pre otestovanie správnej funkcie nástroja.

Riešenie problému pomocou **Overlay Filesystem** taktiež ponúka možnosť izolovaného prostredia pre správne otestovanie programu a to pomocou vytvorenia novej vrstvy súborového systému. Avšak jeho nevýhoda, tak ako pri `chroot`, je zložitá konfigurácia počas automatizovaného testovania. Navyše **Overlay Filesystem** nie je možné použiť na všetkých typoch súborových systémov.

Už existujúca knižnica `mock` pre testovanie v programovacom jazyku Python by bola taktiež dobrá voľba. Avšak pre nástroj `sos` existujú dve testovacie sady, jedna je v programovacom jazyku Python a druhá je v programovacom jazyku Bash. Takže toto riešenie by nebolo vyhovujúce pre obe testovacie sady. Niektoré opravy softvérových chýb sú testované manuálne, čo taktiež neumožňuje použitie tejto knižnice.

Posledná metóda z vybraných existujúcich riešení je nahradenie knižnicových volaní pomocou zdieľanej knižnice a mechanizmu prednačítania pri dynamickom načítaní knižníc. Tento prístup k problému má mnoho výhod. Vlastná implementácia dovoľuje upraviť správanie testovaného programu podľa potreby. Jednoduchá konfigurácia počas automati-

zovaného testovania je jednou z dôležitých výhod. Knižnicu je tiež možné použiť pri manuálnom testovaní. Keďže zdieľaná knižnica nahrádza knižnicové volania, nemá vplyv na to, aký programovací jazyk je použitý v testovacej sade. Nevýhodou tohto riešenia je nutnosť naimplementovať knižnicu vlastnoručne, čo vyžaduje znalosť systémových a knižnicových volaní.

Riešenie problému	Výhody	Nevýhody
Chroot	Vytvorenie nového koreňového adresára so všetkými potrebnými súbormi	Zložitá konfigurácia pri automatizovanom testovaní
Overlay Filesystem	Vytvorenie nového súborového systému so všetkými potrebnými súbormi	Zložitá konfigurácia pri automatizovanom testovaní. Nie je vhodné pre všetky typy súborových systémov
Knižnica mock pre testovanie v Python	Už existujúca implementácia, nie je nutné implementovať knižnicu	Použiteľné iba na testovanie v Python
Nahradenie knižnicových volaní	Jednoduchá konfigurácia pri automatizovanom testovaní. Použiteľné aj v Python aj v Bash. Vlastná implementácia dovoľuje upraviť správanie testovaného programu podla potreby	Nutná znalosť systémových a knižnicových volaní. Knižnicu je potrebné naimplementovať

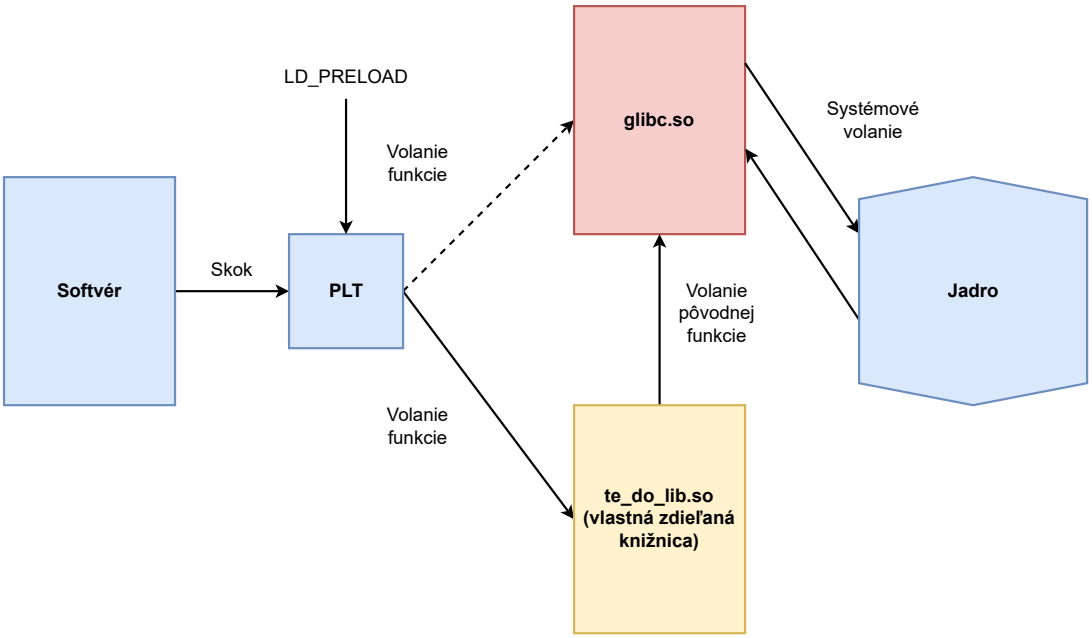
Tabuľka 3.1: Výhody a nevýhody existujúcich riešení

Na základe vykonanej analýzy a porovnania výhod a nevýhod jednotlivých prístupov bolo ako najvhodnejšie riešenie zvolené nahrádzanie štandardných knižnicových volaní pomocou mechanizmu prednačítania pri dynamickom načítaní knižníc.

3.4 Zvolené riešenie problému

Návrh spočíva vo vytvorení zdieľanej knižnice, ktorá bude obsahovať obaľovacie funkcie pre systémové volania s rovnakým názvom ako majú obaľovacie funkcie v **GNU C Library**, čiže štandardnej knižnice programovacieho jazyka C. Pomocou mechanizmu prednačítania pri dynamickom načítaní knižníc bude táto knižnica načítaná v testovanom programe pred štandardnou knižnicou **GNU C Library**. To zaistí, že program bude používať funkcie z tejto zdieľanej knižnice. Viď obrázok 3.3.

Keďže má tento testovací dvojník slúžiť na prácu so súbormi, obaľovacie funkcie boli navrhnuté tak, aby nahradili daný existujúci alebo neexistujúci súbor za súbor, ktorý bol vytvorený testerom pre tento účel. Vytvorený súbor sa môže nachádzať v rovnakom adresári ako daný test. Cesta k tomuto súboru bude zdieľanej knižnici predaná pomocou premennej prostredia. Pomocou ďalšej premennej prostredia bude knižnici predaná informácia, ktorý existujúci alebo neexistujúci súbor má nahradiť vytvoreným súborom.



Obr. 3.3: Návrh testovacieho dvojníka pomocou zdieľanej knižnice a `LD_PRELOAD`

3.5 Zhrnutie kapitoly

V tejto kapitole boli definované a opísané základné požiadavky, ktoré by mal spĺňať navrhovaný testovací dvojník určený na testovanie práce so súborovým systémom. Medzi hlavné požiadavky patrila schopnosť pracovať nielen s bežnými súbormi, ale aj so špeciálnymi súbormi. Testovací dvojník by mal byť zároveň navrhnutý tak, aby bol plnohodnotne využiteľný počas automatizovaného testovania, ako aj pri manuálnom testovaní vykonávanom testermi.

Súčasťou kapitoly bola tiež analýza niekoľkých existujúcich riešení, ktoré sa v praxi využívajú pri testovaní softvérov pracujúcich so súbormi. Jednotlivé prístupy boli vyhodno-

tené z pohľadu ich výhod, nevýhod, zložitosti implementácie a kompatibility. Táto analýza poskytla prehľad možností, ktoré sú dostupné pre daný typ problému.

Na základe tohto porovnania bolo ako najvhodnejšie riešenie zvolené nahrádzanie knižnicových volaní prostredníctvom mechanizmu `LD_PRELOAD`.

