VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

AUTOMATIZOVANÉ TESTOVÁNÍ SYSTÉMU FITCRACK

AUTOMATED TESTING OF FITCRACK SYSTEM

SEMESTRÁLNÍ PROJEKT
TERM PROJECT

AUTOR PRÁCE **JURAJ CHRIPKO**
AUTHOR

VEDOUCÍ PRÁCE **Ing. RADEK HRANICKÝ,**
SUPERVISOR

BRNO 2018

Zadání bakalářské práce/20498/2017/xchrip00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Chripko Juraj**
Obor: Informační technologie
Téma: **Automatizované testování systému Fitcrack**
Automated Testing of Fitcrack System
Kategorie: Analýza a testování softwaru

Pokyny:

1. Seznamte se s architekturou a implementací distribuovaného systému Fitcrack.
2. Nastudujte existující metodologie pro automatizované testování softwaru.
3. S využitím vámi zvolených technik navrhněte sadu testů pro jednotlivé části systému.
4. Navržené řešení implementujte.
5. Vyzkoušejte testování v praxi na virtuální+reálné distribuované síti a zhodnoťte dosažené výsledky.

Literatura:

- D. P. Anderson, "BOINC: a system for public-resource computing and storage," Fifth IEEE/ACM International Workshop on Grid Computing, 2004, pp. 4-10. doi: 10.1109/GRID.2004.14.
- D. P. Anderson, E. Korpela and R. Walton, "High-performance task distribution for volunteer computing," First International Conference on e-Science and Grid Computing (e-Science'05), Melbourne, Vic., 2005, pp. 8 pp.-203. doi: 10.1109/E-SCIENCE.2005.51.
- HRANICKÝ Radek, HOLKOVIČ Martin, MATOUŠEK Petr a RYŠAVÝ Ondřej. On Efficiency of Distributed Password Recovery. The Journal of Digital Forensics, Security and Law. 2016, roč. 11, č. 2, s. 79-96. ISSN 1558-7215.
- HRANICKÝ Radek, ZOBAL Lukáš, VEČEŘA Vojtěch a MATOUŠEK Petr. Distributed Password Cracking in a Hybrid Environment. In: Proceedings of SPI 2017. Brno: Universita Obrany v Brně, 2017, s. 75-90. ISBN 978-80-7231-414-0.

Pro udělení zápočtu za první semestr je požadováno:

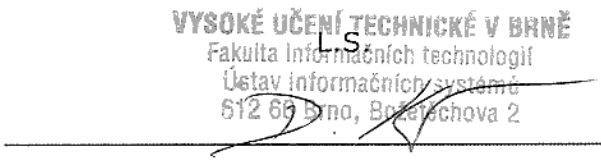
- Body 1 až 3.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Hranický Radek, Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018



doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Táto práca si kladie za cieľ navrhnúť a implementovať automatizované testy pre systém Fitcrack, distribuovaný systém na lámanie hesiel. Testy musia byť takisto distribuované a prenositeľné, keďže časť systému pracuje na klientských zariadeniach. Pre väčšie pokrytie testovacích prípadov sú použité viaceré techniky testovania. V práci je rozobraná teória fungovania systému Fitcrack, ako aj požiadavky na testy, teória prístupov k testovania a ich jednotlivý prínos pri odhaľovaní chýb. Práca ďalej obsahuje návrh systémových testov s ohľadom na zistenia o fungovaní systému Fitcrack.

Abstract

This thesis aims to design and implement automated tests for Fitcrack, distributed password cracking system. Tests also have to be distibuted and cross-platform, as part of the system operates on client's machine. For better code coverage multiple testing technics are combined. In thesis is discussed Fitcrack functioning theory as well as requirements for tests, software testing principles and their contribution to error detection. Thesis further includes Fitcrack tests design with consideration for findings about Fitcrack.

Kľúčové slová

Fitcrack, BOINC, testovanie, jednotkové testy, integračné testy

Keywords

Fitcrack, BOINC, testing, unit testing, integration testing

Citácia

CHRIPKO, Juraj. *Automatizované testování systému Fitcrack*. Brno, 2018. Semestrální projekt. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Hranický,

Automatizované testování systému Fitcrack

Prehlásenie

Prehlasujem, že som túto bakalárskou prácu vypracoval samostatne pod vedením pana Ing. Radka Hranického. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Juraj Chripko
27. januára 2018

Poďakovanie

Rád by som poďakoval vedúcemu práce Ing. Radkovi Hranickému za odbornú pomoc, usmerňovanie pri riešení práce a trpezlivosť.

Obsah

1 Úvod **3**

2 Návrh softvéru **4**
2.1 Model veľkého tresku 4
2.2 Model '*programuj a opravuj*' 4
2.3 Vodopádový model 5
2.4 V model 5
2.5 Špirálový model 6
2.6 Agilné metodológie 6

3 Testovanie softvéru **8**
3.1 Verifikácia a validácia 8
3.2 Metódy testovania softvéru 8
3.2.1 Testovanie čiernej skrinky vs. testovanie bielej skrinky 8
3.3 Úrovne testovania softvéru 9
3.3.1 Jednotkové testy 9
3.3.2 Integračné testy 9
3.3.3 Systémové testy 9
3.3.4 Akceptačné testy 10
3.3.5 Regresné testovanie 10
3.4 Automatizácia testov 10

4 Systém Fitcrack **11**
4.1 Nástroje používané systémom Fitcrack 11
4.1.1 BOINC 11
4.1.2 Hashcat 12
4.2 Vlastné časti systému Fitcrack 12
4.2.1 WebAdmin 13
4.2.2 REST API 13
4.2.3 XtoHashcat 13
4.2.4 Generátor 13
4.2.5 Asimilátor 15
4.2.6 Runner 15

