



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

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

JURAJ CHRIPKO

VEDOUCÍ PRÁCE

SUPERVISOR

Ing. RADEK HRANICKÝ

BRNO 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 navrhnete 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

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
602 00 Brno, Božetěchova 2

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Práca si kladie za cieľ navrhnúť a implementovať automatizované testy pre systém Fitcrack, distribuovaný systém na lámanie hesiel založený na platforme BOINC. Je využité testovanie so znalosťou zdrojových kódov, konkrétne testovanie založené na požiadavkách. Na začiatku práce sú vysvetlené všeobecné praktiky testovania, potom nasleduje kapitola o testovaní založenom na požiadavkách, ktorá je základom praktickej časti. Práca ďalej obsahuje popis architektúry systému Fitcrack, návrh testov, vybrané detaily implementácie ako aj popis a výsledky testovania.

Abstract

This thesis aims to design and implement automated tests for Fitcrack, distributed password cracking system based on BOINC platform. White-box testing is used, specifically requirements-based testing. At the beginning of the thesis, general testing practices are explained, followed by principles of requirement based testing which is the basis of practical part. Thesis also includes a description of the Fitcrack architecture, tests design, selected details of the implementation, chapter about testing itself and tests results.

Kľúčové slová

Fitcrack, BOINC, testovanie, automatizované testy, jednotkové testy, integračné testy, testovanie založené na požiadavkách, Python, SQL Alchemy, unittest

Keywords

Fitcrack, BOINC, testing, automated tests, unit testing, integration testing, requirement-based testing, Python, SQL Alchemy, unittest

Citácia

CHRIPKO, Juraj. *Automatizované testování systému Fitcrack*. Brno, 2018. Bakalářská práce. 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

17. mája 2018

PodĎakovanie

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

Obsah

1	Úvod	3
2	Návrh a testovanie softvéru	5
2.1	Modely životného cyklu softvéru	5
2.2	Verifikácia a validácia	7
2.3	Metódy testovania softvéru	7
2.4	Úrovne testovania softvéru	7
2.5	Automatizácia testov	8
3	Testovanie založené na požiadavkách	9
3.1	Pojmy z testovania	9
3.2	Kritéria pokrytia	10
3.3	Testovanie založené na prechode grafu	11
3.4	Vstupné domény	14
3.5	Fixtures	17
4	Systém Fitcrack	19
4.1	Nástroje používané systémom Fitcrack	19
4.1.1	BOINC	19
4.1.2	Hashcat	20
4.2	Vlastné časti systému Fitcrack	20
4.2.1	WebAdmin	20
4.2.2	Aplikačné rozhranie	21
4.2.3	Generator	21
4.2.4	Asimilator	22
4.2.5	Runner	22
4.2.6	Komunikácia	22
5	Návrh testov pre systém Fitcrack	29
5.1	Návrh testov pre modul Generator	29
5.2	Návrh testov pre modul Asimilator	30
5.3	Návrh testov pre modul Runner	31
5.4	Návrh testov pre aplikačné rozhranie	31
6	Implementácia testovacích sád	36
6.1	Implementácia testovacej sady pre serverový modul Asimilator	37
6.2	Implementácia testovacej sady pre serverový modul Generator	39
6.3	Implementácia testovacej sady pre modul Runner	40

6.4	Implementácia testovacej sady aplikačného rozhrania (API)	42
7	Testovanie	43
7.1	Spúšťanie testov	43
7.2	Experimenty s testovacou sadou	44
7.3	Výsledky	44
8	Záver	46
	Literatúra	48
	Prílohy	49
A	Obsah CD	50

Kapitola 1

Úvod

Softvér. Pri tomto slove si každý predstaví nejakú inú aplikáciu, či už mobilnú na prezeranie a upravovanie fotografií, mzdový systém určený pre používanie na osobných počítačoch, hru pre konzoly alebo webovú stránku, na ktorej sa snažíme nájsť odpovede na tie najzávažnejšie otázky ľudstva. Dnes existuje priam až nespočetné množstvo rôznych softvérov určených na rôzne použitie na rôznych platformách, no len málokto si uvedomuje čo všetko sa skrýva za tou peknou obrazovkou, na ktorú sa každý deň pozeráme. Asi každý používateľ softvéru sa už stretol s tým, že sa softvér nechoval tak, ako by sa mal a pri tom počte nových aplikácii a rýchlosti ich vývoja to vôbec nie je nečakané. Vývoj softvéru sa za posledné roky veľmi zjednodušil a tým pádom sa na ňom môže podieľať obrovské množstvo ľudí, keďže už nemusia byť odborníci ako tomu bolo v minulosti. Firmy vyvíjajúce softvér častokrát uprednostňujú rýchlosť vývoja pred kvalitou produktu. Správne naplánovaný vývoj aplikácie dokáže aj napriek týmto faktorom zlepšiť kvalitu produktu, no to, ako správne odhadnúť náklady časové, finančné, či ľudské je náročné a na prvý pohľad zbytočné. Ďalšia možnosť, ktorá môže zlepšiť kvalitu softvéru je testovanie. V súčasnosti je testovanie považované za samozrejmu súčasť vývoja softvéru, no v praxi to vôbec nemusí byť samozrejmosť, aj keď všetky vyššie spomenuté faktory len potvrdzujú potrebu testovať.

Používanie hesiel je dnes najbežnejší spôsob ako zabezpečiť dáta, no pamätanie hesiel nie silnou ľudskou stránkou. Niekedy je takisto potrebné prelomiť heslo, ktoré zabezpečuje dáta, ktoré môžu pomôcť pri objasňovaní trestnej činnosti. Jediný spôsob ako prelomiť takéto zabezpečenie je lámanie hesiel. Lámanie hesiel funguje na jednoduchom princípe generovania a skúšania hesiel. Tento proces sa dá ľahko automatizovať, no počet možností sa môže pohybovať v miliardách a prelomenie zložitejšieho hesla by aj na výkonných počítačoch mohlo 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.

Ako každá časť vývoja softvéru, tak aj k testovaniu existuje veľké množstvo techník a odporúčaní kedy testovať, čo testovať, ako testovať. Zavedením štandardov do vývoja softvéru sa zvyšuje jeho kvalita, hlavne pri veľkých projektoch, keďže na projekte pracuje množstvo ľudí a je zložité ich koordinovať, alebo práve pri projektoch, ktoré vedú menšie a menej skúsené tímy ľudí. 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.

Fitcrack je rozsiahly systém postavený na platforme BOINC pozostávajúci z niekoľkých komponentov, ktoré stále pribúdajú a vyvíjajú sa. Táto práca si preto neberie za úlohu kompletne otestovať systém Fitcrack, ale skôr postaviť pevné základy testovania založenom na

požiadavkách a poskytnúť vývojárom systému Fitcrack návod, príklady a testy základných funkcií systému.

V druhe kapitole je preto rozobraná problematika vývoja a testovania softvéru. Tretia kapitola je venovaná požiadavkám na testy, samostatná kapitola sa venuje aj architektúre systému Fitcrack s ohľadom na dôležitosť informácii pre testovanie. Piata a šiesta kapitola popisuje návrh testovacích požiadaviek pre každý testovaný modul a ich implementáciu. V predposlednej kapitole je popis samotného testovania ako aj jeho výsledky. Práca ďalej obsahuje zhrnutie a popis práce do budúcnosti.

Kapitola 2

Návrh a testovanie softvéru

Kapitola popisuje vývoj softvéru so zameraním na testovanie. Prvá časť kapitoly približuje životný cyklus vývoja softvéru, druhá sa venuje overovaniu správnosti softvéru, tretia metódam a štvrtá úrovniám testovania. Posledná časť kapitoly stručne popisuje výhody a nevýhody automatizovaného testovania.

2.1 Modely životného cyklu softvéru

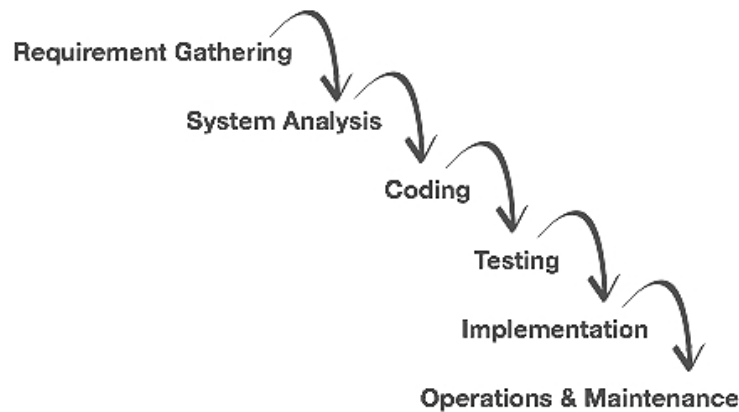
Vývoj softvéru je možné rozdeliť do niekoľkých častí, pričom každá z nich rieši iné problémy a na konci každej časti sú výsledky, ktoré je možné overiť:

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

Existujú modely, ktoré neobsahujú všetky tieto časti a existujú aj také, v ktorých sa môže jedna časť opakovať niekoľko krát.

Vodopádový model

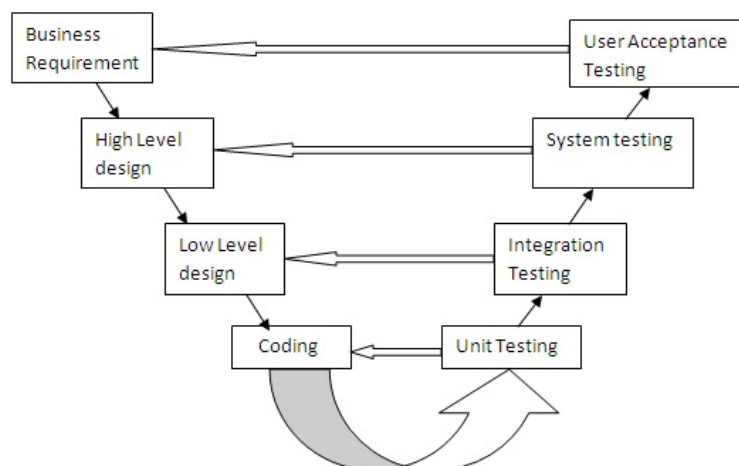
Prvý formálne popísaný a asi najznámejší model životného cyklu softvéru [2.1](#). Bol prezentovaný už v roku 1970 a vyjadruje postupnosť fáz: získavanie požiadaviek, systémová analýza, programovanie, testovanie, implementácia a údržba systému. Je často používaný pri výučbe, pretože je jednoduchý a ľahko sa vysvetľuje. Definuje a plánuje fázy, čím pomáha dodržiavať termíny a výstupy jednotlivých fáz, ale je veľmi málo flexibilný, keďže počíta stým, že požiadavky sa nebudú meniť, takže plánovanie všetkých úloh projektu je potrebné naplánovať do posledného detailu, čo je náročné a drahé [\[8\]](#).



Obr. 2.1: Vodopádový model [5].

V model

Podobný vodopádovému modelu 2.1, no každá fáza pred programovaním má k sebe doplnkovú fázu, ktorá overuje výsledky fázy pred programovaním 2.2. Napríklad, keď sú stanovené požiadavky na softvér, tak môžu byť hneď napísané akceptačné testy 2.4. No reálne overenie môže nastať až keď je softvér implementovaný. V tomto modeli je použitých niekoľko testovacích techník, pričom každá z nich overuje inú časť návrhu, designu, alebo implementácie [8].



Obr. 2.2: V model [5].

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 vzniknúť z rôznych príčin, no všeobecne platí, že čím skôr sa chyba odhalí, tým ľahšie a teda aj lacnejšie je opravitelná. Testovanie je súčasťou verifikácie systému, ktorá je zase súčasťou procesu zabezpečovania kvality softvéru (Quality Assurance, QA). [2]

2.2 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 [11]. **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 metód overuje, že systém odpovedá špecifikácii.
- **Testovanie** skúma softvér spúšťaním programu za účelom zvýšenia jeho kvality. Patrí pod dynamickú analýzu.

Validácia potvrdzuje, že softvér spĺňa obchodné podmienky a požiadavky používateľa, nie technickú špecifikáciu [11].

2.3 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 nepodieľala sa na vývoji softvéru. Táto kapitola ďalej rozoberá rozdiely v testovaní pri rozdielnych znalostiach systému.

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 [7]. Má informácie o úlohe softvéru, ale nie o spôsobe vykonávania 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 odhaľovaní 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 [7]. K takémuto testovaniu 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 voľiť napríklad také testovacie prípady, aby boli pri testovaní použité všetky vetvy kódu.

Existuje prístup kombinujúci testovanie čiernej a bielej skrinky, **testovanie šedej skrinky**. Tester pozná vnútorné dátové štruktúry a algoritmy, ale nepozná reálnu implementáciu, 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.

2.4 Úrovne testovania softvéru

Ako ukazujú modely životného cyklu softvéru 2.1, rôzne časti (úrovne) systému je možné testovať pri rôznych príležitostiach. Nedá sa testovať stále a všetko, je potrebné určiť kedy budú ktoré časti testované.

Jednotkové testy

Jednotkové testy (unit tests) pracujú iba s jednou jednotkou (komponentom, modulom), ktorú testujú [7]. Preto sa im niekedy hovorí aj testovanie komponentov (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.