Kapitola 4

Implementácia zdieľanej knižnice

Na implementáciu testovacieho dvojníka som sa rozhodol použiť programovací jazyk C. Keďže testovacím dvojníkom je zdieľaná knižnica v systéme GNU/Linux, ktorá má nahradiť funkcie z **GNU C Library**, čiže štandardnej knižnice programovacieho jazyka C, bol programovací jazyk C najrozumnejšou voľbou pre túto prácu.

Prvým krokom pred samotným písaním kódu bola analýza systémových a knižnicových volaní, ktoré diagnostický nástroj **sos**, respektíve **sos report**, využíva na prácu so súbormi. Musel som zistiť pôvodnú implementáciu týchto volaní, aké majú vstupné argumenty a aké majú návratové hodnoty. Po získaní všetkých potrebných informácií som mohol začať s vlastnou implementáciou knižnicových volaní vo vlastnej zdieľanej knižnici.

4.1 Analýza systémových volaní

Pred začatím implementovania obaľovacích funkcií v zdieľanej knižnici bolo nutné zistiť, ktoré systémové volania diagnostický nástroj `sos`, respektíve **sos report**, využíva pri práci so súbormi. Na zistenie konkrétnych systémových volaní som použil ladiaci nástroj **strace**, ktorý som spustil spoločne s nástrojom **sos report** a prepínačmi:

- -f / --follow-forks – sledovanie potomkov procesu a ich systémové volania, ak boli vytvorené sledovaným procesom ako výsledok systémových volaní `fork()`, `vfork()` alebo `clone()`
- -v / --no-abbrev – vypísanie neskrátenej verzie systémových volaní a ich argumentov
- --decode-fds=path – dekódovanie informácií spojených so súborovými deskriptormi. V tomto prípade konkrétne vypísanie dekódovanej cesty k súboru
- -o / --output – výstup nástroja **strace** bude vypísaný do určeného súboru

Následne som sa zameral na analýzu výstupu, ktorý som získal z nástroja **strace**. Výstup som detailne preskúmal a identifikoval systémové volania, ktoré nástroj **sos report** použil pri práci so súbormi. Na základe tejto analýzy som vytvoril zoznam s použitými systémovými volaniami. Viď tabuľku 4.1. Tento zoznam som neskôr použil na identifikáciu obaľovacích funkcií daných systémových volaní.

Syscall	Syscall
access	listxattr
newfstatat	ioctl
openat	lseek
fstat	sendfile
getdents64	close

Tabuľka 4.1: Zoznam použitých systémových volaní

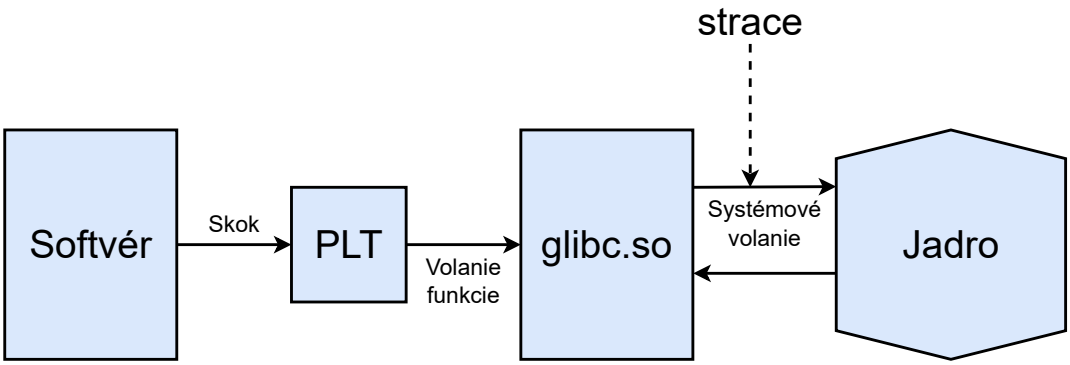
Ladiaci nástroj strace

Táto časť čerpá z manuálových stránok GNU/Linux [30].

Strace je užitočný diagnostický a ladiaci nástroj. Systémoví administrátori, diagnostici a riešitelia problémov ho považujú za neoceniteľný pri riešení problémov s programami, ku ktorým nie je dostupný zdrojový kód, pretože ich nie je potrebné prekladať, aby sa dali sledovať.

Študenti, hackeri a zvedavci zistia, že aj sledovaním obyčajných programov sa dá naučiť veľmi veľa o systéme a jeho systémových volaniach. Programátori zase ocenia, že systémové volania a signály predstavujú udalosti na rozhraní užívateľ – jadro systému, a ich detailné sledovanie je veľmi užitočné pri izolovaní chýb, overovaní správnosti správania a pri pokusoch o zachytenie pretekových podmienok (race conditions).

V najjednoduchšom prípade **strace** spustí zadaný príkaz a sleduje ho až do jeho ukončenia. Zachytáva a zaznamenáva všetky systémové volania, ktoré proces vykonáva, a signály, ktoré proces prijíma. Viď obrázok 4.1. Názov každého systémového volania, jeho argumenty a návratová hodnota sa vypisujú na štandardný chybový výstup (stderr), prípadne do súboru určeného pomocou prepínača -o.



Obr. 4.1: Nástroj strace

4.2 Obaľovacie funkcie systémových volaní

Ďalším krokom v procese tvorby zdieľanej knižnice bolo identifikovanie obaľovacích funkcií z **GNU C Library**, čiže štandardnej knižnice programovacieho jazyka C, ktoré reprezentujú dané systémové volania. Zistiť, ktoré knižnicové volania nástroj **sos report** používa, sa mi podarilo na základe zoznamu systémových volaní, ktorý som si vytvoril z výstupu nástroja **strace** a vďaka ďalšiemu užitočnému nástroju **ltrace**. Ten som spustil spoločne s nástrojom **sos report** a získal som zoznam knižnicových volaní nástroja **sos report**. Tento zoznam, výstup programu **ltrace**, som dôkladne analyzoval a identifikoval potrebné obaľovacie funkcie systémových volaní, z ktorých som vytvoril tabuľku 4.2.

Následne som sa venoval dôkladnému štúdiu manuálových stránok operačného systému GNU/Linux, kde som vyhľadával detailné informácie o vybraných systémových volaniach a ich obaľovacích funkciách. Zaujímali ma najmä ich pôvodné implementácie, aké argumenty tieto funkcie prijímajú a aké návratové hodnoty produkujú. Táto analýza bola kľúčová pre správne pochopenie správania týchto volaní, čo je nevyhnutné pri ich nahradzovaní alebo modifikovaní. Získané informácie som následne doplnil do tabuľky 4.2.