5 Návrh testov pre systém Fitcrack **17**
5.1 Integračné testy 17
5.2 Jednotkové testy 18

6 Záver **20**

Literatúra **21**

Kapitola 1

Úvod

V posledných pár rokoch sme zažili priam explóziu počtu vývojárov softvéru. Netreba sa tomu čudovať, ak sa pozrieme na množstvo používateľov počítačového softvéru. Veď koľko rôznych programov, aplikácii, webových stránok každý z nás denne navštívi, otvorí. Tie isté aplikácie však môžu využívať aj ľudia zapletený v trestnej činnosti. Každá z týchto aplikácii ponúka nejaké riešenie na zašifrovanie našich dát, či sú to už štandardné riešenie používané bežne na napríklad bezpečné prezeranie webu, alebo proprietárne šifrovanie súborov na disku. Otvorenie súboru chráneného heslom môže trvať od niekoľkých minút až po stovky rokov, keďže najefektívnejší spôsob ako otvoriť takýto súbor je zistiť heslo pomocou porovnávania hešov. To znamená vygenerovať nejaké heslo, pomocou ktorého vygenerujeme heš a ten porovnáme s pôvodným. Asi cítite, že zistiť zložitejšie heslo by mohlo pri použití aj tých najlepších osobných počítačov trvať roky. Riešením by mohol byť systém, ktorý túto úlohu rozdelí na veľa malých úloh a do výpočtu zahrnie veľké množstvo bežných osobných počítačov. To je hlavný cieľ systému Fitcrack.

Obrovské množstvo dostupného softvéru a tiež jeho používateľov má za následok aj obrovskú popularizáciu vývoja softvéru. Pred niekoľkými rokmi dokázala softvér vyvíjať len veľmi malá skupinka ľudí o ktorej sme mohli povedať, že sú profesionáli a poznajú nástrahy vývoja softvéru, no aj tak je toto obdobie poznačené katastrofami, ktoré spôsobili pomerne malé, ale zásadné chyby v kóde. [7] Minimalizáciu týchto chýb si kladie za úlohu časť vývojového cyklu softvéru zvaná testovanie. Testovanie softvéru je dnes brané ako nutnosť, nakoľko softvér vyvíjajú už aj ľudia s oveľa menšími znalosťami a skúsenosťami, čo neberiem ako negatívum, práve naopak, ale ďalej to zvyšuje potreby testovania softvéru.

Ako každá časť vývoja softvéru, tak aj k testovaniu existuje veľké množstvo techník a odporúčaní kedy testovať, čo testovať, ako testovať. Zavedením štandartov do vývoja softvéru sa zvyšuje jeho kvalita, hlavne pri veľkých projektoch, keďže sa na projekte uchádza množstvo ľudí a je zložité ich koordinovať, alebo práve projektov, ktoré vedú menšie a menej skúsené týmy ľudí. Veľké firmy samozrejme majú postupy, ktorých sa držia, no menšie firmy môžu ľahko implementovať štandardizované postupy. Preto je aj pre túto prácu dôležité zvoliť správny postup testovania. Niektoré z najväčších softvérových firiem dokonca pokladajú testovanie za tak dôležité, že vyvinuli modely životného cyklu vývoja softvéru postavené na testovaní. Aj keď sa Fitcrack nedrží žiadneho z týchto modelov, ich prehľadu ako aj prehľadu techník testovania sa venujú samostatné kapitoly.

V prvých dvoch kapitolách je preto rozobraná problematika vývoja a testovania softvéru. Práca ďalej obsahuje popis jednotlivých súčastí systému Fitcrack z ohľadom na dôležitosť informácii pri návrhu a implementácii testov pre tento systém, návrh testov pre systém Fitcrack a nakoniec zhrnutie zistení a plány do budúcna.

Kapitola 2

Návrh softvéru

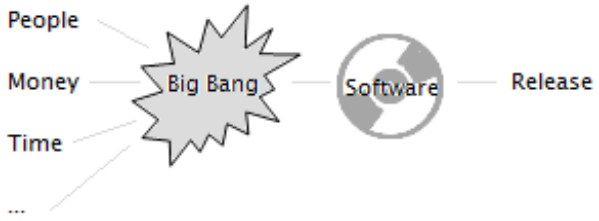
Od myšlienky niečo vytvoriť až po finálny produkt je dlhá cesta, ktorú môžeme rozdeliť na tieto časti:

- **Plánovanie** - Konceptuálny návrh a plánovanie
- **Analýza** - Zhromažďovanie požiadavok a ich analýza
- **Návrh** - Návrh architektúry a špecifikácii
- **Vývoj** - Testovanie, implementácia

Niektoré z týchto fáz sa môžu opakovať, alebo nemusia byť pri vývoji vôbec použité. Proces podľa ktorého sa softvér vytvára sa nazývajú model životného cyklu vývoja softvéru. Niekoľko z nich je nižšie uvedených a vysvetlených. Žiadny z nich nie je správny, alebo vyslovené zlý. Každý z nich má pozitíva a negatíva, no ich myšlienka odhaľuje skutočnosti na ktoré je treba si dať pri vývoji a následne pri testovaní pozor.

2.1 Model veľkého tresku

Najjednoduchší model, je vyobrazený na obrázku 2.1. Tento model pracuje s myšlienkou, že dáte ľuďom dostatok času, zdrojov(peňazí) a po vynaložení množstva energie je hotový produkt. Model nehovorí nič o formálnom návrhu, alebo testovaní, no môže mať za výsledok použiteľný produkt [11].