Integračné testy

Integračné testy overujú funkčnosť rozhrania medzi komponentmi. Môžu byť malé ako pri jednotkových testoch a postupne s vývojom softvéru sa zväčšovať až sa testuje integrácia celého podsystemu do už používaného systému [7]. Tento prístup sa nazýva inkrementálne integrovanie a chyby dokáže odhaliť pomerne skoro. Keďže sa musia simulovať rozhrania, ktoré ešte neboli implementované, je tento prístup drahý. Opačný prístup- integrácia veľkým treskom sa testuje až keď sú všetky súčasti systému implementované. Ide o rýchly a jednoduchý prístup, no je náročné zistiť príčinu zlyhania takejto neskorej integrácie. Ideálny prístup tkvie niekde medzi týmito dvoma extrémnymi prípadmi. Vhodnejšie je testovať väčšie súčasti, ktoré by mohli pri integrácii spôsobovať problémy.

Systémové testy

Systémové testy sú robené na takmer hotovom softvéri a je u nich vyžadovaná nezávislosť. Robia ich viaceré tímy alebo tretia strana. Cieľom systémových testov je overiť, či softvér spĺňa určitú špecifikáciu.

Akceptačné testy

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

Regresné testovanie

Regresné testy sa prevádzajú pri každej zmene nejakej časti softvéru. Zisťuje sa, či nová verzia zdieľa vlastnosti starej a či sa do systému nezanesli nové chyby. Regresné testovanie prebieha často, takže regresné testy sú veľa krát automatizované.

2.5 Automatizácia testov

Testovanie je časovo a finančne náročná činnosť, preto je dobré ho čo najviac automatizovať. Ak by sa mal napríklad manuálne testovať celý systém po každom vydaní novej verzie, ktorá len opravuje chyby starších verzií, bola by to sisyfovská práca. Nie všetko testovanie sa dá automatizovať, čo môže byť aj dobre. Ak automatické testy minú chybu pri prvom spustení, tak tú chybu budú mýňať pri každom ďalšom testovaní. Chyba sa stane imúnna voči testom.

Kapitola 3

Testovanie založené na požiadavkách

Alebo aj **requirement-based testing**¹ je testovanie v ktorom sú testovacie prípady, dáta a podmienky založené na požiadavkách, takže overuje, že testovaný softvér odpovedá požiadavkám [11].

Fázy testovania založeného na požiadavkách:

- definícia kritérií pokrytia,
- návrh testovacích prípadov,
- spustenie testov,
- overenie výsledkov testov,
- zistenie pokrytia testovacej sady,
- zaznamenanie nájdených chýb.

Táto kapitola predstavuje rôzne požiadavky na testy a kritéria pokrytia. Návrh testovacích prípadov je priamo závislý na zvolenom kritériu pokrytia – ciele testovania.

3.1 Pojmy z testovania

Predtým ako budú vysvetlené princípy testovania založeného na požiadavkách je potrebné definovať pojmy, ktoré budú v práci používané.

Test case (testovací prípad)

Testovací prípad sa skladá zo vstupných hodnôt, očakávaných výsledkov a tiež z prefixových a postfixových hodnôt (pre prípravu programu pred testom a vyčistení systému po teste) [7].

Test set / test suite (testovacia sada)

Testovacia sada je množina testovacích prípadov.

¹https://www.tutorialspoint.com/software_testing_dictionary/requirements_based_testing.htm

Test requirement (požiadavka na test)

Požiadavka na test je špecifický element softvéru. Musí daný test pokryť, alebo splniť, môže to byť:

- riadok kódu,
- vetvenie kódu,
- konkrétna trieda,
- konkrétna správa posiadaná objektom,
- ...

Softvérový artefakt

Softvérový artefakt je subjekt programátora pomocou ktorého tvorí výsledný produkt. Softvérové artefakty môžu byť [11]:

- časti kódu (moduly, triedy, metódy, funkcie),
- riadky kódu (príkaz, volaná funkcia, usporiadanie parametrov),
- riadiaca logika (cykly, podmienky, logické výrazy, podmienky),
- práca s dátami (premenné, dátové typy, konštanty, operátory, zmena typu),
- logika behu (usporiadanie sekvencií volaní, atomicita operácií, synchronizačné primitíva),
- predpoklady prostredia (výkon stroja, verzia knižníc, premenné prostredie, konfigurácia OS, sieťové pripojenie a jeho kvalita),
- kombinácia softvérových artefaktov,
- ...

System under test (SUT)

Testovaný systém, alebo iná entita (modul, trieda, ...), ktorá je testovaná.

Coverage (pokrytie)

Pokrytie je miera udávajúca ako veľmi daná testovacia sada skúma testovaný systém (SUT). Pri testovaní softvéru sa používa **code coverage**. Typicky sa udáva v percentách, vzťahuje sa k danému kritériu pokrytia a k danej testovacej sade [7].

Coverage criterion (kritérium pokrytia)

Kritérium pokrytia je pravidlo, alebo predpis pre systematické generovanie požiadaviek na testy.

3.2 Kritéria pokrytia

Testovacia sada by sa mala tvoriť na základe predom zvoleného kritéria s cieľom pokryť ho na 100 %. Kritérium pokrytia definujú požiadavky na testy, **test requirements** (TR). Množina požiadaviek na testy daného kritéria pokrytia môže byť väčšia než testovacia sada, ktorá spĺňa dané pokrytie. Jeden testovací prípad môže uspokojiť/pokryť niekoľko testovacích požiadaviek [10]. Existujú rôzne kritéria pokrytia zaoberajúce sa rôznymi artefaktmi. Sú rôzne silné.

Definícia. Kritérium C_1 zahŕňa kritérium C_2 ($C_1 > C_2$), práve vtedy, keď každá testovacia sada spĺňajúca kritérium C_1 tiež spĺňa kritérium C_2 .

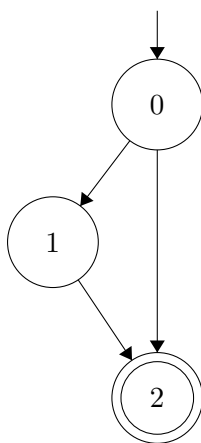
3.3 Testovanie založené na prechode grafu

Graf je jedna z najpoužívanějších štruktúr pre abstrakciu. Je dobre definovaná, veľmi názorná a je jednoduché ňom niečo jasne vyjadriť. V testovaní poznáme niekoľko typov grafov, každý z nich testuje systém iným spôsobom [4]:

- **Control flow graph (CFG)** – graf toku riadenia: zachytáva informácie o tom, ako je v programe predávané riadenie.
- **Data flow graph** – graf toku dát: modifikuje CFG pridaním informácií o dátových tokoch.
- **Dependency graph** – graf súvislostí: zachytáva dátové súvislosti, alebo súvislosti riadenia programu medzi príkazmi programu.
- **Cause-effect graph** – graf príčin a následkov: modeluje vzťahy medzi vstupnými podmienkami – príčinami (causes) s výstupnými podmienkami – následkami (effects).

Každý tento graf testuje funkčnosť SUT iným spôsobom. V tejto práci je ďalej vysvetlený graf toku riadenia a graf toku dát. Postup pri testovaní založenom na prechode grafu je pri všetkých rovnaký [4]:

1. Vytvorenie modelu grafu:
Aké informácie bude graf obsahovať a ako sa budú reprezentovať.
2. Zvoliť požiadavky na testy:
Aké požiadavky musia byť splnené.
3. Výber ciest grafu, ktoré pokryjú zvolené požiadavky.
4. Odvodenie dát, aby mohli byť vybrané cesty spustené.



Obr. 3.1: Príklad jednoduchého grafu.

Pojmy z testovania založenom na prechode grafu

Tak ako pri pojmoch z testovania 3.1 aj tu je potrebné definovať pojmy, ktoré budú použité v nasledujúcich kapitolách práce [11]:

Cesta v grafe ($path \subseteq N^+$) je neprázdná sekvencia uzlov $[n_1, n_2, \dots, n_M]$, taká, že každá dvojica susedných uzlov tvorí hranu v danom grafe:

$$(n_i, n_{i+1}) \in E, 1 \leq i \leq M \quad (3.1)$$

Dĺžka cesty je daná počtom hrán.

Testovacia cesta začína niektorým počiatočným uzlom a končí v niektorom z koncových uzlov.

Podcesta cesty p je súvislá časť retazca daných uzlov.

Stopa (trace) je taká cesta CFG, ktorá je realizovateľná testovaným subjektom.

Beh programu (run) je stopa začínajúca v počiatočnom uzle a končiaca v koncovom uzle, pričom CFG odpovedá celému programu.

Uzol n je **syntaktický dosiahnuteľný (reachable)** z uzlu n_i , ak existuje cesta z uzlu n do uzlu n_i .

Uzol je **sémanticky dosiahnuteľný** z uzlu n_i , ak existuje nejaká stopa z uzlu n do uzlu n_i .

Jednoduchá cesta (simple path) je taká cesta, v ktorej sa žiadny uzol neopakuje. Výnimkou je cesta, ktorá začína aj končí v rovnakom uzle.

Hlavná/primárna cesta (prime path) je taká jednoduchá cesta, ktorá nie je podcestou inej jednoduchej cesty, to znamená, že to je najdlhšia jednoduchá cesta.

Path (T) pre testovaciu sadu T je množina ciest, respektíve stôp, ktoré sú spúšťané testovacími prípadmi z T .

Control flow graph

(CFG) Graf toku riadenia (CFG) je orientovaný graf, v ktorom uzly reprezentujú základné bloky (basic blocks) a hrany reprezentujú cesty toku riadenia [11].

Definícia. Control flow graph (CFG) je štvorica $G = (N, N_0, N_f, E)$, kde:

- N je konečná množina uzlov,
- $N_0 \subseteq N, N_0 \neq \emptyset$, je neprázdná množina počiatočných uzlov,
- $N_f \subseteq N$ je množina koncových uzlov,
- $E \subseteq N \times N$ je množina hrán.

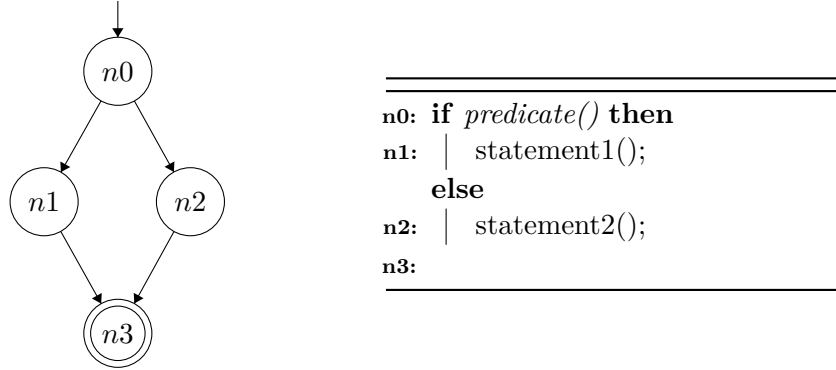
Pozn. Existujú rozšírené definície (napríklad s ohodnotením hrán).

Základný blok (basic block) [10] je postupnosť maximálneho počtu príkazov, pre ktoré platí, že:

- vstupný bod (riadenia) je na prvom príkaze,
- výstupný bod je na poslednom príkaze

- príkazy sa vykonávajú vždy sekvenčne v poradí danom postupnosťou.

Základný blok môže obsahovať príkaz pre vetvenie iba na konci. Každý z uzlov n_0 až n_3 na obrázku 3.2 je základný blok.



Obr. 3.2: Jednoduchý control flow graf.

Kritéria pokrytia riadiacich tokov

Požiadavky na testy v CFG môžeme vyjadriť ako požiadavku na prechod hranou, alebo navštívenie vybraných uzlov – obidve požiadavky zahŕňajú cestu [7]:

Kritérium pokrytia uzlov, **Node Coverage (NC)** vyžaduje, aby požiadavky na test obsahovali každý syntaktický dosiahnuteľný uzol.

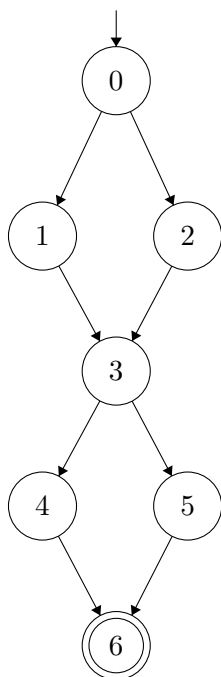
Kritérium pokrytia hrán, **Edge Coverage (EC)** vyžaduje, aby požiadavky na test obsahovali každú syntakticky dosiahnuteľnú cestu o dĺžke 0, alebo 1².

Kritérium pokrytia párov hrán, **Edge-Pair Coverage (EPC)** vyžaduje, aby požiadavky na test obsahovali každú syntakticky dosiahnuteľnú cestu o dĺžke najviac 2.

Kritérium pokrytia hlavných ciest, **prime Path Coverage (PPC)** vyžaduje, aby požiadavky na testy obsahovali všetky hlavné cesty. Príklad je na obrázku 3.4.

Porovnanie kritérií pokrytia riadiacich tokov je na obrázku 3.3.