Obaľovacia funkcia	Systémové volanie	Vstupné argumenty	Návratové hodnoty
access()	access	const char *pathname, int mode	Úspech=0 Chyba=-1 `errno`=typ chyby
stat() stat64()	newfstatat	const char *pathname, struct stat / stat64 *statbuf	Úspech=0 Chyba=-1 `errno`=typ chyby
lstat() lstat64()	newfstatat	const char *pathname, struct stat / stat64 *statbuf	Úspech=0 Chyba=-1 `errno`=typ chyby
fstatat() fstatat64()	newfstatat	int dirfd, const char *pathname, struct stat / stat64 *statbuf, int flags	Úspech=0 Chyba=-1 `errno`=typ chyby
openat()	openat	int dirfd, const char *pathname, int flags	Úspech = deskriptor Chyba=-1 `errno`=typ chyby
open() open64()	openat	const char *pathname, int flags	Úspech = deskriptor Chyba=-1 `errno`=typ chyby
opendir()	openat	const char *pathname	Úspech = ukazovateľ na adresárový tok. Chyba=NULL `errno`=typ chyby
fopen()	openat	const char *filename, const char *mode	Úspech = FILE ukazovateľ na súbor. Chyba=NULL `errno`=typ chyby

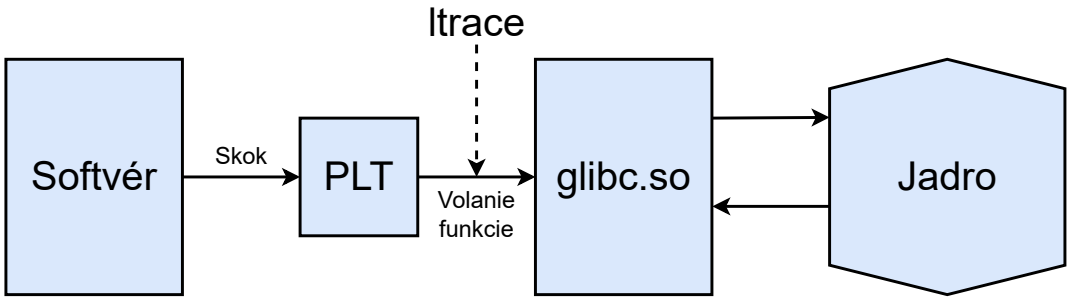
Tabuľka 4.2: Tabuľka knižnicových volaní

Nástroj ltrace

Táto časť čerpá z manuálových stránok GNU/Linux [17].

Nástroj **ltrace** je program, ktorý spustí zadaný príkaz a sleduje ho až pokiaľ neskončí. Zachytáva a zaznamenáva volania dynamických knižníc, ktoré sú volané spusteným procesom, a signály, ktoré tento proces prijíma. Dokáže tiež zachytávať a vypisovať systémové volania, ktoré program vykonáva. Nástroj **ltrace** funguje veľmi podobne ako **strace**. Viď obrázok 4.2.

Program **ltrace** zobrazuje parametre volaných funkcií a systémových volaní. Aby dokázal zistiť, aké argumenty má každá funkcia, potrebuje externé deklarácie prototypov funkcií. Tie sú uložené v súboroch nazývaných prototypové knižnice.



Obr. 4.2: Nástroj ltrace

4.3 Kód zdieľanej knižnice

Posledným krokom v rámci mojej práce bola samotná implementácia vlastných obaľovacích funkcií, ktoré majú za úlohu nahradiť funkcie zo štandardnej knižnice **GNU C Library**. Tieto funkcie som implementoval v programovacom jazyku C. Dôvodom pre voľbu tohto jazyka bola najmä kompatibilita so štandardnou knižnicou **GNU C Library**, ktorá takisto využíva jazyk C pre implementáciu svojich funkcií. Týmto prístupom som zabezpečil, že moje funkcie budú schopné správne spolupracovať so zvyškom systému.

V úvode implementácie každej jednotlivej obaľovacej funkcie som si definoval ukazovateľ na pôvodnú funkciu zo štandardnej knižnice **GNU C Library**. Tento ukazovateľ slúžil ako prostriedok na uchovanie adresy pôvodnej funkcie, ktorú som následne získal pomocou dynamického načítania cez funkciu **dlsym()**. Táto funkcia je súčasťou rozhrania pre dynamické linkovanie a umožňuje programom získať adresu konkrétneho symbolu zo zdieľanej knižnice za behu programu.

Ďalším dôležitým krokom bolo získanie cesty k súboru, ktorý má byť v systéme nahradený. Túto informáciu som získal z premennej prostredia s názvom `REPLACE_FILE`. Na získanie jej hodnoty som využil štandardnú funkciu **getenv()**, ktorá umožňuje čítať hodnoty premenných prostredia. Rovnako som si načítal aj cestu k falošnému súboru, ktorá je definovaná v premennej prostredia `FAKE_FILE_PATH`. Táto cesta bude použitá v prípade, že dôjde k zhode medzi pôvodne požadovaným súborom a súborom, ktorý má byť nahradený.

Práca so súbormi

V každej funkcii, ktorá pracuje so súbormi, sa vykonáva kontrola, či sa cesta k súboru uvedená nástrojom **sos report** alebo samotným užívateľom zhoduje s cestou uvedenou v premennej prostredia `REPLACE_FILE`. Ak sa tieto dve cesty zhodujú, znamená to, že ide o súbor, ktorý má byť nahradený. V takom prípade dôjde k prepísaniu pôvodnej cesty v argumente `pathname` za hodnotu z premennej prostredia `FAKE_FILE_PATH`. Týmto spôsobom je možné nahradiť aj súbor, ktorý v systéme reálne neexistuje. Argument `pathname` je bežne využívaný pri práci so súbormi a reprezentuje cestu k súboru, s ktorým daná funkcia pracuje. Výsledkom tejto zámeny je, že funkcia bude namiesto pôvodného súboru pracovať s falošným, čo umožňuje simulovať jeho správanie.

Funkcia dlsym

Funkcia **dlsym()** prijíma ukazovateľ dynamicky načítaného zdieľaného objektu vráteného funkciou **dlopen()** spolu s nulou ukončeným názvom symbolu a vracia adresu, na ktorej je daný symbol načítaný do pamäte. Ak sa symbol nenájde buď v určenom objekte, alebo v niektorej zo zdieľaných knižníc, ktoré boli automaticky načítané prostredníctvom **dlopen()**, funkcia **dlsym()** vráti `NULL`. [31]

Existujú aj dva špeciálne pseudo-ukazovatele:

- `RTLD_DEFAULT` – nájde prvý výskyt požadovaného symbolu podľa predvoleného poradia vyhľadávania v knižniciach.
- `RTLD_NEXT` – nájde nasledujúci výskyt funkcie v poradí vyhľadávania po aktuálnej knižnici.

To umožňuje vytvoriť obaľovaciu funkciu okolo inej funkcie, ktorá sa nachádza v inej zdieľanej knižnici. [31]

Rodičovské adresáre

Nástroj **sos report** pri práci so súbormi vykonáva predbežnú kontrolu existencie a dostupnosti rodičovského adresára daného súboru v súborovom systéme. To znamená, že predtým, než sa pokúsi získať prístup k samotnému súboru, overí, či adresár, v ktorom sa súbor nachádza, naozaj existuje a či má k nemu oprávnený prístup. Ak táto kontrola zlyhá, ak adresár neexistuje alebo prístup doň je obmedzený z hľadiska oprávnení, **sos report** daný súbor automaticky preskočí a ďalej s ním nepracuje.

Tento mechanizmus má za následok, že ak sa rodičovský adresár súboru, ktorý by mal byť nahradený falošným súborom, nenachádza v systéme, proces zámeny nebude možný. To predstavuje obmedzenie, ktoré môže negatívne ovplyvniť celkovú funkcionalitu implementácie.

Na základe tejto skutočnosti som sa rozhodol implementovať do svojich obaľovacích funkcií podobnú kontrolu existencie rodičovského adresára. V prípade, že pôvodná funkcia s pôvodnou cestou k rodičovskému adresáru vráti chybovú návratovú hodnotu, vykoná sa náhradná operácia. V rámci nej je pôvodná cesta k rodičovskému adresáru nahradená cestou k rodičovskému adresáru falošného súboru.