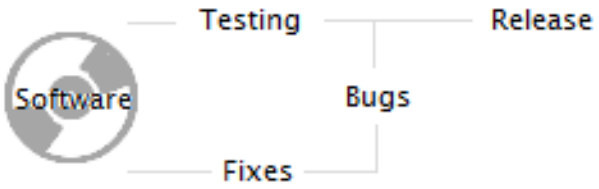


Obr. 2.1: Model veľkého tresku [10]

2.2 Model '*programuj a opravuj*'

O modeli '*programuj a opravuj*' 2.2 môžeme povedať, že sa z veľkej časti podobá na model veľkého tresku 2.1, no je iteratívny a uznáva potrebu testovania. Formálny návrh stále nemá

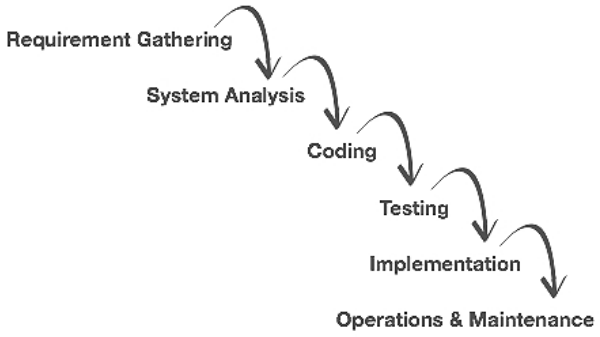
veľkú váhu, takže programátori naprogramujú softvér, ten sa otestuje a vráti vývojárom. Tento postup sa opakuje až kým nie je závažnosť chýb na prijateľnej úrovni. Tento model sa často objavuje v iných modeloch ako fáza vývoja softvéru [11].



Obr. 2.2: Model '*programuj a opravuj*' [10]

2.3 Vodopádový model

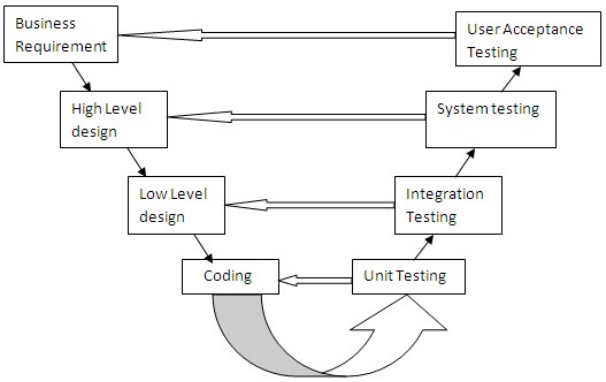
Prvý formálne popísaný a asi najznámejší model životného cyklu softvéru 2.3. Bol prezentovaný už v roku 1970 a vyjadruje postupnosť fáz: získavanie požiadavok, systémová analýza, programovanie, testovanie, implementácia a údržba systému. Často používaný prí výučbe, je jednoduchý a ľahko sa vysvetľuje. Definuje a plánuje fázy, čím pomáha dodržovať termíny a výstupy jednotlivých fáz, ale je veľmi málo flexiblný, keďže počíta stým, že požiadavky sa nebudú meniť, takže plánovanie všetkých úloh projektu je treba naplánovať do posledného detaily, čo je náročné a drahé [11].



Obr. 2.3: Vodopádový model [10]

2.4 V model

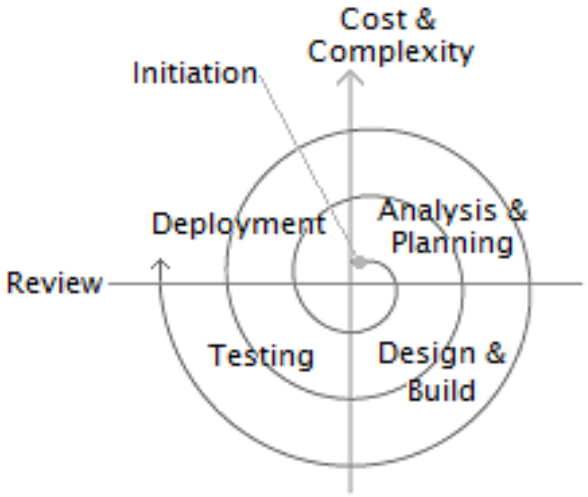
Podobný vodopádovému modelu 2.3, no každá fáza pred programovaním má k sebe doplnkovú fázu, ktorá overuje výsledky fázy pred programovaním 2.4. Napríklad keď sú stanovené požiadavky na softvér, tak hneď môžu byť napísané akceptačné testy 3.3.4. No reálne overenie môže nastať až keď je softvér implementovaný. Vtomto modely je použitých niekoľko testovacích techník, pričom každá z nich overuje inú časť návrhu, designu, alebo implementácie [11].



Obr. 2.4: V model [10]

2.5 Špirálový model

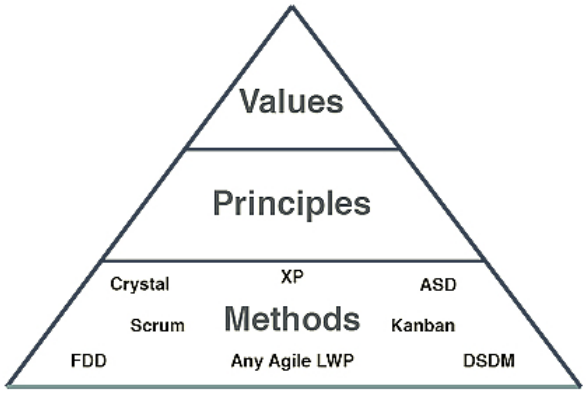
Vychádza z vodopádového modelu 2.3, no vylepšuje jeho najväčší nedostatok, flexibilu, zavedením iterácii 2.5. Každá iterácia má stále súbor výstupov, ktoré musia byť dodržané, no požiadavky sa môžu počas životného cyklu softvéru meniť, čo je v reáli typické. Špirálový model je základom filozofie Agilných metodológii.



Obr. 2.5: Špirálový model [10]

2.6 Agilné metodológie

Agilné metodológie sú modely životného cyklu softvéru postavené na filozofii iteratívnenho a inkrementálneho vývoja softvéru, kde sa požiadavky aj riešenia menia počas vývoja 2.6. Pojem '*agile*' bol spopularizovaný v kontexte '*Manifesto for Agile Software Development*' [9]. Medzi agilné metodológie patrí aj programovanie riadené testami (*Test driven development*), kde sa testy píšu až pre implementáciou a samotná implementácia je riadená testami. Tento model ďalším dôkazom dôležitosti testovania pri vývoji softvéru.