²Požiadavka na cestu dĺžky 0 z uzlu n odpovedá požiadavky na prechod uzlom n – špeciálny prípad grafu bez hrán.



$$path(t_1) = [0, 1, 3, 4, 6]$$

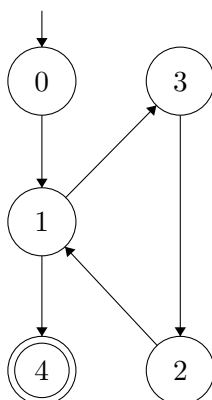
$$path(t_2) = [0, 2, 3, 5, 6]$$

$$path(t_3) = [0, 1, 3, 5, 6]$$

$$path(t_4) = [0, 2, 3, 4, 6]$$

Testovacia sada	NC	EP	EPC
$T_1 = \{t_1\}$	×	×	×
$T_2 = \{t_2\}$	×	×	×
$T_3 = \{t_1, t_2\}$	✓	✓	×
$T_4 = \{t_1, t_2, t_3, t_4\}$	✓	✓	✓

Obr. 3.3: Porovnanie kritérií pokrytia.



$$PP = \{[0, 1, 2], [0, 1, 3, 4], [1, 3, 4, 1], [3, 4, 1, 3], [3, 4, 1, 2]\}$$

$$path(t_1) = [0, 1, 2]$$

$$path(t_2) = [0, 1, 3, 4, 1, 3, 4, 1, 2]$$

$$T_2 = \{t_1, t_2\} \text{ splňuje PPC.}$$

Obr. 3.4: Príklad pokrytia primárnych ciest (PPC).

3.4 Vstupné domény

Vstupná doména (doména vstupu, priestor, input space) je množina všetkých vstupov systému [7]. Môžu to byť:

- vstupné parametre metód,

	b_1	b_2	b_3	b_4	b_5
H	$x < 0$	$x > 0$	$x = 0$	$\text{isinf}(x)$	$\text{isnan}(x)$

Tabuľka 3.1: Príklad rozkladu jednorozmernej domény.

- globálne, alebo statické premenné,
- vstupy od užívateľa, argumenty programu,
- objekty reprezentujúce aktuálny stav systému.

Vstupná doména je rozdelená na menšie celky (bloky, regióny, časti)³, ktoré obsahujú rovnako užitočné hodnoty z pohľadu testovania.

Niektoré behy programu je možné zoskupiť do rovnakej kategórie – pre rôzne vstupy sa program chová rovnako. Cieľom je otestovať všetky takéto kategórie, pričom z každej kategórie stačí vybrať iba jednu hodnotu.

Rozklad vstupných domén

Každá správna charakteristika definuje rozklad domény (q), ktorý musí spĺňať nasledujúce podmienky:

- Jednotlivé bloky rozkladu (B_q) musia byť vzájomne disjunktné, nesmú sa prekrývať:

$$b_i \cap b_j = \emptyset, i \neq j, b_i, b_j \in B_q$$

- Všetky bloky dohromady musia byť kompletne, rozklad musí zahŕňať celú doménu:

$$\bigcup_{b \in B_q} b = D$$

Model vstupných domén

Model vstupnej domény (input domain model) je abstrakcia vstupov testovaného systému. Tester popisuje štruktúru modelu vstupných domén pomocou charakteristík. Pre každú charakteristiku rozkladá domény na jednotlivé bloky. Každý blok predstavuje množinu hodnôt vstupov SUT. Tieto hodnoty sú z hľadiska testov považované za rovnocenné.

Kritéria pokrytia vstupných domén

Ako príklad k rôznym kritériám budú použité 3 rozklady s blokmi:

$$q_1 = \{A, B\}, \quad q_2 = \{1, 2, 3\}, \quad q_3 = \{x, y\} \quad (3.2)$$

- Kritérium pokrytia všetkých kombinácií **all combinations coverage (ACoC)** vyžaduje, aby TR obsahovali všetky kombinácie blokov zo všetkých charakteristík. Na príklad: 3 rozklady s blokmi: TR bude obsahovať týchto 12 požiadaviek:

$(A, 1, x)$	$(A, 1, y)$	$(B, 1, x)$	$(B, 1, y)$
$(A, 2, x)$	$(A, 2, y)$	$(B, 2, x)$	$(B, 2, y)$
$(A, 3, x)$	$(A, 3, y)$	$(B, 3, x)$	$(B, 3, y)$

³V matematike je to rozklad na ekvivalentné triedy

- Kritérium pokrytia každého bloku **Each Choice Coverage (ECC)** vyžaduje, aby TR obsahovali každý blok pre každú charakteristiku. Testovacia sada $T = \{t_1, t_2, t_3\}$:

$$t_1 = (A, 1, x)$$

$$t_2 = (B, 2, y)$$

$$t_3 = (B, 3, y)$$

- Kritérium pokrytia všetkých párov blokov **Pair-Wise Coverage (PWC)** vyžaduje, aby TR obsahovali každý blok každej charakteristiky a každý pár blokov každý z inej charakteristiky. TR bude obsahovať týchto 16 požiadaviek:

$(A, 1)$	$(B, 1)$	$(1, x)$	$(1, y)$
$(A, 2)$	$(B, 2)$	$(2, x)$	$(2, y)$
$(A, 3)$	$(B, 3)$	$(3, x)$	$(3, y)$
(A, x)	(A, y)	(B, x)	(B, y)

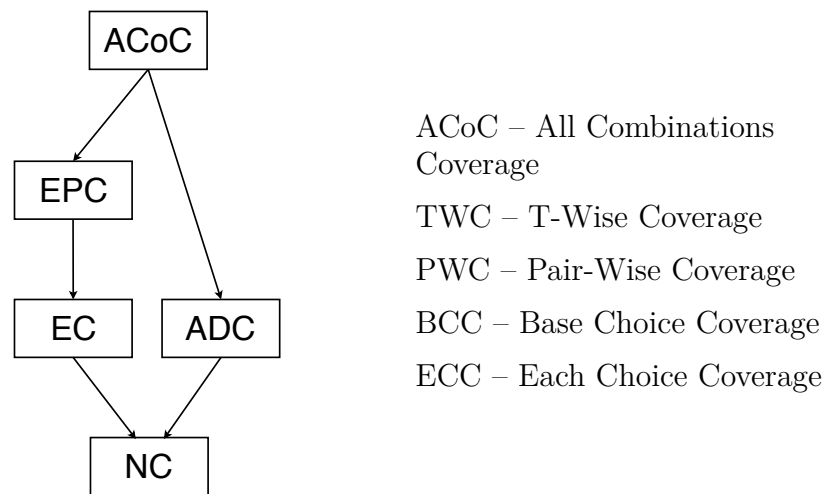
Stačí testovacia sada:

$$\begin{array}{llll} t_1 : (A, 1, x) & t_2 : (A, 2, x) & t_3 : (A, 3, x) & t_4 : (A, -, y) \\ t_5 : (B, 1, y) & t_6 : (A, 2, y) & t_7 : (B, 3, y) & t_8 : (B, -, x) \end{array}$$

- PWC je možné zobecniť na **T-Wise coverage** (nie dvojice, aby T-tice blokov).
- Kritérium pokrytia základných blokov **Base Choice Coverage (BCC)** vyžaduje, aby TR obsahovali kombinácie všetkých základných blokov každej charakteristiky a každý jeden iný ako základný blok v kombinácii s základnými blokmi. Bázové bloky (typické bloky): $A, 1, x$

TR bude obsahovať požiadavku: $(A, 1, x)$ a nasledujúce:

$$(B, 1, x), \quad (A, 2, x), \quad (A, 3, x), \quad (A, 1, y) \tag{3.3}$$



Obr. 3.5: Vzťah kritérií pokrytia vstupných domén.

3.5 Fixtures

Test fixture podľa vzoru XUnit⁴ je množina predpokladov alebo stavov potrebných na spustenie testu. Vývojár testov by mal pripraviť tieto podmienky pred spustením testov a po testoch by mal SUT vrátiť do pôvodného stavu [1].

XUnit takisto definuje fázy testov:

- Setup (príprava prostredia pre test),
- Exercise (spustenie SUT),
- Verify (overenie výsledkov),
- Teardown (vrátenie SUT do pôvodného stavu).

Test fixture sú teda všetky vstupné dáta pre SUT, aby bolo možné spustiť test opakovane a bez ohľadu na aktuálny kontext [10].

Test Double (dvojník) je objekt, ktorý má reprezentovať pôvodný objekt na ktorom je SUT, aj test, závislý. Test double musí mať rovnaké rozhranie ako objekt, ktorý nahrádza. Ak to programovací jazyk dovoľuje, stačí aj podmnožina rozhrania využívaná SUT počas testu. Dvojník by nemal mať ďalšie závislosti, okrem tých, ktoré sú využívané SUT. Podľa schopností existuje niekoľko typov dvojníkov [11]:

- Dummy – existuje, ale nič nerobí,
- Stub – vracia prednastavenú hodnotu,
- Spy – sleduje a zaznamenáva,

⁴<http://xunitpatterns.com/>

- Mock – sleduje, overuje, vracia,
- Shims – upravuje skompilovaný kód za behu (run-time), aby injektoval s spúšťal metódy substitúcie (tzv. detour),
- Fakes – takmer plnohodnotné náhrada (napríklad databáza v pamäti).

V testovacej komunite sú najpoužívanéjšie Stub a Mock objekty:

Stub

- Nezáleží na vstupoch, stále vráti preddefinovanú hodnotu (objekt).
- Nemôže spôsobiť zlyhanie testu.
- Vzor **State Verification**⁵ – overuje sa výsledný stav SUT.

Mock

- Kontroluje vstupné hodnoty. Výstupné hodnoty posiela v závislosti na vstupných hodnotách.
- Môže spôsobiť zlyhanie testu.
- Vzor **Behavior Verification**⁶ – overuje sa aj vnútorné chovanie SUT.

⁵<http://xunitpatterns.com/State%20Verification.html>

⁶<http://xunitpatterns.com/Behavior%20Verification.html>

Kapitola 4

System Fitcrack

Fitcrack [3] je distribuovaný systém pre obnovu hesiel šifrovaných médií a prelomenie kryptografických hashov. Pre samotné lámanie hashov používa Hashcat, ktorý je momentálne najrýchlejšie riešenie na trhu. Správu hostov a rozdeľovanie úloh zabezpečuje Berkeley Open Infrastructure framework (BOINC). Viac o týchto nástrojoch obsahuje nasledujúca kapitola. Vlastné súčasti systému Fitcrack sú ďalej popísané v kapitole 4.2.

4.1 Nástroje používané systémom Fitcrack

Táto kapitola popisuje nástroje používané systémom Fitcrack. Pre testovanie je hlavné dôležité poznať ich funkciu v systéme Fitcrack, nie podrobnosti ich fungovania.

4.1.1 BOINC

Berkeley Open Infrastructure for Network Computing (BOINC)¹ 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 *Search for Extraterrestrial Intelligence* (SETI)² 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 pridelenie pracovných úloh klientom, zaisťuje stahovanie 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 dis-

¹<http://setiathome.berkeley.edu/>

²<https://setiathome.berkeley.edu/>

tribuoovaný výpočet cez internet ponúka množstvo bezpečnostných mechanizmov na prácu v nedôveryhodnom prostredí.

4.1.2 Hashcat

Hashcat³ je svetovo najrýchlejším riešením na lámanie hesiel na jednom stroji. Je zadarmo, má otvorený zdrojový kód, je spustiteľný na operačnom systéme Windows, Linux aj macOS. Používa OpenCL, ktoré je už dnes kompatibilné s väčšinou zariadení (CPU, GPU, DSP, FPGA, ...) a podporuje množstvo formátov.

OpenCL⁴ (Open Computing Language) je štandard pre paralelné programovanie na rôznych typoch procesorov používaných v osobných počítačoch, serveroch, mobilných telefónoch a vstavaných systémoch.

Generovanie hesiel

Pri lámaní hesiel má okrem výkonu veľký význam aj poradie generovania hesiel. Väčšina ľudí si negeneruje náhodné heslá, takže dopredu pripravený slovník obsahujúci často používané heslá dokáže zrýchliť výpočet. Hashcat podporuje rôzne druhy útokov, no Fiterack momentálne využíva nasledujúce:

Útok, generujúci heslá po znakoch sa na nazýva **brute/mask**, keďže používa hrubú silu a masky. *Maska* špecifikuje znaky a ich pozíciu v hesle, napríklad: `?l?d` – maska o hovorí, že heslo má práve 2 znaky, prvý znak môže byť ľubovoľné malé písmeno (`?l`) a druhý ľubovoľné číslo (`?d`).

Slovníkový (dictionary) útok skúša heslá zo zadaného slovníka – zoznamu hesiel, ku ktorému môže byť špecifikované pravidlo. Napríklad: pridaj na koniec každého hesla znaky 123.

Kombinačný (combinator) útok kombinuje heslá z dvoch slovníkov, no ešte predtým môže na každý slovník aplikovať iné pravidlo.