Týmto spôsobom je zabezpečené, že funkcia bude schopná pokračovať v práci, aj keď pôvodný adresár nie je prístupný, a zároveň bude možné úspešne simulovať alebo zachytiť prístup k súborom, ktoré by inak boli nástrojom **sos report** vynechané. Táto stratégia

zvyšuje robustnosť implementácie a zabezpečuje vyššiu mieru flexibility pri manipulácii s cestami k súborom a adresárom.

Simulovanie dynamicky generovaného obsahu súborov

Niektoré súbory v súborovom systéme operačného systému GNU/Linux nepredstavujú klasické statické súbory uložené na disku, ale ich obsah je generovaný dynamicky za behu systému. To znamená, že vždy, keď používateľ alebo softvér pristupuje k takémuto súboru, napríklad pomocou čítania, je jeho obsah vytvorený v reálnom čase operačným systémom, konkrétne jeho jadrom. [26]

Typickým a zároveň veľmi známym príkladom takého správania je adresár **/proc**, ktorý predstavuje virtuálny alebo pseudo súborový systém. Tento adresár neobsahuje reálne súbory na disku, ale poskytuje rozhranie pre prístup k rôznym systémovým informáciám, ako sú údaje o aktuálne bežiacich procesoch, stave jadra, systémovej pamäti alebo konfiguračných parametroch hardvéru. Virtuálny súborový systém **/proc** je vytváraný dynamicky pri spustení systému a jeho obsah zaniká pri vypnutí systému, keďže nie je trvalo uložený. [26]

Ako konkrétny príklad je možné uviesť súbor **/proc/uptime**, ktorý obsahuje informácie o dobe, počas ktorej je systém nepretržite spustený. Pri každom čítaní tohto špeciálneho súboru sa jeho obsah opätovne generuje, nejedná sa o statický textový súbor, ale o hodnoty vypočítané v danom okamihu priamo jadrom. [26]

V rámci mojej práce som chcel pokryť aj túto špecifickú funkcionalitu a umožniť simuláciu správania takýchto dynamických súborov. Cieľom bolo zabezpečiť, aby pri opakovanom čítaní rovnakého špeciálneho súboru nebol vrátený rovnaký obsah, ale aby sa jeho hodnota menila.

Na vyriešenie tohto problému som navrhol a implementoval mechanizmus, ktorý umožňuje obaľovacím funkciám pracovať s viacerými falošnými súbormi. Tieto súbory som pripravil s odlišným obsahom, a ich cesty som uložil do premennej prostredia `FAKE_FILE_PATH`. Na oddelenie viacerých ciest v rámci jednej premennej som zvolil ako separátor znak ";". V obaľovacích funkciách potom dochádza k spracovaniu tejto premennej prostredia a získaný reťazec s cestami je rozdelený na reťazce s jednotlivými cestami.

Pri každej ďalšej práci s daným špeciálnym súborom sa ako hodnota vstupného argumentu `pathname` použije ďalšia cesta zo zoznamu. To zaručí, že každé volanie funkcie pracuje stále s iným súborom. Týmto spôsobom je možné simulovať dynamicky meniaci sa obsah. Ak funkcia vyčerpá všetky predom definované cesty falošných súborov, ďalšie volania pracujú už len s poslednou použitou cestou.

4.4 Zhrnutie kapitoly

V tejto kapitole bola podrobne opísaná implementácia obaľovacích funkcií zdieľanej knižnice, ako aj pomocné nástroje a techniky, ktoré boli počas tohto procesu využité. Cieľom tejto časti práce bolo nielen priblížiť samotný technický postup programovania, ale aj vysvetliť všetky kroky, ktoré implementácii predchádzali.

Súčasťou tejto implementácie bola analýza systémových volaní, ktoré nástroj **sos report** využíva pri práci so súbormi. Táto analýza zahŕňala zbieranie dát pomocou diagnostických nástrojov a následné spracovanie získaných údajov.

Na základe získaných systémových volaní nasledovala fáza identifikácie konkrétnych obaľovacích funkcií zo štandardnej knižnice **GNU C Library**, ktoré reprezentujú dané systémové volania. V tejto časti kapitoly bol podrobne popísaný proces tejto identifikácie,

vrátane použitých nástrojov, dostupnej dokumentácie a postupov. Výstupom tejto časti bola prehľadná tabuľka, ktorá slúžila ako technický základ pre návrh a vývoj vlastných obaľovacích funkcií.

Záverečná časť kapitoly sa venuje samotnej implementácii zdieľanej knižnice, ktorá zabezpečuje nahrádzanie pôvodných knižnicových volaní. Podrobne je tu vysvetlené, ako knižnica pracuje s bežnými a špeciálnymi súbormi, aké mechanizmy využíva na získavanie vstupných dát, a ako pristupuje k dynamickému načítavaniu pôvodných funkcií, ako pristupuje k neexistujúcim rodičovským adresárom.

Kapitola 5

Výsledky experimentov testovacieho dvojníka

Testovanie zdieľanej knižnice a jej funkcií som uskutočnil pomocou experimentov na zariadení s operačným systémom **RHEL-9.7** a architektúrou procesora **x86_64**. Na pomoc pri testovaní som si vytvoril tri sady testov, z ktorých dve sady testujú priame volanie obaľovacích funkcií a ďalšia sada testuje knižnicu pomocou **shell** príkazov, ktoré pracujú so súbormi. Na záver som knižnicu manuálne otestoval spoločne s nástrojom **sos report**.

5.1 Základné testovanie obaľovacích funkcií

V programovacom jazyku C som si vytvoril sadu testov, krátkych kódov pre každé upravené knižnicové volanie. Programovací jazyk C som zvolil z dôvodu možnosti priameho volania týchto obaľovacích funkcií.Aby som overil funkčnosť knižnice, každý z týchto testov som spustil aj bez použitia zdieľanej knižnice, aj s použitím zdieľanej knižnice, ktorú som načítal pomocou mechanizmu `LD_PRELOAD`. Kód jednotlivých testov pozostával zo zavolania danej funkcie s určenou cestou k súboru a následným overením výsledku danej funkcie.

Prvý beh testov bol prevedený s použitím cesty k neexistujúcemu súboru súčasne za použitia zdieľanej knižnice a bez použitia zdieľanej knižnice. Výsledky týchto testov boli podľa očakávania. V prípade, v ktorom nebola použitá knižnica, výsledky testov indikovali, že súbor neexistuje. Naopak, v prípade, v ktorom bola použitá aj knižnica, výsledkami testov bolo úspešné prevedenie funkcií. Viď prílohu A.1 a tabuľku 5.1.

	Bez použitia knižnice		S použitím knižnice	
Testovaná funkcia	Očakávaný výsledok	Reálny výsledok	Očakávaný výsledok	Reálny výsledok
access()	neúspech	neúspech	úspech	úspech
stat()	neúspech	neúspech	úspech	úspech
lstat()	neúspech	neúspech	úspech	úspech
fstatat()	neúspech	neúspech	úspech	úspech
openat()	neúspech	neúspech	úspech	úspech
opendir()	neúspech	neúspech	úspech	úspech
open()	neúspech	neúspech	úspech	úspech
fopen()	neúspech	neúspech	úspech	úspech

Tabuľka 5.1: Porovnanie výsledkov testovacieho dvojníka s neexistujúcim súborom

Druhý beh testov bol realizovaný s použitím existujúceho súboru so zámerom otestovať nahradenie existujúceho súboru za mnou vytvorený testovací súbor. Tieto testy taktiež prebehli podľa očakávania a výsledky boli pozitívne. Viď prílohu A.2 a tabuľku 5.2.