Obr. 2.6: Pyramída agilných metodológii [10]

Kapitola 3

Testovanie softvéru

Testovanie softvéru je aktivita, ktorá pomáha vyhodnocovať, či chovanie systému zodpovedá tomu, čo je pre neho definované. Proces testovania pomáha tvorcom softvéru vyhodnotiť, ktoré časti sa správajú správne a ktoré nie, čiže identifikuje chyby v programe. Chyby môžu vznikať z rôznych príčin, no všeobecne platí, že čím skôr sa chyba odhalí, tým ľahšie a teda aj lacnejšie je opraviteľná. Testovanie je súčasťou verifikácie systému, ktorá je zase súčasťou procesu zabezpečovania kvality softvéru (Quality Assurance, QA). [6]

3.1 Verifikácia a validácia

Tieto 2 pojmy sú často zamieňané, alebo považované za synonymá. Aj keď obidva pojmy slúžia na zvyšovanie kvality softvéru, každý z nich znamená niečo iné. Verifikácia overuje, či softvér spĺňa technickú špecifikáciu, čiže úlohu na ktorú bol navrhnutý. Verifikovať sa dá viacerými postupmi:

- **Statická analýza** skúma vlastnosti softvéru bez jeho spúšťania.
- **Dynamická analýza** overuje vlastnosti softvéru práve jeho spúšťaním.
- **Formálna verifikácia** pomocou formálnych metôd overuje, že systém odpovedá špecifikácii.
- **Testovanie** skúma softvér spúšťaním programu za účelom zvýšenia jeho kvality.

Validácia potvrdzuje, že softvér spĺňa obchodné podmienky a požiadavky používateľa, nie technickú špecifikáciu.

3.2 Metódy testovania softvéru

Pri návrhu testov je veľmi dôležité do akej miery tester pozná vnútornú štruktúru systému. Ak sa podieľal na vývoji a dobre ho pozná bude testy navrhovať úplne iným spôsobom ako osoba, ktorá môže mať skúsenosti s testovaním, no systém nepozná. Táto kapitola ďalej rozoberá rozdiely v testovaní pri rozdielnych znalostiach o systéme.

3.2.1 Testovanie čiernej skrinky vs. testovanie bielej skrinky

Pri testovaní metódou čiernej skrinky sa softvér považuje za nepriehľadnú skrinku a tester nemá žiadne informácie o jej vnútornej štruktúre. Má informácie o úlohe softvéru, ale nie

o spôsobe vkonávania tejto úlohy. Pri tejto metóde je dôležité poznať vlastnosti množiny vstupných dát a vybrať z nich podmnožinu, ktorá otestuje čo najväčšiu časť možných prípadov. Tento prístup je výhodný pri odhãlovaní chýb v špecifikácii. Metóda sa používa pri akceptačnom aj regresnom testovaní.

Testovanie bielej skrinky je naopak postavené na fakte, že tester pozná vnútornú štruktúru softvéru. K takémuto testovanie je potrebný zdrojový kód, alebo pseudokód popisujúci chovanie softvéru. Tým, že tester presne pozná štruktúru softvéru dokáže lepšie odhadnúť chovanie systému a pri výbere testovacích prípadov môže voliť napríklad také testovacie prípady aby boli pri testovaní použité všetky vetvy kódu.

Existuje prístup kombinujúci testovanie čiernej a testovanie bielej skrinky, **testovanie šedej skrinky**. Tester pozná vnútorné dátové šktruktúry a algoritmy, ale nepozná reálnu imlplementáciu, čiže nedokáže navrhovať testy s ohľadom na pokrytie kódu. Tento prístup je čím ďalej, tým viac používaný hlavne kvôli zjednodušeniu návrhu testov bez nutnosti sa podrobne vyznať v implementácii.

3.3 Úrovne testovania softvéru

Ako ukazujú modely životného cyklu softvéru, rôzne časti (úrovne) systému je možné testovať pri rôznych príležitostiach. Nedá sa testovať stále a všetko, je potrebné určiť kedy budú ktoré časti testované.

3.3.1 Jednotkové testy

Jednotkové testy (Unit tests) pracujú iba s jednou jednotkou(komponentom, modulom), ktorú testujú. Preto sa im niekedy hovorí aj testovanie komponent (Component testing), alebo testovanie modulov (Module testing). Jednotka by mala byť čo najmenšia, ale dostatočne veľká, aby bola samostatne testovateľná. Tieto testy sú často robené priamo programátormi a chyby sú odstraňované ihneď ako je to možné.

3.3.2 Integračné testy

Integračné testy overujú funkčnosť rozhrania medzi komponentami, ktoré môžu byť malé ako pri jednotkových testoch a postupne s vývojom softvéru sa zväčšovať až sa testuje integrácia celého podsystému do už používaného systému. Tento prístup sa nazýva inkrementálne integrovanie a chyby dokáže odhaliť pomerne skoro, no je drahý, keďže sa musia simulovať rozhrania, ktoré ešte neboli implementované. Opačný prístup k integračný testom sa nazýva integrácia veľkým treskom a testuje sa až keď sú všetky súčasti systému implementované. Tento prístup je rýchly a jednoduchý, no je náročne zistiť príčinu zlyhania takejto neskorej integrácie. Ideálny prístup je niekde medzi týmito dvoma extrémnimy prípadmi. Testovať väčšie súčasti, ktoré by mohli spôsobovať problémy.

3.3.3 Systémové testy

Systémové testy sú robené na takmer hotovom sofvéry a je u nich vyžadovaná nezávislosť, takže ich robia viaceré tými alebo tretia strana. Cieľom systémových testov je overiť, že softvér spĺňa špecifikáciu.