4.2 Vlastné časti systému Fiterack

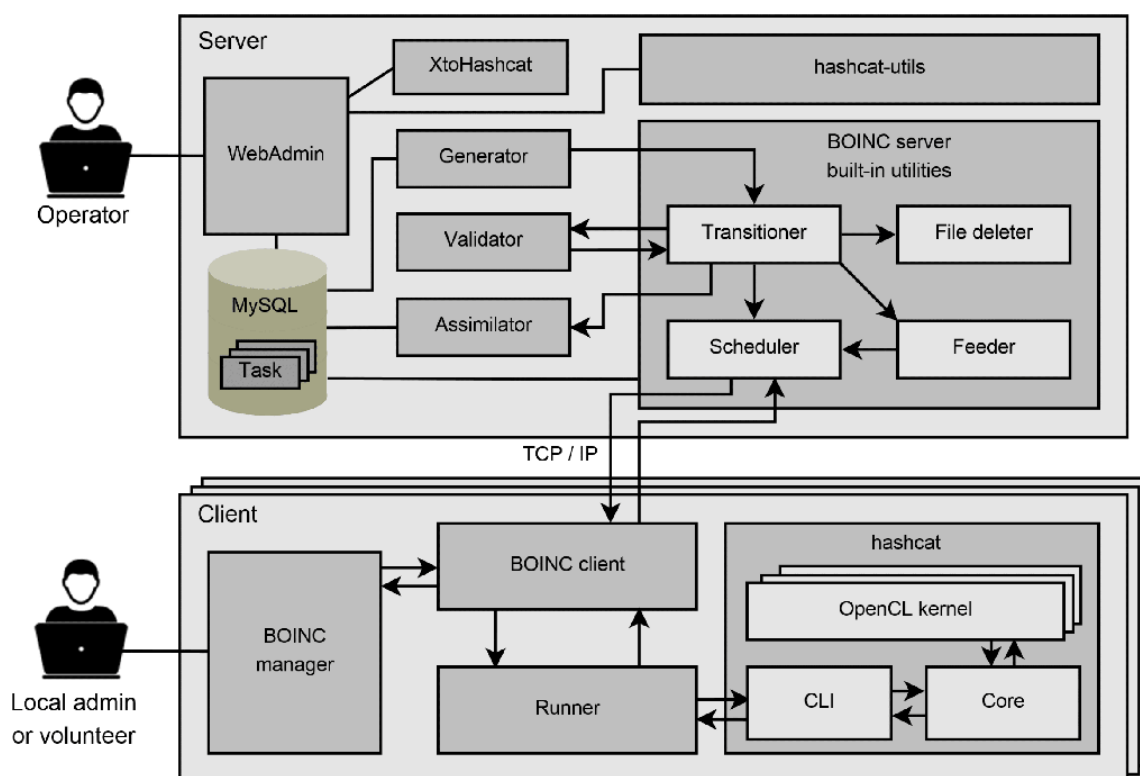
Architektúru systému Fiterack znázorňuje obrázok 4.1. **BOINC server**, **BOINC client**, **BOINC manager** a **Validator** sú súčasťou platformy BOINC a neboli nijak modifikované. **Hashcat-utils** je súbor nástrojov uľahčujúci prácu s programom Hashcat. **XtoHashcat** je skript na extrakciu hashu zo súboru. Podkapitoly obsahujú bližší popis častí ktoré budú testované: **WebAdmin**, **Generator**, **Asimilator** a **Runner**

4.2.1 WebAdmin

WebAdmin je grafické užívateľské rozhranie pre ovládanie systému Fiterack. Paralelne s touto prácou je vyvíjaná aj nová verzia, ktorá bude rozdelená na dve časti: grafické užívateľské (GUI) rozhranie a aplikačné rozhranie (API). Súčasťou tejto práce je aj testovanie časti aplikačného rozhrania.

³<https://hashcat.net/hashcat/>

⁴<https://www.khronos.org/opencl/>



Obr. 4.1: Architektúra systému Fitcrack [3].

4.2.2 Aplikačné rozhranie

Vyššie spomenuté **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 pomocou databázy. Toto API je schopné vytvárať nové úlohy, sledovať pripojených klientov, mapovať ich na existujúce úlohy, riadiť výpočet úloh, ... [6].

4.2.3 Generator

Generator, 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). Generator vytvára 2 typy úloh: **benchmark** a **normal**. Úlohy typu **benchmark** sa generujú klientom, ktorý sa práve pripojil, alebo keď výpočet na danom klientovi skončil chybou. Úlohy typu **normal** sú pracovné úlohy obsahujúce všetky potrebné informácie pre spustenie klientskej časti systému Fitcrack. Úloha môže byť v jednom z nasledujúcich stavov:

- **0 - ready.** Výpočet neprebíha.
- **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 moduli Generator je implementovaný plánovací algoritmus, ktorý vytvára pracovné úlohy na mieru konkrétnej klientskej stanice. Princíp činnosti modulu Generator je popísaný algoritmom 1.

4.2.4 Asimilator

Asimilator je takisto BOINC server démon, ktorý spracováva validné výsledky. Nekonečný cyklus v ktorom beží je poskytnutý platformou BOINC, ale samotná akcia, ktorá sa vykonáva pri prijatí výsledku je implementovaná tvorcami systému Fitcrack. Funkčnosť akcie je prezentovaná algoritmom 2.

4.2.5 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, ktorý sleduje stav jeho výpočtu a po skončení predá výsledky BOINC klientovi. Ten ich následne posiela na server.

Generator a Asimilator komunikujú výhradne pomocou databázy. Zatiaľ čo Validator a Runner riadi priamo platforma BOINC. Okrem BOINC klienta nie je potrebné na klientskom zariadení nič inštalovať, BOINC klient sám zo servera sťahuje najnovšie spustiteľné súbory v závislosti na architektúre klientského zariadenia.

4.2.6 Komunikácia

Fitcrack používa na komunikáciu s modulom Runner 2 typy súborov:

- **TLV slovník**, pre komunikáciu server → klient a
- **runner output** súbor pre komunikáciu smerom klient → server.

TLV slovník⁵ používa Generator, aby predal informácie modulu Runner ohľadom konkrétnej úlohy a **runner output** súbor obsahuje výsledky takejto úlohy, ktoré spracúva Asimilator.

Komunikácia server → klient

Pri vytvorení nového balíčka sa do databázi uloží konfiguračný súbor vo formáte TLV (type, length, value) slovníka, ktorý nesie základné informácie ako je napríklad typ útoku alebo typ hashu. Pri generovaní úloh Generator tento slovník dopĺňa o špecifické informácie o konkrétnej úlohe a posiela ho klientovi. Spolu s TLV slovníkom musia byť klientovi poslané ďalšie súbory obsahujúce hash, prípadne zoznamy hesiel na skúšanie – slovníky. Nasledujúci zoznam popisuje aké súbory sa posielajú pri rôznych typoch útokov:

⁵<http://en.citizendium.org/wiki/Type-Length-Value>

Algoritmus 1: Princíp činnosti systémového modulu Generator.

```
while 1 do
    /* Inicializácia */
0   zmaž uzly, ktoré sa podieľajú na riešení úlohy, ktorých pracovný balíček je už
    dokončený, buď úspešne (finished) alebo neúspešne (exhausted)
    if niektorý z balíčkov presiahol stanovenú dobu ukončenia then
1       | nastav jeho stav na finished (12)
2   foreach bežiaci pracovný balíček (stav  $\geq 10$ ) do
3       | if nie je nastavený čas zahájenia then
4           | nastáva čas zahájenia na aktuálny čas
5       | if k balíčku sa viažu masky hesiel then
6           | ulož ich do príslušného poľa, ktoré odpovedá balíčku
7       najdi uzly, ktoré sa majú podieľať na výpočte (a zatiaľ sa nepodieľajú) a
        ulož ich do databázy
    /* Benchmark */
8   foreach pridelený aktívny uzol, ktorý ma stav Benchmark (0) do
9       | if uzol ešte nemá naplánovaný benchmark then
10          | naplánuj benchmark pre tento uzol
    /* Výpočet */
11  foreach aktívny uzol v stave Normal (1) do
12      | if počet naplánovaných úloh pre uzol  $\geq 2$  then
13          | pokračuj na ďalší uzol
14      | if Stav uzlu je Running (10) then
15          | vygeneruj novú úlohu podľa typu balíčka, prípadne znovu pridel
            nedokončené úlohy
16      | if stav uzle je Finished (12) then
17          | znovu pridel nedokončené úlohy, ak taká neexistuje, nastav stav uzlu
            na Done (3)
    /* Kontrola stavu */
18  if stav balíčka je Finished (12) a neobsahuje žiadne úlohy then
19      | if aktuálny čas > plánovaný čas ukončenia then
20          | nastav stav balíčka na Timeout (4)
        else
21          | if aktuálny index  $\geq$  maximálny index then
22              | nastav stav balíčku na Exhausted (2)
              /* vyčerpaný stavový priestor */
        else
23          | nastav stav balíčku na Ready(0)
          /* výpočet bol pozastavený */
24  čakaj stanovený časový interval
```

benchmark :

- **config** – TLV slovník obsahující najmä typ hashu.

mask attack :

- **config** – TLV slovník,
- **data** – obsahuje hash.

slovníkový útok :

- **config** – TLV slovník,
- **data** – obsahuje hash,
- **dict1** – fragment slovníka, ktorý musí byť celý vyskúšaný.

kombinačný útok :

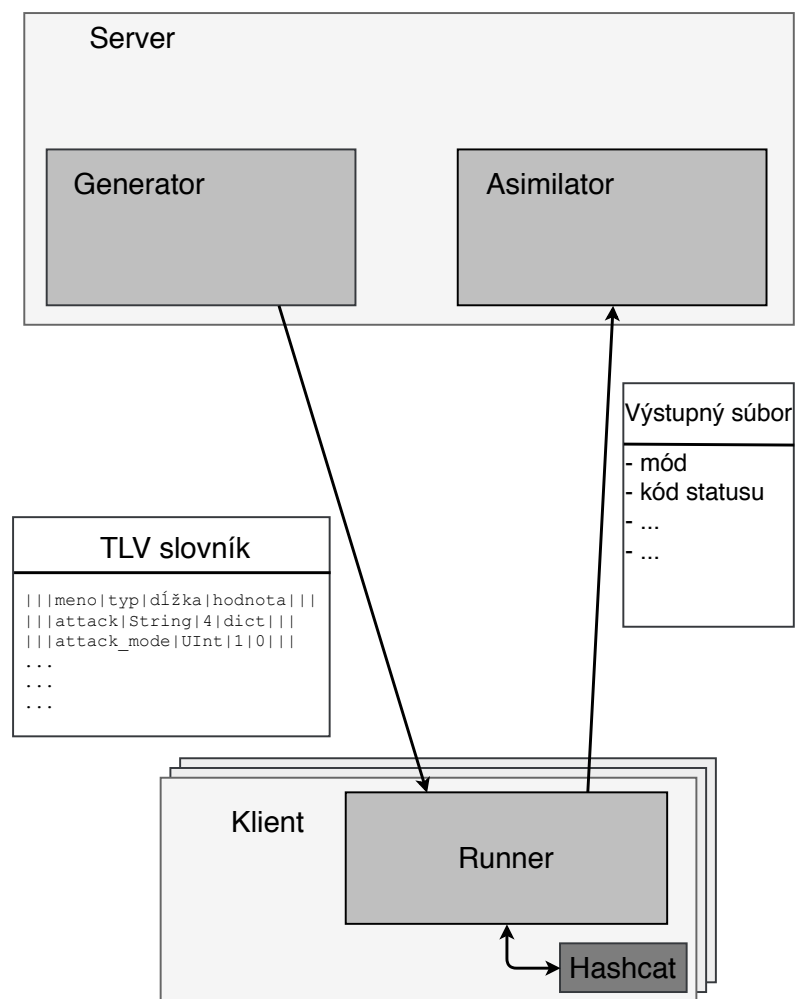
- **config** – TLV slovník obsahujúci najmä typ hashu,
- **data** – obsahuje hash,
- **dict1** – prvý (ľavý) slovník,
- **dict2** – druhý (pravý) slovník.

Formát TLV slovníka je nasledovný:

|||meno|typ|dĺžka|hodnota|||

- **meno** – identifikátor záznamu,
- **typ** – typ položky hodnota,
- **dĺžka** – počet znakov položky hodnota,
- **hodnota** – samotná hodnota záznamu.

Slovník môže obsahovať rôzne typy záznamov, používané sú v tabuľke [4.1](#)



Obr. 4.2: Schéma komunikácie.

Komunikácia klient → server

Klient vytvára výstupný súbor s informáciami o výsledku danej úlohy. Súbor obsahuje na jednom riadku práve jednu informáciu, tá sa podľa typu úlohy a výsledku môže líšiť, no prvé 2 riadky sú stále rovnaké:

```
<mode>
<status_code>
```

Nasledujúci zoznam popisuje mód, možné návratové kódy a špecifikuje ďalšie informácie:

- **Benchmark**
 - <mode>: b
 - Status code môže mať hodnoty:

- **0**: úloha benchmark prebehla v poriadku. Formát výstupného súboru⁶:

```
b
0
<cracking_speed (power)> - integer
<cracking_time> - double
```

- **4**: úloha typu benchmark skončila chybou. Formát výstupného súboru⁷:

```
b
4
<hashcat_exit_code> - integer
<hashcat_exit_info> - string
```

- **Normal**

<mode>: n

Status code môže mať hodnoty:

- **0**: bolo nájdené heslo. Formát výstupného súboru⁸:

```
n
0
<password (base64 encoded)> - string
<cracking_time> - double
```

- **1**: heslo nebolo nájdené. Formát výstupného súboru:

```
n
1
<cracking_time> - double
```

- **4**: pri výpočte úlohy nastala chyba. Formát výstupného súboru:

```
n
4
<hashcat_exit_code> - integer
<hashcat_exit_info> - string
```

⁶cracking_speed je počet vyskúšaných hashov za sekundu a cracking_time je počet sekúnd trvania úlohy.