	Bez použitia knižnice		S použitím knižnice	
Testovaná funkcia	Očakávaný výsledok	Reálny výsledok	Očakávaný výsledok	Reálny výsledok
access()	úspech	úspech	úspech	úspech
stat()	Veľkosť súboru	Veľkosť súboru	Rozdielna veľkosť súboru	Rozdielna veľkosť súboru
lstat()	Veľkosť odkazu	Veľkosť ozdkazu	Rozdielna veľkosť odkazu	Rozdielna veľkosť odkazu
fstatat()	Veľkosť súboru	Veľkosť súboru	Rozdielna veľkosť súboru	Rozdielna veľkosť súboru
openat()	úspech	úspech	úspech	úspech
opendir()	úspech	úspech	úspech	úspech
open()	úspech	úspech	úspech	úspech
fopen()	úspech	úspech	úspech	úspech

Tabuľka 5.2: Porovnanie výsledkov testovacieho dvojníka s existujúcim súborom

V poslednom behu som použil ďalšiu sadu testov, s ktorou som testoval simuláciu súborov s dynamicky generovaným obsahom, ako napríklad pri súbore **/proc/uptime**. Dané obaľovacie funkcie boli zavolané štyrikrát s tou istou súborovou cestou, avšak pri každom volaní funkcie bol použitý iný mnou vytvorený súbor s rozdielnou veľkosťou a obsahom. Výsledky týchto testov boli taktiež pozitívne. Viď prílohu A.3 a tabuľku 5.3.

	S použitím knižnice	
Testovaná funkcia	Očakávaný výsledok	Reálny výsledok
stat()	Rozdielna veľkosť súboru pri každom zavolaní	Rozdielna veľkosť súboru pri každom zavolaní
fstatat()	Rozdielna veľkosť súboru pri každom zavolaní	Rozdielna veľkosť súboru pri každom zavolaní
openat()	Rozdielny obsah súboru pri každom zavolaní	Rozdielny obsah súboru pri každom zavolaní
open()	Rozdielny obsah súboru pri každom zavolaní	Rozdielny obsah súboru pri každom zavolaní
fopen()	Rozdielny obsah súboru pri každom zavolaní	Rozdielny obsah súboru pri každom zavolaní

Tabuľka 5.3: Porovnanie výsledkov testovacieho dvojníka pri simulácii špeciálneho súboru

Zhrnutie

Testovanie funkcií ako samostatné jednotky prebehlo úspešne, všetky obaľovacie funkcie pracovali podľa očakávania a nenastal žiaden neočakávaný prípad.

5.2 Komplexnejšie testy

Ďalšiu testovaciu sadu som vytvoril v jazyku Python pomocou modulu **unittest**. V týchto testoch som použil volanie obaľovacích funkcií pomocou **shell** príkazov, ktoré boli v Pythone volané príkazom `run` z modulu **subprocess**. Zvolil som **shell** príkazy, ktoré umožňujú prácu so súbormi. Výhodou **shell** príkazov je, že používajú viacero systémových volaní súčasne, čiže som mohol otestovať knižnicu ako celok. Tieto testy som taktiež spustil s oboma testovacími prípadmi, čiže aj s použitím zdieľanej knižnice, aj bez použitia tejto knižnice.

	Bez použitia knižnice		S použitím knižnice	
Shell príkaz	Očakávaný výsledok	Reálny výsledok	Očakávaný výsledok	Reálny výsledok
test	neúspech	neúspech	úspech	úspech
find	neúspech	neúspech	úspech	úspech
head	neúspech	neúspech	úspech	úspech
tail	neúspech	neúspech	úspech	úspech
cat	neúspech	neúspech	úspech	úspech
ls	neúspech	neúspech	úspech	neúspech
wc	neúspech	neúspech	úspech	úspech

Tabuľka 5.4: Porovnanie výsledkov testovacieho dvojníka pri použití **shell** príkazov

Zhrnutie

Testovanie obaľovacích funkcií za pomoci **shell** príkazov prebehlo úspešne a hodnoty výsledkov boli podľa očakávania, až na jeden prípad. Viď prílohu A.4 a tabuľku 5.4. Test so **shell** príkazom `ls` skončil neúspechom aj pri použití zdieľanej knižnice. Po dôkladnom preskúmaní tohto príkazu za pomoci nástrojov **strace** a **ltrace** som zistil, že tento príkaz využíva na prácu so súborom aj knižnicové volania, ktoré nie sú implementované v mojej zdieľanej knižnici. Keďže knižnica bola primárne vytvorená pre nástroj **sos report**, niektoré knižnicové volania, ktoré pracujú so súbormi a zrovna ich nástroj **sos report** nepoužíva, sa v knižnici nenachádzajú. Tento problém plánujem do budúcna opraviť a rozšíriť zoznam obaľovacích funkcií v zdieľanej knižnici.

5.3 Použitie testovacieho dvojníka v praxi

V poslednom kroku testovania som manuálne otestoval testovacieho dvojníka spoločne s nástrojom **sos report**, konkrétne s verziou **sos-4.9.1-1.el9.noarch**. Zvolil som prípady, kedy nástroj **sos report** pracuje s:

1. neexistujúcim bežným súborom
2. existujúcim bežným súborom
3. existujúcim bežným súborom

Nahradenie neexistujúceho bežného súboru

V prvom prípade som spustil nástroj **sos report** najprv bez zdieľanej knižnice, aby som dokázal, že ak súbor neexistuje, softvér **sos report** ho nezozbiera.

```
# ls /var/lib/insights/
# sos report -o insights --batch
...
# tar -xvf /var/tmp/sosreport*.tar.xz | grep "/var/lib/insights"
sosreport*/var/lib/insights/
```

Následne som vytvoril pomocný súbor a spustil ten istý príkaz, ale tentokrát aj s mechanizmom `LD_PRELOAD`, pomocou ktorého som načítal zdieľanú knižnicu, a s potrebnými premennými prostredia.

```
# ls /var/lib/insights/
# mkdir /root/files
# touch /root/files/host-details.json
# LD_PRELOAD=/root/bp/src/te_do_lib.so
  REPLACE_FILE=/var/lib/insights/host-details.json
  FAKE_FILE_PATH=/root/files/host-details.json
  sos report -o insights --batch
...
# tar -xvf /var/tmp/sosreport*.tar.xz | grep "var/lib/insights"
sosreport*/var/lib/insights/
sosreport*/var/lib/insights/host-details.json
```

Testovanie dopadlo podľa očakávania, testovací dvojník nahradil neexistujúci súbor za pomocný súbor a nástroj **sos report** ho zozbieral.

Nahradenie existujúceho bežného súboru

V druhom testovacom prípade som testoval nahradenie existujúceho súboru za pomocný súbor. Najprv som spustil nástroj **sos report** bez testovacieho dvojníka, aby som získal výsledok na porovnanie.

```
# cat /var/log/rhsm/rhsmcertd.log
Tue Apr 29 10:10:50 2025 [INFO] rhsmcertd is shutting down...
# sos report --batch
...
# tar -xvf /var/tmp/sosreport*.tar.xz | grep "/var/log/rhsm/rhsmcertd.log"
sosreport*/var/log/rhsm/rhsmcertd.log
# cat sosreport*/var/log/rhsm/rhsmcertd.log
Tue Apr 29 10:10:50 2025 [INFO] rhsmcertd is shutting down...
```

Rovnaký príkaz som spustil aj s testovacím dvojníkom, aby som otestoval nahradenie súboru.

```
# cat /var/log/rhsm/rhsmcertd.log
Tue Apr 29 10:10:50 2025 [INFO] rhsmcertd is shutting down...
# cat >> /root/files/test_file << EOF
> Nahradenie specialneho suboru
> EOF
# cat /root/files/test_file
Nahradenie specialneho suboru
# LD_PRELOAD=/root/bp/src/te_do_lib.so
  REPLACE_FILE=/var/log/rhsm/rhsmcertd.log
  FAKE_FILE_PATH=/root/files/test_file
  sos report --batch
...
# tar -xvf /var/tmp/sosreport*.tar.xz | grep "/var/log/rhsm/rhsmcertd.log"
sosreport*/var/log/rhsm/rhsmcertd.log
# cat sosreport*/var/log/rhsm/rhsmcertd.log
Nahradenie specialneho suboru
```

Po porovnaní výsledkov je možné potvrdiť, že daný súbor bol nahradený za pomocný súbor.