3.3.4 Akceptačné testy

Akceptačné testy sú narozdiel od ostatných typov testov zodpovednosťou zákazníka, keďže overujú, či softvér spĺňa požiadavky zákazníka a či je pripravený na prevádzku. Hľadanie chýb nie je primárnou úlohou akceptačných testov.

3.3.5 Regresné testovanie

Regresné testy sú robené stále, ak sa zmení nejaká časť softvéru na zaistenie, že nová verzia zdieľa vlastnosti starej a že sa do systému nezaniesli nové chyby. Toto testovanie prebieha často, takže regresné testy sú veľakrát automatizované.

3.4 Automatizácia testov

Testovanie je časovo a finančne náročná činnosť, preto je dobré ho čo najviac automatizovať. Napríklad ak by sa mal manuálne testovať celý systém po každom vydaní novej verzie, ktorá len opravuje chyby starších verzií, bola by to sizyfovská práca. No nie všetko testovanie sa dá automatizovať, čo môže byť aj dobre. Ak automatické testy minú chybu pri prvom spustení, tak tú chybu budú míňať pri každom ďalšom testovaní. Chyba sa stane imúnna voči testom.

Kapitola 4

Systém Fitcrack

Fitcrack [8] je distribuovaný systém pre obnovu hesiel šifrovaných médii a prelomenie kryptografických hešov. Pre samotné lámanie hešov používa Hashcat, ktorý je momentálne najrýchlejšie riešenie na trhu. Správu hosťov a rozdeľovanie úloh zabezpečuje Berkeley Open Infrastructure framework. Viac o týchto nástrojoch obsahuje nasledujúca kapitola. Vlastné súčasti systému Fitcrack sú ďalej popísané v 4.2

4.1 Nástroje používané systémom Fitcrack

Táto kapitola popisuje nástroje požívané systémom Fitcrack. Ich presné fungovanie je pre testovanie nepodstatné vedieť, ale dôležitá je ich funkcia v systéme.

4.1.1 BOINC

Berkeley Open Infrastructure for Network Computing (BOINC) [3] je platforma pre distribuované výpočty, ktorá natívne podporuje dynamické pripojovanie uzlov cez internet. BOINC bol primárne vytvorený ako nástroj na verejné zdieľanie výpočtových prostriedkov v oblastiach ako je meteorológia, medicína, astrofyzika a ďalšie. Dobrovoľník môže poskytnúť svoje výpočtové kapacity niektorému z projektov. Každý projekt sa zaoberá niečím iným, napríklad projekt *Search for Extraterrestrial Intelligence* (SETI) [4] využíva výpočtový výkon dobrovoľníckych staníc na analyzovanie signálov z vesmíru a hľadanie mimozemského života. Tento princíp sa nazýva volunteer computing. BOINC podporuje aj tzv. grid computing, teda zapojenie množstva počítačov a výpočtových stredísk z rôznych geografických lokalít.

Boinc funguje na princípe server/klient, kde server predstavuje server projektu ku ktorému sa pripája ľubovoľný počet klientov. Klient sa môže pripojiť z rôznych zariadení, na ktorom beží rôzny operačný systém a sú vybavené rôznymi jednotkami, na ktorých ma byť prevádzaný výpočet (CPU, GPU, FPGA). Server zodpovedá za prideľovanie pracovných úloh klientom, zaisťuje sťahovanie najnovších spustiteľných súborov na klientské zariadenia. Klient môže byť pripojený na viac projektov a sám si určuje koľko výpočtového výkonu chce poskytnúť na jednotlivé úlohy. BOINC ráta s prípadmi, kedy sa klienti pripoja alebo odpoja počas výpočtu a ponúka niekoľko spôsobov riešenia, no konkrétne riešenie je implementované tvorcom projektu. Keďže je BOINC priamo navrhnutý pre distribuovaný výpočet cez internet ponúka množstvo bezpečnostných mechanizmov na prácu v nedôveryhodnom prostredí.