⁷hashcat_exit_code je návratový kód nástroja Hashcat a hashcat_exit_info je chybová hláška, ktorú vypíše Hashcat.

⁸password je heslo zakódované vo formáte base64 a cracking_time je počet sekúnd, ktoré boli potrebné na prelomenie hesla.

Algoritmus 2: Princíp činnosti modulu Asimilator.

```
while 1 do
0  switch typ do
    case benchmark do
1      if výsledok je v poriadku (kód 0) then
2          prečítaj zmeraný výkon
3          if mód útoku je mask then
4              | preveď prepočet na hashcat-indexy
5              ulož výkon hosta do databázy
6              prečítaj čas benchmarku a ulož do databázy
          else
              | naplánuj nový benchmark na dlhší čas
    case normal do
7      if heslo bol nájdené (kód 0) then
8          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
9          if heslo nebolo nájdené (kód 1) then
10             prečítaj čas výpočtu a ulož do databázy
11             if úloha mala dostatočnú veľkosť then
12                 | uprav výkon hosta podľa dĺžky výpočtu poslednej úlohy
13                 pričítaj veľkosť spracovane úlohy k počtu už overených hesiel
            else
                /* Chyba výpočtu */
                zruš všetky bežiace úlohy daného hosta
                nastav týmto hostom príznak "retry"
                nastav výkon hosta na 0
                nastav stav hosta na benchmark (0)
                ukonči asimiláciu
14  if systém je nastavený na mazanie úloh then
15      | zmaž položku úlohy z databázy
      else
16      | nastav položke úlohy príznak "finished"
```

Meno	Typ	Popis	Vyplňuje
attack	String	[brute, dict, combinator]	webAdmin
attack_mode	UInt	režim útoku Hashcatu (-a)	webAdmin
hash_type	UInt	0, 9400, 10500, ...	webAdmin
name	String	meno úlohy	webAdmin
left_rule	String	pravidlo pre ľavý slovník	webAdmin
right_rule	String	pravidlo pre pravý slovník	webAdmin
mask	String	hashcat maska	Generator
start_index	BigUInt	počiatočný index	Generator
hc_keyspace	BigUInt	počet hashcat indexov	Generator
mask_hc_keyspace	BigUInt	potrebný pre výpočet pokroku	Generator
mode	String	[n, b]	Generator

Tabuľka 4.1: Zoznam záznamov TLV slovníka.

Kapitola 5

Návrh testov pre systém Fitcrack

K návrhu testov pre systém Fitcrack potrebujeme poznať:

- architektúru systému Fitcrack 4,
- moduly, ktoré budú testované:
 - Asimilator 4.2.4,
 - Generator 4.2.3,
 - Runner 4.2.5,
 - API 4.2.2.
- kedy budú ktoré moduly testované 2.1,
- spôsoby testovania 2.4,

Fitcrack pozostáva z niekoľkých častí, tak ako sú navrhnuté platformou BOINC [12]. V mojej práci som sa rozhodol testovať s použitím zdrojových kódov, keďže nemám k dispozícii kompletnú špecifikáciu systému, alebo jeho komponentov. Tvorba testov vychádza z požiadaviek na konkrétny komponent. U modulov Generator a Asimilator, ktorých činnosť je popísaná pseudokódmi 1 a 2 som sa rozhodol vytvoriť CFG a vytvoriť požiadavky na základe prechodu grafu. Na nájdenie primárnych ciest grafu som použil nástroj Prime Path Coverage¹.

Podľa Google blogu o testovaní² spadajú do kategórie integračných testov 2.4, no každý test testuje iba jediný modul, takže budem používať názov testovanie modulov.

Prístup do databázi je pomocou frameworku SQL Alchemy³. Je to jeden z najpoužívanejších frameworkov v jazyku Python, dovoľuje používať ORM (Object Relation Mapper)⁴ a využíva ho aj API, takže ho nebude potrebné inštalovať.

5.1 Návrh testov pre modul Generator

Generator má na starosti generovanie nových úloh pre klientov a rozdeľovanie nových, či neúspešne dokončených úloh. Komunikuje výhradne s databázou, beží v nekonečnom cykle,

¹<https://github.com/heshenghuan/Prime-Path-Coverage>

²<http://blog.codepipes.com/testing/software-testing-antipatterns.html>

³<https://www.sqlalchemy.org/>

⁴https://en.wikipedia.org/wiki/Object-relational_mapping

kde periodicky kontroluje bežiacie balíky a ak je to potrebné generuje nové úlohy, alebo prerozdeľuje neúspešne dokončené úlohy. Generator dopĺňa TLV slovník 4.2.6, ktorý sa vytvára pri pridaní nového balíčka. Doplnený TLV slovník je určený pre Runner, ktorý podľa neho vie o akú úlohu sa jedná a aké parametre má poslať nástroju Hashcat.

Generator je najzložitejší modul a tomu odpovedá aj graf toku riadenia 5.1, ktorý obsahuje aj veľa cyklov. Pre testovanie bol pôvodný zámer implementovať pokrytie primárnych ciest n3, no po vygenerovaní všetkých primárnych ciest pomocou vyššie spomínaného programu Prime path coverage bolo od tohto zámeru upustené, keďže počet primárnych ciest je 777. Nakoniec som uznal za vhodné vybrať kritérium pokrytia všetkých uzlov (node coverage).

Kontrolované bude či Generator vytvára správne záznamy v databáze, či správne mení hodnoty existujúcich záznamov v databáze a či správne vytvára TLV konfiguračné súbory 4.2.6.

5.2 Návrh testov pre modul Asimilator

Asimilator je podobne ako Generator serverový modul a komunikuje výhradne pomocou databázy. Jeho funkcia spočíva v spracovávaní výsledkov. Narozdiel od modulu Generator je Asimilator volaný z platformy BOINC, keď je v databáze overený výsledok úlohy. Pred každým testom je teda potrebné pripraviť záznamy tabuliek databázy so správnymi hodnotami, aby bol Asimilator zavolaný. Vstupný bod modulu Asimilator je funkcia `assimilate_handler()`⁵.

Asimilator je jednoduchší modul, ako Generator, takže je možné dosiahnuť pokrytie všetkých primárnych ciest. Z CFG modulu Asimilator (obrázok 5.2) som vytvoril 14 primárnych ciest 3.3:

```
[0, 6, 14, 15]
[0, 6, 14, 16]
[0, 1, 5, 14, 15]
[0, 1, 5, 14, 16]
[0, 6, 7, 8, 14, 15]
[0, 6, 7, 8, 14, 16]
[0, 6, 7, 13, 14, 15]
[0, 6, 7, 13, 14, 16]
[0, 1, 2, 3, 4, 14, 15]
[0, 1, 2, 3, 4, 14, 16]
[0, 6, 7, 9, 10, 12, 14, 15]
[0, 6, 7, 9, 10, 12, 14, 16]
[0, 6, 7, 9, 10, 11, 12, 14, 15]
[0, 6, 7, 9, 10, 11, 12, 14, 16]
```

Pri testovaní modulu Asimilator bude potrebné kontrolovať aké zmeny boli prevedené v databáze a či je daná úloha úspešne asimilovaná.

⁵<https://boinc.berkeley.edu/trac/wiki/AssimilateIntro>

5.3 Návrh testov pre modul Runner

Runner má za úlohu spúšťať Hashcat podľa konfiguračného TLV súboru 4.2.6, kontrolovať priebeh výpočtu nástroja Hashcat a keď výpočet skončí Runner by mal vytvoriť výstupný súbor s výsledkami úlohy 4.2.6.

Testovanie modulu Runner je potrebné rozdeliť na 2 časti: testovanie, že Runner spúšťa Hashcat so správnymi parametrami a testovanie, že Runner vytvára správne výstupné súbory podľa výstupu nástroja Hashcat. Neexistuje presná špecifikácia modulu Runner, takže nie je možné použiť žiadnu z metód popisovaných v kapitole 3. Je však možné inšpirovať sa testovaním založenom na vstupných doménach 3.4, keďže nepoznáme vnútornú špecifikáciu, ale vieme ako sa ma správať pre rôzne kategórie vstupov a výstupov. Aby sme mali možnosť testovať argumenty, s ktorými Runner spúšťa Hashcat a takisto mohli kontrolovať výstup nástroja Hashcat, je potrebné implementovať náhradu/dvojníka za nástroj Hashcat. V našom prípade to bude mock 3.5, keďže nesprávne argumenty môžu spôsobiť zlyhanie testu a náhrada bude vracať rôzne výstupy v závislosti na vstupe. Pre Runner máme 2 modely vstupných domén 3.4:

- **typ úlohy:**
 - **benchmark**,
 - **normal** – ktorý sa ďalej delí podľa typu útoku:
 - * útok s použitím masky,
 - * slovníkový útok,
 - * kombinačný útok.
- **výstup nástroja Hashcat:**
 - úspech,
 - úspech s upozornením,
 - neúspech.

Kombináciou týchto blokov nám vznikne 12 požiadaviek na testy.

5.4 Návrh testov pre aplikačné rozhranie

Aplikačné rozhranie (API) z časti nahrádza funkciu modulu WebAdmin, vzhľadom nato, že je cez neho možné ovládať všetky časti systému Fitcrack ako pridávanie nových balíčkov, hostov, spúšťanie, pozastavenie lámania alebo celého projektu cez BOINC ovládacie prvky a iné. API je rozdelené do koncových bodov, pričom každý má na starosti inú časť systému 4.2.2. Koncové body sú ďalej rozdelené na funkcie a majú rôzne parametre. Na funkcie a ich parametre je možné aplikovať testovanie založené na vstupných doménach 3.4. Odpovede na požiadavky v API sú posielané vo formáte **json**, takže pre každý objekt používaný v databázi a dotazovaní pomocou API bude potrebné implementovať ekvivalentný prevod objektu na **json**, tak ako to robí API.

Vzhľadom nato, že API je stále vo vývoji nie je možné otestovať všetky jeho koncové body (endpoints), preto som vybral niekoľko už implementovaných a dôležitých koncových bodov pre fungovanie systému Fitcrack:

- **hashcat** – slúžia na prácu s nástrojom Hashcat,

- **host** – operácie s hosťami,
- **jobs** – operácie s balíčkami,
- **serverInfo** – slúži na zistenie stavu servera,
- **dictionary** – operácie so slovníkmi,
- **masks** – operácie s maskami,
- **rule** – operácie so súbormi obsahujúcimi pravidlá pre slovníky,
- **charset** – operácie s charset súbormi.

Každý z týchto koncových bodov obsahuje množinu operácií a každá operácia má niekoľko parametrov. Jeden koncový bod bude reprezentovaný jednou triedou. Každý parameter bude testovaný najskôr samostatne a následne aj všetky kombinácie parametrov. Hodnoty parametrov budú rozdelené do blokov 3.4. Pre tieto koncové body bude vytvorená testovacia sada, ktorá bude spĺňať pokrytie všetkých kombinácií vstupných domén pre všetky funkcie.

Rozdelenie parametrov do blokov

V špecifikácii rozhrania sú uvedené prípustné hodnoty parametrov. q_n predstavuje rozklad domény 3.4 Môžu byť špecifikované:

- zoznamom prípustných hodnôt – bloky takéhoto parametru budú pozostávať z:
 - predvolená hodnoty, ak existuje,
 - všetkých prípustných hodnôt a
 - jednej neprípustnej hodnoty,

Príklad:

Zobrazenie hostov podľa stavu (stav môže byť active, alebo inactive, 0 znamená, že sa nastaví predvolená hodnota):

$$q_1 = (0, \text{active}, \text{inactive}, \text{hungry}) \quad (5.1)$$

- typom – parametre špecifikované typom bude rozdelené do blokov pozostávajúcich z:
 - platnej hodnoty nachádzajúcej sa v databáze,
 - platnej hodnoty nenachádzajúcej sa v databáze.