Nahradenie existujúceho špeciálneho súboru

V poslednom testovacom prípade bola testovaná práca so špeciálnym súborom. Vybral som príklad, kde **sos report** zbiera špeciálny súbor `/proc/diskstats`. Nástroj som najprv spustil s pôvodným špeciálnym súborom, bez použitia testovacieho dvojníka.

```
# cat /proc/diskstats
 253 0 vda 13559 62 878461 8017 3668 2403 307207 3339 0 7855 12602 0 0 0 0
 253 1 vda1 13323 62 867621 7969 3658 2403 307199 3338 0 8170 11307 0 0 0 0
# sos report -o block --batch
...
# tar -xvf /var/tmp/sosreport*.tar.xz | grep "/proc/diskstats"
sosreport*/proc/diskstats
# cat sosreport*/proc/diskstats
 253 0 vda 13559 62 878461 8017 3691 2407 307555 3354 0 7863 12625 0 0 0 0
 253 1 vda1 13323 62 867621 7969 3681 2407 307547 3353 0 8178 11322 0 0 0 0
```

Na záver som spustil **sos report** spolu s testovacím dvojníkom a pripravenými premennými prostredia.

```
# cat /proc/diskstats
 253 0 vda 13559 62 878461 8017 3668 2403 307207 3339 0 7855 12602 0 0 0 0
 253 1 vda1 13323 62 867621 7969 3658 2403 307199 3338 0 8170 11307 0 0 0 0
# cat >> /root/files/test_file << EOF
> Nahradenie specialneho suboru
> EOF
# cat /root/files/test_file
Nahradenie specialneho suboru
# LD_PRELOAD=/root/bp/src/te_do_lib.so
  REPLACE_FILE=/proc/diskstats
  FAKE_FILE_PATH=/root/files/test_file
  sos report -o block --batch
...
# tar -xvf /var/tmp/sosreport*.tar.xz | grep "/proc/diskstats"
sosreport*/proc/diskstats
# cat sosreport*/proc/diskstats
Nahradenie specialneho suboru
```

Aj v poslednom testovacom prípade testovací dvojník pracoval podľa očakávania a nahradil špeciálny súbor za pomocný súbor a **sos report** tento súbor zozbieral.

Zhrnutie

Výsledky testovania boli pozitívne. Testovací dvojník pracoval podľa očakávania. Nenastal žiaden nečakaný stav. Viď tabuľku 5.5.

	Bez použitia knižnice		S použitím knižnice	
Testovací prípad	Očakávaný výsledok	Reálny výsledok	Očakávaný výsledok	Reálny výsledok
Neexistujúci bežný súbor	Súbor nebude zozbieraný	Súbor nebude zozbieraný	Súbor bude nahradený a zozbieraný	Súbor bude nahradený a zozbieraný
Existujúci bežný súbor	Súbor bude zozbieraný	Súbor bude zozbieraný	Súbor bude nahradený a zozbieraný	Súbor bude nahradený a zozbieraný
Existujúci špeciálny súbor	Súbor bude zozbieraný	Súbor bude zozbieraný	Súbor bude nahradený a zozbieraný	Súbor bude nahradený a zozbieraný

Tabuľka 5.5: Porovnanie výsledkov testovacieho dvojníka pri použití s nástrojom **sos report**

Kapitola 6

Záver

Cieľom tejto práce bolo vytvorenie testovacieho dvojníka pre nástroj **sos report** od spoločnosti Red Hat, ktorý je schopný pracovať so špeciálnymi a bežnými súbormi.

Pre úspešné vytvorenie testovacieho dvojníka bolo nutné preskúmať problematiku testovania softvéru a naštudovať všetky potrebné informácie ohľadom testovania, testovacích dvojníkov a systémoch GNU/Linux. Získané poznatky boli spracované a dôkladne popísané v druhej kapitole tejto práce.

Ďalším krokom k vytvoreniu testovacieho dvojníka bolo štúdium a analýza existujúcich riešení problému práce so súbormi. Vybrané existujúce riešenia boli zanalyzované a ich výhody a nevýhody boli spracované do tabuľky, na základe ktorej bolo spravené porovnanie a zvolené konkrétne riešenie v podobe zdieľanej knižnice.

Výsledkom tejto práce je navrhnutý testovací dvojník v podobe zdieľanej knižnice, ktorá je pomocou mechanizmu prednačítania dynamických knižníc načítaná pred štandardnú knižnicu **GNU C Library** programovacieho jazyka C. Na implementáciu knižnice bol zvolený programovací jazyk C, aby sa zachovala kompatibilita s **GNU C Library**. Testovací dvojník dokáže správne pracovať so špeciálnymi a bežnými súbormi a nahradiť zvolené súbory pomocou premenných prostredia pre testovacie účely nástroja **sos report**. Testovací dvojník je taktiež schopný simulovať dynamické generovanie obsahu súborov prostredníctvom vytvorených pomocných súborov.

Testovanie funkcionality testovacieho dvojníka prebehlo pomocou sady umelých testov na zariadení s operačným systémom RHEL-9.7. Testované boli ako konkrétne jednotlivé funkcie, tak aj knižnica ako celok. Testovanie dopadlo úspešne, získané výsledky sa zhodovali s očakávanými výsledkami, až na jeden prípad, kedy testy narazili pri príkaze `ls` na obaľovaciu funkciu, ktorá nie je implementovaná v zdieľanej knižnici. Na záver bolo otestované správanie testovacieho dvojníka v praxi, kde bola zdieľaná knižnica použitá s nástrojom **sos report**. Výsledky tohto testovania dopadli takisto podľa očakávania a potvrdili funkčnosť testovacieho dvojníka.

Literatúra

[1] Ammann, P. a Offutt, J. *Introduction to Software Testing*. 1. vyd. Cambridge University Press, 2008. ISBN 978-0-521-88038-1.

[2] Bharadwaj, R. *Mastering Linux Kernel Development*. 1. vyd. Birmingham: Packt Publishing Ltg., 2017. ISBN 978-1-78588-305-7.

[3] bordergate. *LD_PRELOAD Exploitation*. 2023. Dostupné z: https://www.bordergate.co.uk/ld_preload-exploitation/#:~:text=LD_PRELOAD%20Exploitation%20,intercept%20calls%20from%20a%20program. Citované: 06.04.2025.

[4] Brendel, R.; Wesarg, B.; Tschüter, R.; Weber, M.; Ilsche, T. et al. Generic Library Interception for Improved Performance Measurement and Insight. In: Bhatele, A.; Boehme, D.; Levine, J. A.; Malony, A. D. a Schulz, M., ed. *Programming and Performance Visualization Tools*. Springer International Publishing, 2019, s. 21–37. ISBN 978-3-030-17872-7. Dostupné z: https://doi.org/10.1007/978-3-030-17872-7_2.