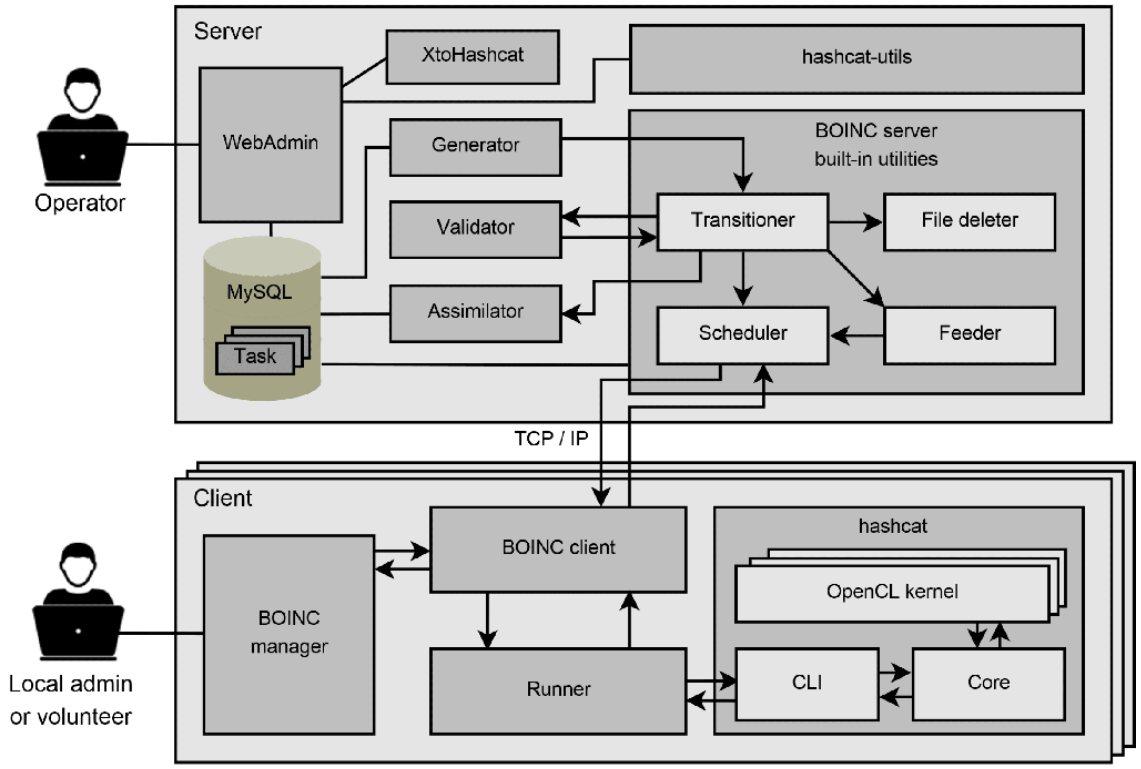
4.1.2 Hashcat

Hashcat [1] je svetovo najrýchlejším riešením na lámanie hesiel na jednom stroji. Je zadarmo, má otvorený zdrojový kód, je spustiteľný na Windowse, Linuxe aj na macOS. Používa OpenCL, ktoré je už dnes kompatibilné s väčšinou zariadení (CPU, GPU, DSP, FPGA, ...) a podporuje množstvo formátov.

OpenCL (Open Computing Language) [2] je štandard pre paralelné programovanie na rôznych typov procesorov používaných v osobných počítačoch, serveroch, mobilných telefónov a vstavaných systémov.

4.2 Vlastné časti systému Fitcrack

Ostatné súčasti sú popísané v tejto kapitole.



Obr. 4.1: Architektúra systému Fitcrack. [8]

BOINC server, BOINC client, BOINC manager a Validator sú súčasti BOINC-u a neboli nijak modifikované.

Je potrebné testovať iba časti systému aspoň čiastočne implementované tvorcami Fitcracku:

- **Generátor**,
- **Asimilátor**,
- **Runner**,
- **XtoHashcat**.

4.2.1 WebAdmin

WebAdmin je grafické užívateľské rozhranie pre ovládanie systému Fitcrack a komunikuje s databázou pomocou REST API, na ktoré sa zameriava študent Matúš Múčka v svojej budúcej bakalárskej práci. Testovanie WebAdmina nie je súčasťou tejto práce, keďže práve prebieha vývoj nového grafického užívateľského rozhrania.

4.2.2 REST API

Vyššie spomenuté **REST API** slúži na komunikáciu GUI a serverových modulov, konkrétne XtoHashcat a Hashcat-utils, čo sú programy podporujúce Hashcat (overovanie formátu hashu, ...). So zbytkom serverových modulov komunikuje cez databázu. Toto API je schopné vytvárať nové úlohy, sledovať pripojených klientov, mapovať ich na existujúce úlohy, riadiť výpočet úloh, ...

4.2.3 XtoHashcat

XtoHashcat je skript v jazyku Python3 vyvinutý tvorcami systému Fitcrack, ktorý dokáže zobrať na vstup ľubovoľný súbor z množiny podporovaných súborov a na výstup dá heš pre Hashcat. To znamená, že dokáže detekovať formát súboru a extrahovať heš zo súboru.

4.2.4 Generátor

Generátor, alebo Work Generator je BOINC server démon bežiaci v nekonečnom cykle, v ktorom generuje nové pracovné úlohy(work units) pre pripojených klientov k danej úlohe(job). Generátor vytvára 2 typy úloh: benchmark a normálna. Benchmark úlohy sa generujú klientom, ktorý sa práve pripojili, alebo keď na danom klientovi nastala chyba. Normálne úlohy sú pracovné úlohy obsahujúce všetky potrebné informácie pre spustenie klientskej časti systému Fitcrack. Úloha môže byť v jednom z nasledujúcich stavov:

- **0 - ready**. Výpočet neprebieha.
- **1 - finished**. Výpočet bol dokončený a heslo nájdené.
- **2 - exhausted**. Výpočet bol dokončený, ale heslo nebolo nájdené.
- **3 - malformed**. Úloha obsahuje chybný vstup.
- **4 - timeout**. Úloha bola ukončená z dôvodu prekročenia časového limitu.
- **10 - running**. Prebieha výpočet.
- **12 - finishing**. Negenerujú sa nové pracovné úlohy, ale výpočet stále prebieha na niektorých klientských staniciach.

V module Generátor je implementovaný plánovací algoritmus, ktorý vytvára pracovné úlohy na mieru konkrétnej klientskej stanici. Princíp činnosti modulu Generátor je popísaný algoritmom 1.

Algoritmus 1: Princíp činnosti systemového modulu Generátor

```
while 1 do
    /* Inicializácia                                                    */
    Zmaž uzly, ktoré sa podieľajú na riešení úlohy, ktorých pracovný balíček je už
     dokončený, buď úspešne (finished) alebo neúspešne (exhausted)
    Keď niektorý z balíčkov presiahol stanovenú dobu ukončenia, nastav jeho stav
     na finished (12)
    foreach Bežiaci pracovný balíček (stav ≥ 10) do
        if Nie je nastavený čas zahájenia then
            Nastáva čas zahájenia na aktuálny čas
        end
        if K balíčku sa viažu masky hesiel then
            Ulož ích do príslušného poľa, ktoré odpovedá balíčku
        end
        Nájdi uzly, ktoré sa majú podieľať na výpočte (a zatiaľ sa nepodieľajú) a
         ulož ich do databázy
        /* Benchmark                                                    */
        foreach Pridelený aktívny uzol, ktorý ma stav Benchmark (0) do
            if Uzol ešte nemá naplánovaný benchmark then
                Naplánuj benchmark pre tento uzol
            end
        end
        /* Výpočet                                                      */
        foreach Aktívny uzol v stave Normal (1) do
            if Počet naplánovaných úloh pre uzol ≥ 2 then
                Pokračuj na ďalší uzol
            end
            if Stav uzlu je Running (10) then
                Vygeneruj novú úlohu podľa typu balíčka, prípadne znovu prideľ
                 nedokončené úlohy
            end
            if Stav uzle je Finished (12) then
                Znovu prideľ nedokončené úlohy, ak taká neexistuje, nastav stav uzlu
                 na Done (3)
            end
            /* Kontrola stavu                                           */
            if Stav balíčka je Finished (12) a neobsahuje žiadne úlohy then
                if Aktuálny čas > plánovaný Čas ukončenia then
                    Nastav stav balíčka na Timeout (4)
                else
                    if Aktuálny index ≥ maximálny index then
                        Nastav stav balíčku na Exhausted (2)
                        /* vyčerpaný stavový priestor                   */
                    else
                        Nastav stav balíčku na Ready(0)
                        /* výpočet bol pozastavený                      */
                    end
                end
            end
        end
    end
    Čakaj stanovený časový interval
end
```