Príklad:

Zobrazenie konkrétneho hosta podľa id:

$$q_2 = (1, 795684318) \quad (5.2)$$

Funkcia môže obsahovať viac parametrov, takže nakoniec bude otestovaná aj kombinácia týchto blokov. Napríklad pri filtrovaní užívateľov, ktorý sa podieľajú na lámaní – užívateľov môžeme filtrovať podľa statusu a podľa id balíčka na ktorom pracujú. Povedzme, že status

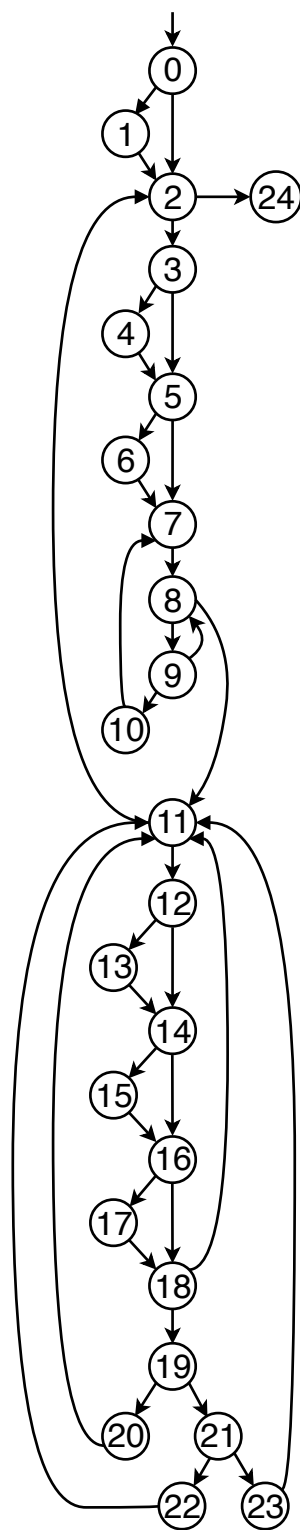
užívateľa môže byť *cracking* a *ready* a že v databázi sa nachádza práve 1 balíček a má id 1. q_1 bude predstavovať rozklad domény statusov a q_2 rozklad domény id:

$$q_1 = (0, \textit{cracking}, \textit{ready}, \textit{hungry})$$

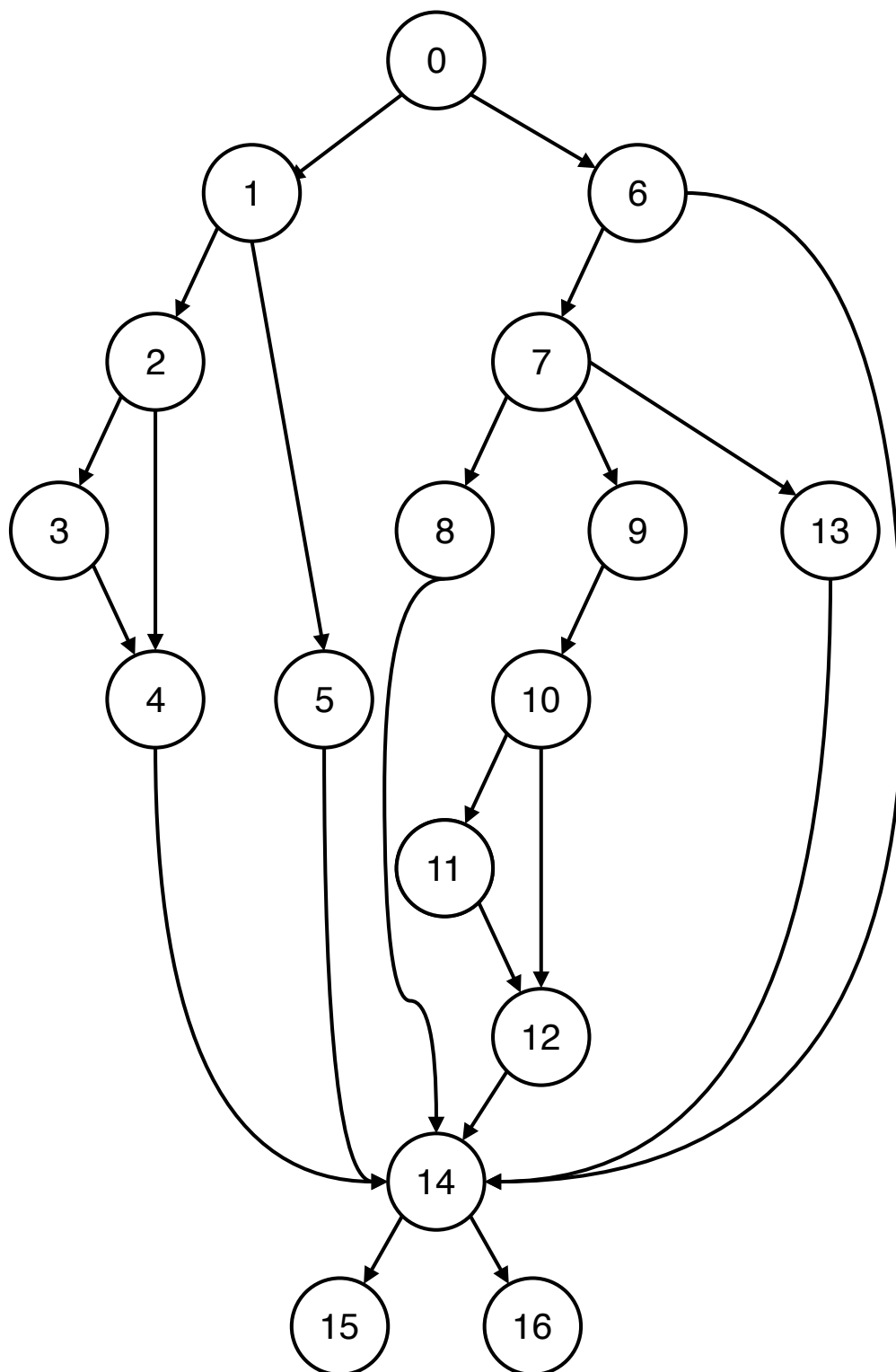
$$q_2 = (1, 2)$$

Kritérium pokrytia všetkých blokov 3.4 hovorí, že testovacia sada musí pokryť tieto požiadavky:

(0, 1)	(<i>cracking</i> , 1)	(<i>ready</i> , 1)	(<i>hungry</i> , 1)
(0, 2)	(<i>cracking</i> , 2)	(<i>ready</i> , 2)	(<i>hungry</i> , 2)



Obr. 5.1: CFG modulu Generator vytvorené z pseudokódu 1.



Obr. 5.2: CFG modulu Asimilator vytvořené z pseudokódu 2.

Kapitola 6

Implementácia testovacích sád

Pre implementáciu bol vybraný framework `unittest` [9] z čoho vyplýva, že testy sú implementované v jazyku Python3. Aby bolo možné moduly systému Fitcrack testovať samostatne je testovacia sada pre každý modul implementovaná vo vlastnom súbore, ktorý obsahuje práve 1 triedu, výnimkou je testovacia sada pre API, kde je niekoľko tried. Každá obsahuje testovaciu sadu pre práve jeden koncový bod 5.4.

Funkcie pre ovládanie systému Fitcrack, jeho modulov a definície tried používaných pri testovaní zoskupuje súbor s názvom `fc_test_library.py`. Takisto definuje triedy reprezentujúce súbory používané v systéme Fitcrack:

- **FitcrackTLVConfig**, reprezentuje TLV konfiguračný súbor posielať klientovi. Objekt tejto triedy je možné vytvoriť zadaním parametrov, alebo prečítať priamo zo súboru. Tento objekt je následne možné reprezentovať ako reťazec a zapísať do súboru. Trieda je používaná na generovanie a kontrolovanie konfiguračných súborov vytvorených modulom Generator.
- **RunnerOutput** reprezentuje výstupný súbor modulu Runner. Objekt triedy môže obsahovať rôzne atribúty v závislosti na type úlohy z ktorej bol generovaný.

Samotné testovacie sady sú implementované v súboroch s názvami vo formáte `test_module_name.py`, kde `module_name` môže byť `generator`, `assimilator`, `runner`, alebo `api`. Každému z týchto súborov je venovaná samostatná sekcia.

Komunikácia s databázou

Vzhľadom nato, že všetky testované moduly okrem modulu Runner komunikujú výhradne s databázou, bola vytvorená knižnica funkcií na prístup k objektom databázy. Funkcie používajú framework `SQLAlchemy`. Mapovanie objektov na tabuľky je zhodné s tým ktoré používa API [6], preto moje zdrojové kódy potrebujú symbolický odkaz na zdrojové kódy API. Testovacie prípady by mali využívať túto knižnicu na prístup k databáze, no pre špeciálne prípady prístupu k dátam je možné použiť aj metódy frameworku `SQLAlchemy`.

Fixtures

Testy využívajú vzor fixtures 3.5, konkrétne funkcie z frameworku `unittest`:

- **setUpClass** je volaná pred spustením prvého testu a pripravuje systém na testovanie modulu. Typicky vypína všetky systémové moduly, okrem testovaného a pripravuje

databázu: vymaže všetky záznamy z tabuliek, ktoré budú testované a pridá záznamy, ktoré sa nebudú meniť počas testovania celého modulu, napríklad pridanie balíčka pri testovaní modulu Asimilator.

- **tearDownClass** je volaná po skončení posledného testu a má za úlohu vrátiť SUT do stavu pred testovaním, takže opäť zapne ostatné moduly a vymaže záznamy, ktoré boli pridané počas testovania.
- **setUp** je funkcie volaná pred každým testom, typicky nastavuje parametre systému Fitcrack, ktoré sa menia počas testu, alebo pridáva záznamy.
- **tearDown** je volaná po skončení každého testu a väčšinou má na starosti mazanie záznamov tabuliek, ktoré by mohli ovplyvniť následné testy.

6.1 Implementácia testovacej sady pre serverový modul Asimilator

Pri bližšom pohľade na CFG modulu Asimilator 5.2 a na zoznam primárnych ciest 5.2 je jasne viditeľné, že každá cesta sa opakuje len so zmenou posledného uzlu. Uzol číslo 15 znamená, že asimilovaná úloha sa má zmazať a uzol číslo 16 nastavuje asimilovanej úlohe príznak **finished**. Framework unittest ponúka pre tieto účely možnosť definovať si **subtest**¹, ktorý slúži na spúšťanie podobných testov. Ak sa testovací prípad líši napríklad v jednom parametri, môže byť tento parameter predaný funkciou **subtest**. Ak následne zlyhá testovací prípad, tak na výpise sa bude aj hodnota tohto parametru. V tomto prípade je parametrom príznak mazania dokončených úloh.

Príprava SUT a návrat SUT do stavu pred testovaním

setUpClass: pred spustením testov je potrebné vymazať z databázy všetky záznamy, ktoré by mohli ovplyvniť testy, zastaviť všetky serverové moduly okrem modulu Asimilator a pridať balíček, pre ktorý budú generované dokončené úlohy, keďže Asimilator má na starosti spracovať výsledky dokončených úloh.

tearDownClass: po testoch je analogicky potrebné spustiť všetky serverové moduly, vymazať tabuľky, ktoré boli ovplyvnené testami a zrušiť príznak mazania dokončených úloh z databázy.

BOINC hierarchická štruktúra priečinkov

Súčasťou dokončenej úlohy je výstupný súbor modulu Runner, ktorý sa však nenachádza priamo v databáze, keďže Fitcrack podporuje hierarchickú štruktúru priečinkov so stihnutými a odslanými súbormi². V praxi to funguje tak, že na serveri existujú zložky **downloads** a **uploads**. Každá z týchto zložiek ďalej obsahuje množstvo priečinkov s 3 znakovými názvami pozostávajúcimi z písmen, alebo čísel. Tieto názvy sa vypočítajú ako hashe názvov súborov, ktoré obsahujú. BOINC poskytuje nástroj **dir_hier_path**, ktorý berie na vstup meno súboru a výstup je cesta k tomuto súboru.

¹<https://docs.python.org/3.6/library/unittest.html#distinguishing-test-iterations-using-subtests>

²<https://boinc.berkeley.edu/trac/wiki/DirHierarchy>

Databáza potom obsahuje XML súbor s odkazmi súborov patriacich k danej úlohe. BOINC dokáže v tejto štruktúre vyhľadávať súbory len podľa názvu.

Pre potreby testovania je naopak potrebné umiestniť súbory do správnych priečinkov a vytvoriť XML súbor, ktorý je predaný cez databázu modulu Asimilator.

Čakanie na dokončenie asimilácie

Čas testovania je najviac ovplyvnený tým, ako test zistí, že Asimilator dokončil úlohu. Tým, že je Asimilator volaný z platformy BOINC, tak nie je možné presne zistiť kedy dokončil úlohu. Ponúkajú sa 3 riešenia:

- Najjednoduchšie riešenie je počkať dopredu stanovený limit, no ten je veľmi ťažké určiť, keďže nie je známe ako často BOINC kontroluje obsah tabuliek a volá vstupnú funkciu modulu Asimilator `assimilate_handler()`. Takisto je ťažké určiť koľko bude trvať, kým sa do databázy zapíšu všetky záznamy.
- Ďalšia možnosť je počkať, kým Asimilator zapíše niečo do logovacieho súboru. Na začiatku testovania bol tento spôsob využívaný, ale testy trvali veľmi dlho, v priemere okolo 9 sekúnd na test, pričom údaje boli v databáze zapísané už skôr. Predsa len súborový systém nie je navrhnutý nato zapísať zmeny v súbore čo najrýchlejšie. Spoliehať sa pri testovaní nato, že SUT zapíše niečo do logovacieho súboru tiež nie je veľmi dobré riešenie.
- Nakoniec bolo implementované cyklické dotazovanie databázy na zmenu príznaku `assimilate_state`, alebo na vymazanie konkrétnej úlohy, podľa toho, či je v danom teste nastavený príznak úlohy `delete_finished`. Vďaka použitiu frameworku SQL Alchemy, ktorý sa sám stará o obnovovanie dát z databázy je toto riešenie jednoduché a funguje dobre.