[5] Brown, N. *Overlay Filesystem*. 2024. Dostupné z: https://docs.kernel.org/_sources/filesystems/overlayfs.rst.txt. Citované: 01.05.2025.

[6] Ceelen, R. *7 Manual Testing Types Explained*. 2022. Dostupné z: https://www.testmonitor.com/blog/7-manual-testing-types-explained. Citované: 28.04.2025.

[7] ComputerNetworkingNotes. *Different Types of Files in Linux*. 2025. Dostupné z: https://www.computernetworkingnotes.com/linux-tutorials/different-types-of-files-in-linux.html. Citované: 07.04.2025.

[8] Eteimorde, Y. *"Everything is a file" Explained*. 2023. Dostupné z: https://dev.to/eteimz/everything-is-a-file-explained-g2a. Citované: 06.04.2025.

[9] Foundation, P. S. *Unittest.mock — mock object library*. 2024. Dostupné z: https://docs.python.org/3/library/unittest.mock.html. Citované: 25.04.2025.

[10] Fowler, M. *Test Double*. 2006. Dostupné z: https://martinfowler.com/bliki/TestDouble.html#:~:text=messages%20it%20was%20sent.%20,the%20calls%20they%20were%20expecting. Citované: 06.04.2025.

[11] Gomes, P. *Unit Testing and the Arrange, Act and Assert (AAA) Pattern*. 2017. Dostupné z: https://medium.com/@pjbgf/title-testing-code-ocd-and-the-aaa-pattern-df453975ab80. Citované: 30.04.2025.

[12] Hope, C. *Regular file*. 2024. Dostupné z: https://www.computerhope.com/jargon/r/regular-file.htm. Citované: 12.04.2025.

[13] Jelínek, L. *Jádro systému Linux*. 1. vyd. Brno: Computer Press, a. s., 2008. ISBN 978-80-251-2084-2.

[14] Kili, A. *Explanation of "Everything is a File" and Types of Files in Linux*. 2023. Dostupné z: https://www.tecmint.com/everything-is-file-and-types-of-files-linux/. Citované: 07.04.2025.

[15] Koul, A. *LD_PRELOAD a Powerful Tool!!!* 2024. Dostupné z: https://medium.com/%40akankshakoul/ld-preload-a-powerful-tool-c5170f90d9d0. Citované: 06.04.2025.

[16] Leloudas, P. *Introduction to Software Testing: A Pratical Guide to Testing, Design, Automation, and Execution*. 1. vyd. APress, 2023. ISBN 978-1-4842-9513-7.

[17] ltrace developers. *Ltrace - A library call tracer*. 2023. Dostupné z: https://man7.org/linux/man-pages/man1/ltrace.1.html. Linux User Commands, Section 1.

[18] Meszaros, G. XUnit Test Patterns: Refactoring Test Code. In:. 1. vyd. Addison-Wesley, 2007, kap. 23. ISBN 978-0-13-149505-0.

[19] Mydreams.cz. *Jak nastavit a používat chroot prostředí pro izolaci aplikací v Linuxu?* 2025. Dostupné z: https://www.mydreams.cz/cz/wiki/5863-jak-nastavit-a-pouzivat-chroot-prostredi-pro-izolaci-aplikaci-v-linuxu.html. Citované: 27.04.2025.

[20] Myers, G. J.; Sandler, C. a Badgett, T. *The Art of Software Testing*. 3. vyd. John Wiley & Sons, Inc, 2011. ISBN 978-1-118-03196-4.

[21] Osherove, R. a Khorikov, V. *The Art of Unit Testing*. 3. vyd. Shelter Island: Manning Publications Co., 2024. ISBN 978-1-61729-748-9.

[22] Pavlíček, M. *Chroot prostředí*. 2003. Dostupné z: https://www.abclinuxu.cz/clanky/bezpecnost/chroot-prostredi-i. Citované: 27.04.2025.

[23] RedHat. *What is an sos report and how to create one in Red Hat Enterprise Linux?* 2025. Dostupné z: https://access.redhat.com/solutions/3592. Citované: 20.04.2025.

[24] Ruiz, A. *Mock Objects: Shortcomings and Use Cases*. 2007. Dostupné z: https://www.oracle.com/technical-resources/articles/enterprise-architecture/mock-shortcomings.html. Citované: 05.04.2025.

[25] Shubham, P. *Journey of a Linux file (Part 1) - What is a File?* 2024. Dostupné z: https://prasha7.medium.com/journey-of-a-linux-file-part-1-what-is-a-file-202d9edea3fe. Citované: 07.04.2025.

[26] Site24x7. *Introduction to Linux Uptime.* Dostupné z: https://www.site24x7.com/learn/linux/uptime.html. Citované: 18.04.2025.

[27] Sobell, M. G. Mistrovství v RedHat a Fedora Linux: pro verze Fedora Core 2 až 5 a RedHat 3 a 4. In:. 1. vyd. Preložil David Krásenský and Lubomír Ptáček. Brno: Computer Press, a.s., 2006, kap. 27. ISBN 80-251-1152-0.

[28] Sobell, M. G. Mistrovství v RedHat a Fedora Linux: pro verze Fedora Core 2 až 5 a RedHat 3 a 4. In:. 1. vyd. Preložil David Krásenský and Lubomír Ptáček. Brno: Computer Press, a.s., 2006, kap. 12. ISBN 80-251-1152-0.

[29] Stapp, L.; Roman, A. a Pilaeten, M. *ISTQB® Certified Tester Foundation Level: A Self-Study Guide Syllabus v4.0.* 4. vyd. Cham: Springer Nature Switzerland, 2023. ISBN 978-3-031-42766-4.

[30] strace developers. *Strace - trace system calls and signals.* 2023. Dostupné z: https://man7.org/linux/man-pages/man1/strace.1.html. Linux User Commands, Section 1.

[31] The Linux man-pages project. *Dlsym(3) - obtain the address of a symbol from a shared object.* 2023. Dostupné z: https://man7.org/linux/man-pages/man3/dlsym.3.html. Linux Programmer's Manual, Section 3.

[32] The Linux man-pages project. *Syscall(2) - indirect system call.* 2023. Dostupné z: https://man7.org/linux/man-pages/man2/syscall.2.html. Linux Programmer's Manual, Section 2.

[33] The Linux man-pages project. *Syscalls(2) - Linux system calls.* 2023. Dostupné z: https://man7.org/linux/man-pages/man2/syscalls.2.html. Linux Programmer's Manual, Section 2.

[34] Turner Trauring, I. *Code Injection on Linux and macOS with LD_PRELOAD.* 2017. Dostupné z: https://www.getambassador.io/blog/code-injection-on-linux-and-macos. Citované: 15.04.2025.

[35] University, N. M. S. *Introduction to Linux.* 2023. Dostupné z: https://hpc.nmsu.edu/onboarding/linux/files-folders/. Citované: 12.04.2025.

[36] Whittaker, J. A. What Is Software Testing? And Why Is It So Hard? *IEEE Software.* IEEE, 2000, zv. 17, č. 1, s. 70–79. Dostupné z: https://doi.org/10.1109/52.819971.

[37] Xiao, L.; Zhao, G.; Wang, X. et al. An empirical study on the usage of mocking frameworks in Apache software foundation. *Empirical Software Engineering*, 2024, zv. 29, č. 2, s. 39. Dostupné z: https://doi.org/10.1007/s10664-023-10410-y.