4.2.5 Asimilátor

Asimilátor je takisto BOINC server démon, ktorý spracováva valídne výsledky. Nekonečný cyklus v ktorom beží je poskytnutý BOINC-om, ale samotná akcia, ktorá sa vykonáva pri prijatí výsledku je implementovaná tvorcami Fitcrack-u. Funkčnosť akcie je prezentovaná algoritmom 2.

Algoritmus 2: Princíp činnosti modulu Asimilátor

```
while 1 do
    switch typ do
        case benchmark do
            if Výsledok je v poriadku (kód 0) then
                Prečítaj zmeraný výkon a ulož ho do databázy
            else
                Naplánuj nový benchmark na dlhší čas
            end
        end
        case normal do
            if Heslo bol nájdené (kód 0) then
                Prečítaj heslo a ulož ho do databázy (prípadne aj do cache)
                Uprav stav úlohy na Finished (1)
                Pošli všetkým zapojeným uzlom pokyn k ukončeniu výpočtu
                Odstráň/archivuj nedokončené úlohy
                Prečítaj časvýpočtu úlohy a ulož ho
            else
                if Heslo nebolo nájdené (kód 1) then
                    Aktualizuj veľkosť úlohy
                    Aktualizuj počet overených hesiel
                else
                    /* Chyba výpočtu                                    */
                    Zruš úlohy pro daný uzol a nastav im príznak retry
                    Vynuluj výkon uzu v databázi a naplánuj nový benchmark
                end
            end
        end
    end
end
```

4.2.6 Runner

Runner je aplikácia bežiaca na klientskom počítači ovládajúca Hashcat. BOINC server pošle údaje o pracovnej úlohe BOINC klientovi, ktorý pomocou systémových volaní spúšťa runner a predáva mu tieto informácie v podobe vstupných súborov. Runner následne tieto informácie predá nástroju Hashcat. Runner sleduje stav výpočtu a po je jeho skončení predá výsledky BOINC klientovi.

Generátor a Asimilátor komunikujú výhradne pomocou databázy, kde to validátor a runner riadi priamo BOINC. Okrem BOINC klienta nie je potrebné na klientskom zariadení

nič inštalovať, BOINC klient sám zo servera sťahuje najnovšie spustiteľné súbory v závislosti na architektúra klientského zariadenia.

Kapitola 5

Návrh testov pre systém Fitcrack

K návrhu testov pre systém Fitcrack potrebujeme poznať architektúru systému Fitcrack 4 a moduly, ktoré budú testované 4.2. Nápovedu kedy sa budú jednotlivé moduly testovať by nám mala poskytnúť kapitola 2 a spôsoby testovania popisuje zase kapitola 3.3. Požiadavky na testy vyplývajú z úvodu práce, ako aj z kapitoly o systéme Fitcrack 4. Táto časť práce používa znalosti z predošlých kapitol a jej súčasťou je návrh testov pre systém Fitcrack, ktorý je podložený znalosťami z kapitol 2, 3 a 4.

Ideálne by sa návrh testov mal držať V modelu životného cyklu softvéru 2.4, no keďže systém Fitcrack je vyvíjaný pre väčšie organizácie, ktoré si určujú vlastnosti systému a tie sa neustále menia, tak vývoj systému, ako aj jeho testov musí prebiehať iteratívne, tak ako to popisuje špirálový model 2.5

Pre testovanie systému Fitcrack som sa rozhodol použiť kombináciu jednotkových (unit) testov a integračného testovania. Jednotkové testy nedokážu odhaliť chyby na serveri spojené s komunikáciou cez databázu a integračné testy sú v systéme Fitcrack možné iba cez databázu, čo je jediné rozhranie, ktoré väčšina modulov používa a tieto testy nemajú dostatočné pokrytie funkcionality systému. Kombinácia týchto testov bude takisto slúžiť na regresné testovanie. Z pohľadu metódy testovania sa na moduly budem pozerať ako na šedé skrinky 3.2.1, keďže poznám ako moduly fungujú, no presnú implementáciu nepoznám. Testovanie bude prebiehať v reálnom prostredí a úplne overenie funkčnosti systému sa prevedie až keď sa k projektu pripojí klient.

5.1 Integračné testy

Integračné testy 3.3.2 budú implementované v jazyku Python3 a bude použitý framework unittest. Unittest, tiež známy ako PyUnit, [5] je štandardný unit test framework pre Python. Pre implementáciu integračných testov som si vybral Python hlavne kvôli jeho jednoduchosti, použiteľnosti frameworku unittest, možnosti jednoducho používať REST API, ako aj pripojenie do databázy. Testovací prípad (test case) bude pozostávať z viacerých častí, keďže na začiatku je potrebné overiť, či je testovaný endpoint REST API dostupný, či nám vráti očakávanú hodnotu a nakoniec aj či zmena v databázi odpovedá vrátenej hodnote. Na zmenu v databázi by mal reagovať niektrorý z testovaných serverových modulov a vykonať akciu, ktorú je potrebné skontrolovať. Jeden testovací prípad popisuje algoritmus 3.