Pokrytie primárnych ciest

Nasledujúci zoznam popisuje pre každý testovací prípad/funkciu zoznam primárnych ciest, ktoré pokrýva 3.3. Ako bolo spomínané v návrhu 5.2, testovacia sada spĺňa pokrytie všetkých primárnych ciest n3. Zoznam primárnych ciest bol vytvorený z grafu 5.2, ktorý bol zase vytvorený z pseudokódu modulu Asimilator 2.

- `test_bench_error`: $t_1 = \{[0, 1, 5, 14, 15], [0, 1, 5, 14, 16]\}$
- `test_normal_found`: $t_2 = \{[0, 6, 7, 8, 14, 15], [0, 6, 7, 8, 14, 16]\}$
- `test_normal_error`: $t_3 = \{[0, 6, 7, 13, 14, 15], [0, 6, 7, 13, 14, 16]\}$
- `test_bench_ok_mask`: $t_4 = \{[0, 1, 2, 3, 4, 14, 15], [0, 1, 2, 3, 4, 14, 16]\}$
- `test_normal_not_found_small`: $t_5 = \{[0, 6, 7, 9, 10, 14, 15], [0, 6, 7, 9, 10, 14, 16]\}$
- `test_normal_not_found`: $t_6 = \{[0, 6, 7, 9, 10, 11, 12, 14, 15], [0, 6, 7, 9, 10, 11, 12, 14, 16]\}$,
- `test_bench_no_file`: $t_7 = \{[0, 1, 14, 15], [0, 1, 14, 16]\}$
- `test_bench_ok`: $t_8 = \{[0, 1, 2, 4, 14, 15], [0, 1, 2, 4, 14, 16]\}$

Cesty `[0, 6, 14, 15]` a `[0, 6, 14, 16]` sú syntakticky nedosiahnuteľné, keďže typ úlohy nemôže byť iný ako `normal` alebo `benchmark`.

$$T_1 = t_1, t_2, \dots t_6 \text{ splňuje PPC.}$$

Bližšie informácie o implementácii jednotlivých testovacích prípadov poskytujú zdrojové kódy.

6.2 Implementácia testovacej sady pre serverový modul Generator

Testovanie modulu Generator rozsiahle a implementované testy síce splňajú kritérium pokrytia všetkých uzlov pseudokódu 1, ale pseudokód nespomína niektoré celkom závažné detaily implementácie, ako rôzny typy úloh, výpočet veľkosti úlohy v závislosti na sile hosta, delenie úlohy pri opätovnom pridelení nedokončenej úlohy a iné.

Je potrebné overovať správnosť konfiguračných súborov vo formáte XML, ako aj TLV konfiguračných slovníkov vytvorených modulom Generator pre Runner. Pred spustením testov musí databáza obsahovať slovníky používané pri testovaní.

Čakanie na vygenerovanie úlohy

Podobne ako pri module Asimilator 6.1 boli 3 možnosti ako zistiť, že Generator skončil činnosť:

- čakanie pevne stanovenú dobu,
- čakanie na zápis do logovacieho súboru,
- alebo čakanie na zmenu v databáze.

Pre Asimilator bola najlepšia možnosť čakať na zmenu v databáze, keďže stačilo sledovať 2 typy zmien, no Generator vytvára a mení rôzne záznamy rôzne, preto nie je stále možné použiť túto možnosť.

Generator pracuje v nekonečnom cykle a do logovacieho súboru zapisuje niekoľko krát za sekundu. Použitie druhej metódy, čakanie na zápis do logovacieho súboru, podstatne zrýchli testy, no očakávaná zmena nemusela nastať hneď prvom cykle, ktorý bol monitorovaný, čo môže skresľovať výsledky testov.

Nakoniec bola použitá metóda čakania na zmenu v databáze všade tam, kde to bolo možné. Inak je použitá prvá metóda, čakanie pevne stanovenú dobu, aj napriek spomaleniu vykonávania testov. Implementácia obsahuje všetky 3 metódy, hlavne pre rozširovanie testovacej sady, keďže pri module Generator nie je jasné, ktorú je najlepšie používať.

Pokrytie uzlov

Pri module Generator bolo v návrhu 5.1 stanové ako cieľ pokrytie všetkých uzlov (NC) n3. Uzly sú z grafu 5.1, ktorý bol vytvorený z pseudokódu 1. Nasledujúci zoznam hovorí, ktorý testovací prípad pokrýva ktoré uzly:

- `test_del_fin_host`: $t_1 = [0, 24]$,
- `test_del_exh_host`: $t_2 = [0, 24]$,
- `test_passed_time`: $t_3 = [0, 1, 24]$,

- `test_package_set_start_time`: $t_4 = [0, 2, 3, 4, 5, 7, 8, 11, 24]$,
- `test_make_benchmark`: $t_5 = [0, 2, 8, 9, 10, 11, 24]$,
- `test_add_host_to_package`: $t_6 = [0, 2, 3, 5, 7, 24]$,
- `test_enough_jobs`: $t_7 = [0, 2, 3, 5, 7, 11, 12, 13, 11, 24]$,
- `test_make_job_<attack>3`: $t_8 = [0, 2, 3, 5, 6, 7, 11, 12, 14, 15, 16, 18, 24]$,
- `test_set_job_fin`: $t_9 = [0, 2, 3, 5, 7, 8, 11, 12, 14, 16, 17, 18, 24]$,
- `test_retry_job_<attack>3`: $t_{10} = [0, 2, 3, 5, 6, 7, 11, 12, 14, 16, 17, 18, 24]$,
- `test_set_package_timeout`: $t_{11} = [0, 2, 3, 5, 7, 8, 11, 12, 14, 16, 18, 19, 20, 24]$,
- `test_set_package_exhausted`: $t_{12} = [0, 2, 3, 5, 7, 8, 11, 12, 14, 16, 18, 19, 21, 22, 24]$,
- `test_set_package_ready`: $t_{13} = [0, 2, 3, 5, 7, 8, 11, 12, 14, 16, 18, 19, 21, 23, 24]$.

$T_1 = t_1, t_3, t_4, \dots t_{13}$ splňuje NC.

Bolo implementovaných viac testov ako bolo potrebné na pokrytie NC, no pseudokód nerozlišuje medzi typmi útokov a inými implementačnými podrobnosťami.

Podrobnosti k implementácii testov je možné vyčítať zo samotných zdrojových kódov, ako aj z komentárov k zdrojovým kódom.

6.3 Implementácia testovacej sady pre modul Runner

Pred začatím testov by mali byť splnené tieto podmienky:

- zložka s testami bude obsahovať spustiteľný súbor modulu Runner,
- v konfiguračnom súbore `config.py` bude správne meno spustiteľného súboru modulu Runner,
- dvojník nástroja Hashcat `hashcat_mock.py` je prekopírovaný / premenovaný tak ako to vyžaduje Runner, momentálne `hashcat64.bin`,
- premenovaný Hashcat mock musí mať nastavené spustiteľné práva, aby ho bolo možné spustiť aj bez špecifikácie interpretu: `./hashcat64.bin`.

Architektúra

Testovací prípad pripraví súbory 4.2.6 v závislosti na úlohe a zavolá Runner. Do súboru `local.conf` zapíše dodatočné parametre ak je to potrebné:

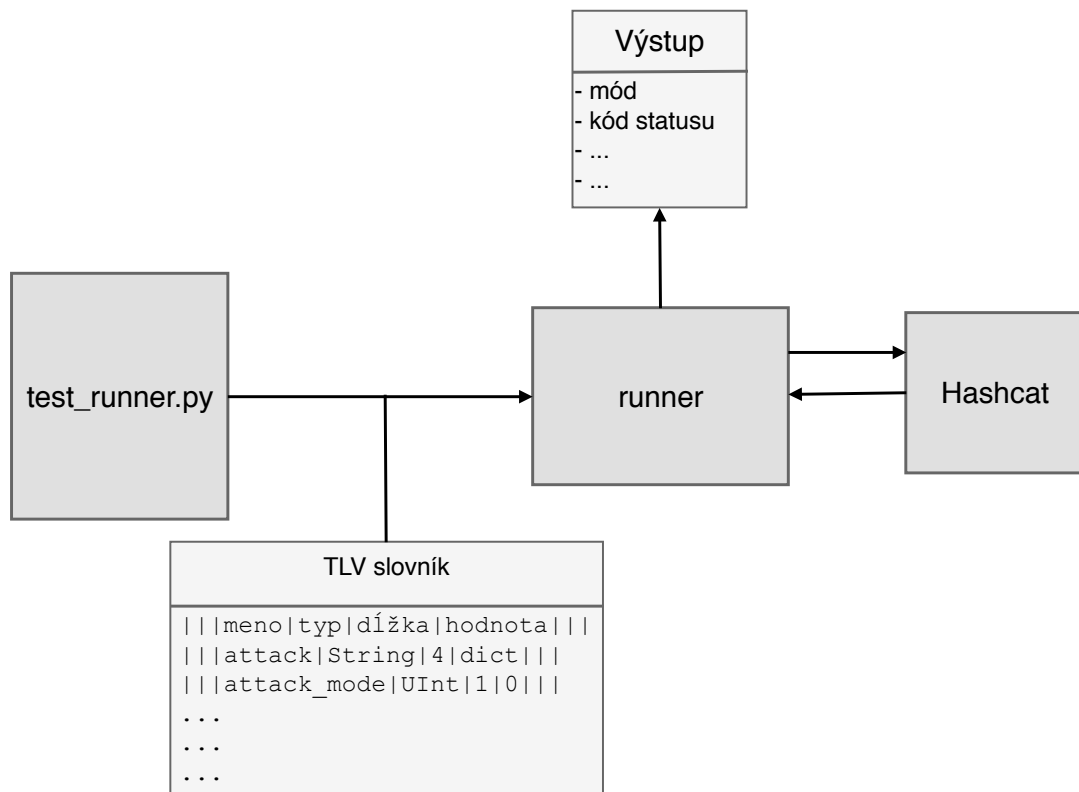
- `-error`: mock prekopíruje na svoj výstup súbor obsahujúci chybový výstup Hashcatu a skončí s návratovou hodnotou -1,

³<attack> môže byť dict (slovníkový), comb (kombinačný), mask, alebo error, kedy má typ útoku neplatnú hodnotu.

- **-warning**: na výstupe je platný výstup z nástroja Hashcat, ale s varovaniami,
- **-found**: slúži na rozlíšenie, či heslo bolo nájdené, alebo nie.

Runner spúšťa mock nástroja Hashcat, kontroluje jeho výstup. Runner vytvára výstupný súbor posielať modulu Asimilator. Hashcat mock spracováva argumenty 6.3 a podľa argumentov kopíruje na výstup jeden z predpripravených výstupov nástroja Hashcat. Do súboru zapíše aj všetky parametre, ktoré mu poslal Runner.

Keď Runner skončí je overované s akým návratovým kódom skončil, aké argumenty poslal nástroju Hashcat 6.3 a obsah výstupného súboru. Schéma je znázornená aj na obrázku 6.1.



Obr. 6.1: Schéma architektúry testov modulu Runner.

Spracovávanie argumentov

Pre spracovávanie argumentov, ktoré Runner posiela nástroju Hashcat som použil modul **argparse**⁴, ktorý funguje tak, že je potrebné vytvoriť objekt **parser**, ktorému je možné neskôr špecifikovať argumenty. Súbor **hashcat_parsers.py** definuje 2 objekty:

- **initial/benchmark parser** – slúži na zistenie základných informácií, ktoré by mali byť posielať stále, alebo iba pri úlohe typu **benchmark**,
- **normal parser** – slúži na spracovávanie parametrov normálnej úlohy.

⁴<https://docs.python.org/3/library/argparse.html>

Súbor ďalej obsahuje aj funkcie na prácu s týmito objektmi. Prvá je stále volaná funkcia `initial_parse()`, ktorá vráti dvojicu: objekt reprezentujúci argumenty a pole nespracovaných argumentov. Podľa typu úlohy, alebo útoku je volaná jedna z nasledujúcich funkcií:

- `parse_mask_attack()`,
- `parse_dict_attack()`,
- `parse_comb_attack()`.

Testovacie prípady by mali pracovať výhradne s týmito funkciami.

6.4 Implementácia testovacej sady aplikačného rozhrania (API)

Súbor `test_api.py` je jediný zo súborov s testovacou sadou, ktorý obsahuje viac tried. Na testovanie každého koncového bodu (endpoint) API slúži práve 1 trieda, tak ako to opisuje návrh 5.4, tým pádom je možné viac využívať `setUp` a `tearDown` funkcie. Koncový bod takisto posiela iné dáta a práve na vytváranie json objektov v rovnakom formáte slúžia funkcie zo súboru `api_response_models.py`. Pre každý koncový bod môžu byť dáta toho istého typu posielané v inom formáte, v závislosti na potrebe posielaných informácií. Napríklad formát objektu `host` je rozdielny ak sa posiela v kontexte pripojeného výpočtového uzla, kde je informácií viac ako v kontexte balíčka, kde sa posiela zoznam objektov `host` len so základnými informáciami.