Príloha A

Výsledky testov

A.1 Testovanie obaľovacích funkcií - nahradenie neexistujúceho súboru

```
========== access_test without library ===============
./access_test
access: No such file or directory
========== access_test with library ==================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH=/root/bp/tests/c_tests/test.txt
./access_test
File exists.
========== fstatat_test without library ==============
./fstatat_test
fstatat: No such file or directory
========== fstatat_test with library =================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH=/root/bp/tests/c_tests/test.txt
./fstatat_test
Size: 0 bytes
========== stat_test without library =================
./stat_test
stat: No such file or directory
========== stat_test with library ====================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH=/root/bp/tests/c_tests/test.txt
./stat_test
Size 0 bytes
========== lstat_test without library ================
./lstat_test
lstat: No such file or directory
========== lstat_test with library ===================
```

```
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH=/root/bp/tests/c_tests/symlink.txt
./lstat_test
Size of link: 8 bytes
========== openat_test without library ===============
./openat_test
openat: No such file or directory
========== openat_test with library ==================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH=/root/bp/tests/c_tests/test.txt
./openat_test
File open with openat(): fd = 3
========== opendir_test without library ==============
./opendir_test
opendir: No such file or directory
========== opendir_test with library =================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH=/root/bp/tests/c_tests/test.txt
./opendir_test
Directory open.
========== open_test without library =================
./open_test
open: No such file or directory
========== open_test with library ====================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH=/root/bp/tests/c_tests/test.txt
./open_test
File open with open(): fd = 3
========== fopen_test without library ================
./fopen_test
fopen: No such file or directory
========== fopen_test with library ===================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH=/root/bp/tests/c_tests/test.txt
./fopen_test
File open with fopen().
```

A.2 Testovanie obaľovacích funkcií - nahradenie existujúceho súboru

```
========== access_test without library ===============
./access_test
File exists.
========== access_test with library ==================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH=/root/bp/tests/c_tests/test.txt
./access_test
File exists.
========== fstatat_test without library ==============
./fstatat_test
Size: 104857600 bytes
========== fstatat_test with library =================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH=/root/bp/tests/c_tests/test.txt
./fstatat_test
Size: 0 bytes
========== stat_test without library =================
./stat_test
Size 104857600 bytes
========== stat_test with library ====================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH=/root/bp/tests/c_tests/test.txt
./stat_test
Size 0 bytes
========== lstat_test without library ================
./lstat_test
Size of link: 104857600 bytes
========== lstat_test with library ===================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH=/root/bp/tests/c_tests/symlink.txt
./lstat_test
Size of link: 8 bytes
========== openat_test without library ===============
./openat_test
File open with openat(): fd = 3
========== openat_test with library ==================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH=/root/bp/tests/c_tests/test.txt
./openat_test
```

```
File open with openat(): fd = 3
========== opendir_test without library ==============
./opendir_test
Directory open.
========== opendir_test with library =================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH=/root/bp/tests/c_tests/test.txt
./opendir_test
Directory open.
========== open_test without library =================
./open_test
File open with open(): fd = 3
========== open_test with library ====================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH=/root/bp/tests/c_tests/test.txt
./open_test
File open with open(): fd = 3
========== fopen_test without library ================
./fopen_test
File open with fopen().
========== fopen_test with library ===================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH=/root/bp/tests/c_tests/test.txt
./fopen_test
File open with fopen().
```

A.3 Testovanie obaľovacích funkcií - simulácia dynamického generovania obsahu

```
========== fstatat_test with library =================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH="/root/bp/tests/c_tests_dynamic/test1.txt;
/root/bp/tests/c_tests_dynamic/test2.txt;
/root/bp/tests/c_tests_dynamic/test3.txt"
./fstatat_test
Size 26 bytes File: /root/mydir/myfile.txt
Size 21 bytes File: /root/mydir/myfile.txt
Size 17 bytes File: /root/mydir/myfile.txt
Size 17 bytes File: /root/mydir/myfile.txt
========== stat_test with library ====================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH="/root/bp/tests/c_tests_dynamic/test1.txt;
/root/bp/tests/c_tests_dynamic/test2.txt;
/root/bp/tests/c_tests_dynamic/test3.txt"
./stat_test
Size 26 bytes File: /root/mydir/myfile.txt
Size 21 bytes File: /root/mydir/myfile.txt
Size 17 bytes File: /root/mydir/myfile.txt
Size 17 bytes File: /root/mydir/myfile.txt
========== openat_test with library ==================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH="/root/bp/tests/c_tests_dynamic/test1.txt;
/root/bp/tests/c_tests_dynamic/test2.txt;
/root/bp/tests/c_tests_dynamic/test3.txt"
./openat_test
File open with openat(): fd = 3
Content of /root/mydir/myfile.txt:
test file1.............
File open with openat(): fd = 3
Content of /root/mydir/myfile.txt:
test file2.........
File open with openat(): fd = 3
Content of /root/mydir/myfile.txt:
test file3 .....
File open with openat(): fd = 3
Content of /root/mydir/myfile.txt:
test file3 .....
========== open_test with library ====================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
```

```
FAKE_FILE_PATH="/root/bp/tests/c_tests_dynamic/test1.txt;
/root/bp/tests/c_tests_dynamic/test2.txt;
/root/bp/tests/c_tests_dynamic/test3.txt"
./open_test
File open with open(): fd = 3
Content of /root/mydir/myfile.txt:
test file1.............
File open with open(): fd = 3
Content of /root/mydir/myfile.txt:
test file2.........
File open with open(): fd = 3
Content of /root/mydir/myfile.txt:
test file3 .....
File open with open(): fd = 3
Content of /root/mydir/myfile.txt:
test file3 .....
========== fopen_test with library ==================
LD_PRELOAD=/root/bp/src/te_do_lib.so
REPLACE_FILE=/root/mydir/myfile.txt
FAKE_FILE_PATH="/root/bp/tests/c_tests_dynamic/test1.txt;
/root/bp/tests/c_tests_dynamic/test2.txt;
/root/bp/tests/c_tests_dynamic/test3.txt"
./fopen_test
File open with fopen().
Content of /root/mydir/myfile.txt:
test file1.............
File open with fopen().
Content of /root/mydir/myfile.txt:
test file2.........
File open with fopen().
Content of /root/mydir/myfile.txt:
test file3 .....
File open with fopen().
Content of /root/mydir/myfile.txt:
test file3 .....
```

A.4 Testovanie pomocou shell príkazov

```
test_cat_nonexistent_file_with_library (test1.TestCat) ... ok
test_cat_nonexistent_file_without_library (test1.TestCat) ... ok
test_stat_nonexistent_file_with_library (test1.TestFind) ... ok
test_stat_nonexistent_file_without_library (test1.TestFind) ... ok
test_head_nonexistent_file_with_library (test1.TestHead) ... ok
test_head_nonexistent_file_without_library (test1.TestHead) ... ok
test_ls_nonexistent_file_with_library (test1.TestLs) ... FAIL
test_ls_nonexistent_file_without_library (test1.TestLs) ... ok
test_tail_nonexistent_file_with_library (test1.TestTail) ... ok
test_tail_nonexistent_file_without_library (test1.TestTail) ... ok
test_test_nonexistent_file_with_library (test1.TestTest) ... ok
test_test_nonexistent_file_without_library (test1.TestTest) ... ok
test_wc_nonexistent_file_with_library (test1.TestWc) ... ok
test_wc_nonexistent_file_without_library (test1.TestWc) ... ok

======================================================================
FAIL: test_ls_nonexistent_file_with_library (test1.TestLs)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/root/bp/tests/test1.py", line 101,
  in test_ls_nonexistent_file_with_library
    self.assertEqual(result.returncode,0)
AssertionError: 2 != 0
----------------------------------------------------------------------
Ran 14 tests in 0.024s
FAILED (failures=1)
```