Algoritmus 3: Princíp integračných testov.

```
Zavolaj endpoint REST API 4.2.2
Skontroluje hodnotu, ktorú vrátilo API s referenčnou hodnotou
if Akcia vykoná zmenu v databázi then
    Skontroluj, či databáza obsahuje správny záznam
    /* Zmenu v databázi je možné porovnať ako s vráteným výsledkom,
       tak aj s referenčnou hodnotou.                                */
end
Počkaj, kým testovaný serverový modul vykoná akciu
Skontroluj správnosť vykonanej akcie
```

5.2 Jednotkové testy

Jednotkové testy 3.3.1 budú napísané pre Generátor, Asimilátor a Runner. Všetky tieto moduly sú napísané v C/C++, takže je možné použiť napríklad Google Test framework. Moduly budú testované metódou šedej skrinky 3.2.1, keďže ich implementácia je pomerne zložitá, no je potrebné pokryť čo najväčšiu časť kódu, hlavne pri module Generátor 4.2.4 a klientskej časti, module Runner 4.2.6.

Keďže je Generátor asi najzložitejší modul a jeho plánovací algoritmus sa integračným testovaním overiť nedá, je potrebné implementovať jednotkové testy pre všetky časti tohto modulu samostatne. Je potrebné testovať nasledujúce funkcie:

- plánovanie pracovnej úlohy typu benchmark,
- plánovanie normálne pracovnej úlohy,
- plánovací algoritmus (čierna skrinka),
- prerozdeľovanie nedokončených pracovných úloh,
- ukončenie výpočtu na ostatných staniciach po nájdení hesla.

Princíp činnosti modulu Asimilátor 4.2.5 je pomerne jednoduchý, ale takisto jeho správnosť sa integračným testovaním overiť nedá. Celý modul Asimilátor bude braný ako jedna šedá skrinka a podľa toho budú napísané testy.

Runner bude testovaný pred spustením benchmarku na klientovi, z nasledujúcich dôvodov:

- Kedže runner musí pracovať rovnako na rôznych platformách, musí byť aj testovaný na rôznych platformách
- Runner priamo spúšťa a kontroluje Hashcat, ktorého správanie závisí od zariadenia na ktorom je spustený (ovládače, typ GPU, typ CPU, ...)
- Benchmark sa spúšťa pre každého klienta keď sa 1. krát pripojí, alebo sa niečo zmenilo, alebo nastala chyba
- Pri benchmarku je spúšťaný runner a Hashcat
- Simulovanie chýb Hashcatu by bolo veľmi náročné a neefektívne

Pri module Runner bude testované:

- komunikácia so serverom,
- schopnosť správne čítať konfiguračný súbor,
- spustenie akcie benchmark,
- správnosť kombinácie argumentov pri spúšťaní aplikácie Hashcat 4.1.2,
- ďalšie funkcie pridávané počas vývoja.

Kapitola 6

Záver

V rámci práce boli zhromaždené a naštudované informáciu potrebné pre návrh a implementáciu testov, pričom návrh testov je súčasťou tejto práce. Potrebné informácie sú rozdelené na: požiadavky na testy systému, prehľad modelov životného cyklu vývoja softvéru, teórie testovania, princíp systému Fitcrack a návrhu samotných testov. Takisto bola overená funkčnosť architektúry akceptačných testov pomocou skriptu, ktorý dokáže čítať a interpretovať údaje z databázy. Problém by mohol byť s posielaním výsledkov testovania runner-u na klientovi, no spolieham sa na BOINC a funkcionalitu.

Ďalšia práca by sa mala zamerať na meranie pokrytia kódu/logiky systému testovacími prípadmi a samotnú iplementáciu testovacích prípadov, pre oba spôsoby testovania. Je potrebné stanoviť metriky a spôsoby merania pokrytia logiky systému testovacími prípadmi. Pre implementáciu jednotkových testov je vítaná úzka spolupráca s vývojármi konkrétnych modulov, no nie je vyžadovaná. Kapitola o testovaní softvéru 3 obsahuje veľké množstvo pojmov, ktoré nie sú formálne definované, preto by bolo vhodné ešte pred samotnou implementáciou testov naštudovať a doplniť tieto definície.

Testy systému by mohli byť použité aj na diagnostiku softvéru v prevádzke, no to bude vyžadovať spoluprácu s vývojármi WebAdmin-a a REST API, ktoré komunikuje so serverovými modulmi, aby výsledky testov/ diagnostiky mohli byť vizualizované užívateľovi.

Literatúra

[1] hashcat *advanced password recovery*. 2018.
URL https://hashcat.net/hashcat/

[2] OpenCL Overview. 2018.
URL https://www.khronos.org/opencl/

[3] Overview of BOINC. 2018.
URL https://boinc.berkeley.edu/trac/wiki/BoincIntro

[4] SETI@home. 2018.
URL http://setiathome.berkeley.edu/

[5] Unit testing framework. 2018.
URL https://docs.python.org/3.6/library/unittest.html

[6] Bc. PAVOL HORNICKÝ: NÁSTROJ NA TESTOVÁNÍ SÍŤOVÝCH APLIKACÍ. 2012.

[7] Harley, N.: 10 of the most costly software errors in history. 2014, [Online; navštívené 16.1.2018].
URL https://raygun.com/blog/10-costly-software-errors-history/

[8] Hranický, R.; Zobal, L.; Večeřa, V.: Distribuovaná obnova hesel. Technická zpráva, 2017.
URL http://www.fit.vutbr.cz/research/view_pub.php.cs?id=11568

[9] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas: Manifesto for Agile Software Development. 2001, [Online; navštívené 16.1.2018].
URL http://agilemanifesto.org/

[10] MMA Testing Team: Software Development Life Cycle. 2017, [Online; navštívené 16.1.2018].
URL http://www.althority.com/sdlc/

[11] Patton, R.: *Testování softwaru*. Praha : Computer Press, 2002, iSBN 80-7226-636-5.