Testovacia sada API často využíva `Subtest` 6.1 a snaží sa kombinovať bloky domén vytvorených z parametrov funkcií. Pre väčšiu prehľadnosť je najskôr funkcia testovaná iba s jedným parametrom. Je otestovaná funkčnosť funkcie so zástupcom z každého bloku testovaného parametru. Takto sú postupne otestované všetky parametre jednej funkcie (ostatné parametre majú predvolené hodnoty). Až keď prejdú testovacie prípady pre každý parameter zvlášť je spustený testovací prípad kombinujúci všetky parametre. Tento testovací prípad sám spĺňa pokrytie všetkých kombinácií blokov parametrov (All Combination Coverage) 3.4. Funkcia, ktorá kombinuje všetky parametre môže vygenerovať tisíce testovacích prípadov, takže pri závažnejšej chybe, ktorá spôsobí zlyhanie väčšieho množstva prípadov je ťažké určiť príčinu.

Kapitola 7

Testovanie

Táto kapitola popisuje spôsoby testovania. Prvá časť môže slúžiť ako návod na spúšťanie testov a odporúčanie ako čo najefektívnejšie zistiť príčinu chyby. Ako príklad spúšťania a výpisov bude použitý skript na testovanie modulu Runner.

7.1 Spúšťanie testov

Testy je možné spúšťať niekoľkými spôsobmi:

- s použitím modulu `unittest` jazyka Python, kde je potrebné špecifikovať názov modulu – názov súboru bez prípony a voliteľne je možné špecifikovať aj triedu a konkrétnu funkciu. Modulu `unittest` je možné pridať aj parametre ako napríklad množstvo informácií, ktoré má vypísať pomocou parametru `-v` [9]:

```
python3 -m unittest test_runner
python3 -m unittest test_runner.TestRunner
python3 -m unittest -v 3 test_runner.TestRunner.test_benchmark_ok
```

- klasicky, ako Python skript:

```
python3 test_runner.py
```

- pomocou skriptu `main.py`, ktorý postupne spúšťa testy pre všetky testované moduly:

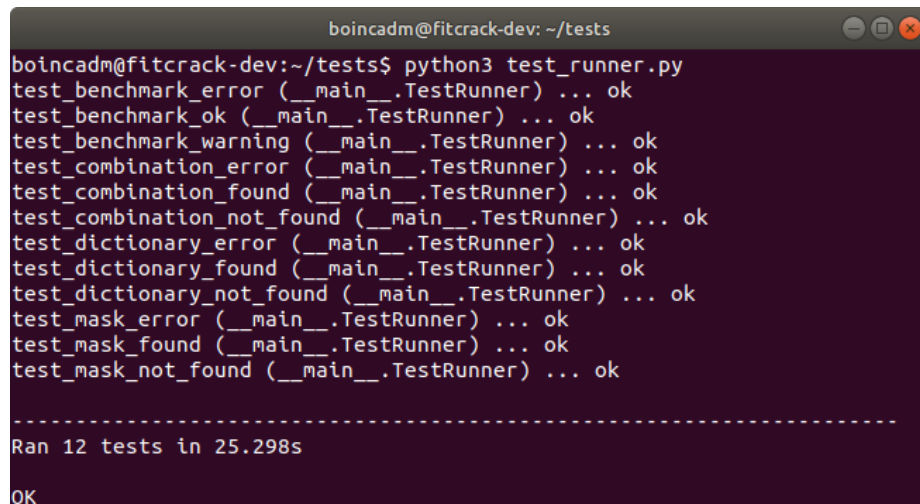
```
python3 main.py
```

Prvé 2 metódy majú ekvivalentné chovanie, ale líšia sa výpisom informácií. Pri spúšťaní testov ako skript je výstup testovacích metód presmerovaný do súboru s názvom `module_name_output.txt`, takže pre Runner to bude `test_runner_output.txt`, ale je nastavený parameter `verbosity` na hodnotu 3, takže sa vypisuje názov testovacieho prípadu. Príklad výpisu testov spúšťaných ako skript je na obrázku 7.1.

Mne sa najlepšie overilo spúšťať testy pomocou Python skriptu, kvôli strohému výpisu a prehľadnosti výsledkov a ak niektorý z testov neprejde (fail), tak pomocou modulu `unittest` spustiť konkrétny test, kde mi rozsiahlejší výpis môže pomôcť pri identifikácii problému.

Pri spúšťaní testov s použitím modulu `unittest` sú vypísané všetky informácie, napríklad informácie, že do databázy je pridaný záznam, alebo pri testovaní modulu Runner je presmerovaný celý výstup modulu, ako to zobrazuje obrázok 7.2

Zároveň s implementáciou testovacích sád prebieha aj odhaľovanie chýb v testovaných moduloch. Bolo otestovaných niekoľko verzií modulov Generator, Runner a aj API.



```
boincadm@fitcrack-dev: ~/tests
boincadm@fitcrack-dev:~/tests$ python3 test_runner.py
test_benchmark_error (__main__.TestRunner) ... ok
test_benchmark_ok (__main__.TestRunner) ... ok
test_benchmark_warning (__main__.TestRunner) ... ok
test_combination_error (__main__.TestRunner) ... ok
test_combination_found (__main__.TestRunner) ... ok
test_combination_not_found (__main__.TestRunner) ... ok
test_dictionary_error (__main__.TestRunner) ... ok
test_dictionary_found (__main__.TestRunner) ... ok
test_dictionary_not_found (__main__.TestRunner) ... ok
test_mask_error (__main__.TestRunner) ... ok
test_mask_found (__main__.TestRunner) ... ok
test_mask_not_found (__main__.TestRunner) ... ok

-----
Ran 12 tests in 25.298s

OK
```

Obr. 7.1: Výpis testovania modulu Runner spúšťaného ako skript.

7.2 Experimenty s testovacou sadou

Ako pri každom produkte, tak aj pri testoch je vhodné overiť funkčnosť. Konkrétne pri mnou implementovanej sade testov bolo nutné overiť, či nezávisí od dát v databáze, keďže podľa vzoru fixtures 3.5 by všetky potrebné dáta a nastavenia mal pripraviť samotný test a po prevedení by SUT mal byť vrátený do pôvodného stavu.

Testy boli spúšťané opakovane, s tým, že medzi spusteniami boli robené ručné zmeny v databáze, alebo boli v zdrojovom kóde zakomentované funkcie, ktoré sa starajú o mazanie dát po prevedení testu. Takisto bola overená funkčnosť na novo vytvorenom projekte, takže projektová databáza neobsahovala žiadne dáta. Bola teda overená funkčnosť testovacej sady pri prázdnej databáze, aj pri databáze, ktorá obsahovala niekedy aj stovky záznamov v jednotlivých tabuľkách.

7.3 Výsledky

Testovanie modulu Asimilator odhalilo, že ak bol nastavený príznak mazania dokončených úloh, tak Asimilator nemazal úlohy. Problém bol v komunikácii s databázou a chyba bola opravená takmer okamžite. Asimilator je pomerne jednoduchý modul, ktorý už funguje nejakú dobu, no aj tak testovacia sada dokázala odhaliť chybu. V budúcnosti nie sú plánované žiadne zmeny pre tento modul, takže nie je potrebné rozširovať jeho testovaciu sadu.

Testovanie modulu Generator neodhalilo žiadne chyby. Zároveň s testovaním bola vo vývoji aj nová verzia modulu Generator, ktorá už bola takisto otestovaná. Testovanie neodhalilo žiadne chyby, ale testovacia sada môže byť ďalej použiteľná pri regresnom testovaní 2.4. S pokračujúcim vývojom tohto modulu budú pridávané aj testovacie prípady, keďže nový Generator rozširuje funkčnosť starého.

S novou verziou modulu Generator je takisto vo vývoji aj nová verzia modulu Runner. Testovanie zatiaľ prebehlo iba na starej verzii, kde testy prešli. Nová verzia pridáva množstvo novej funkcionality a mierne mení formáty súborov, takže pre novú verziu bude musieť byť testovacia sada pozmenená.

Pri testovaní aplikačného rozhrania (API) boli zistené viaceré chyby, ako napríklad chyba pri filtrovaní balíčkov podľa času a statusu, chyba pri overovaní hashov a chyba pri


```
boincadm@fitcrack-dev: ~/tests
Progress 5.12716e+06
Progress 5.12716e+06
Progress 5.12716e+06
Progress 5.24091e+06
Progress 5.24091e+06
Progress 5.24091e+06
Progress 5.24091e+06
Progress 5.35447e+06
Progress 5.35447e+06
Progress 5.35447e+06
Progress 5.46822e+06
ExitCode: 1 Error: no code
Password: Not found
.file: local.conf localConfigContent: --error
ExitCode: 255 Error: # hashcat (v3.6.0)Can't find X server!Trying to open directly...Device open failed
Password: Trying to open directly...Device open failed
.file: local.conf localConfigContent: --found
Progress 9.28134e+06
ExitCode: 0 Error: test_pass
Password: test_pass
.file: local.conf localConfigContent:
Progress 1.99378e+07
ExitCode: 1 Error: no code
Password: Not found
.file: local.conf localConfigContent: --error
ExitCode: 255 Error: # hashcat (v3.6.0)Can't find X server!Trying to open directly...Device open failed
Password: Trying to open directly...Device open failed
.file: local.conf localConfigContent: --found
Progress 1
Progress 36
Progress 1296
Progress 0
ExitCode: 0 Error: 2580
Password: 2580
.file: local.conf localConfigContent:
Progress 819200
Progress 1.6384e+06
Progress 2.4576e+06
Progress 3.2768e+06
Progress 4.096e+06
Progress 4.9152e+06
Progress 5.7344e+06
Progress 6.71744e+06
Progress 7.53664e+06
Progress 8.35584e+06
Progress 9.33888e+06
Progress 1.01581e+07
Progress 1.09773e+07
Progress 1.18814e+07
ExitCode: 1 Error: no code
Password: Not found
:
-----
Ran 12 tests in 25.223s

OK
boincadm@fitcrack-dev:~/tests$
```

Obr. 7.2: Výpis testovania modulu Runner spúšťaného pomocou modulu unittest.

vyhľadávanií objektov `host`. Priložená verzia API obsahuje tieto chyby, no žiadna z nich vážne neobmedzuje funkcionálnosť systému Fitcrack.

Kapitola 8

Záver

Cieľom práce, ako je to spomenuté už v úvode nie je plne otestovať systém Fitcrack, keďže je to rozsiahly systém skladajúci sa z niekoľkých modulov, ktoré sú stále vo vývoji. Cieľom práce je otestovať základnú funkcionálnu a pripraviť vývojárom prostredie a návod na vytváranie ďalších testovacích prípadov. Veľká časť práce je preto zameraná na teóriu testovania, konkrétne na testovanie založenom na požiadavkách. Na implementáciu testov bol použitý jazyk `Python3` a jeho framework `unittest`.

Asimilator je asi najjednoduchší modul a bolo možné dosiahnuť 100 % pokrytie všetkých primárnych ciest (PPC). Cesty ako aj graf vychádzajú iba zo pseudokódu, ktorý zjednodušuje niektoré časti reálnej implementácie. Je preto na zváženie, či by mali byť na testovanie využité zdrojové kódy, čo by neúnosne zvýšilo náročnosť testovania, alebo o ktoré časti by mal byť pseudokód doplnený a následne analogicky prerobené aj primárne cesty a aj samotná testovacia sada.

Generator je podstatne zložitejší modul ako Asimilator a tomu aj odpovedá jeho pseudokód. Implementovať testovaciu sadu, ktorá by mala pokryť 777 primárnych ciest je časovo náročné, preto som zvolil iba pokrytie všetkých uzlov (node coverage). Toto kritérium bolo splnené na 100 %, no opäť sú v pseudokóde vynechané niektoré podstatné časti reálnej implementácie. Generator je natoľko zložitý, že iba jeho testovanie by mohla byť práca na niekoľko mesiacov, keďže jedna z jeho funkcií je napríklad adaptívne plánovanie veľkosti úlohy pre každý uzol. Zložitosť modulu je hlavný dôvod, prečo by sa na jeho ďalšom testovaní mal primárne podieľať aj jeho autor. Táto práca by mu mala pomôcť efektívne testovať Generator.

Runner je multiplatformová aplikácia, ktorá ovláda nástroj na lámanie hashov – Hashcat. Pre jeho testovanie bolo potrebné implementovať dvojníka nástroja Hashcat, ktorý sa bude správať tak, ako to testy potrebujú. K modulu Runner neexistuje žiadna špecifikácia, takže bolo problematické vymyslieť požiadavky pre testy, no nakoniec bolo zvolené rozdelenie vstupov do blokov. Výstup nástroja Hashcat je pre Runner tiež vstup. Toto riešenie pokrýva všetky momentálne implementované možnosti modulu Runner.

Bolo testované aj aplikačné rozhranie (API), ktoré bolo vyvíjané paralelne s touto prácou a nahrádza staré webové rozhranie WebAdmin. API nebolo otestované celé, ale iba časti, ktoré zabezpečujú základnú funkcionálnu systém Fitcrack. Väčšinou sú to práve tie časti, ktoré boli súčasťou starého webového rozhrania. API je rozdelené na koncové body (endpoints) a tie obsahujú funkcie s rôznym počtom parametrov. Nad každým parametrom bol prevedený rozklad na bloky podľa odpovedajúcej charakteristiky. Tieto bloky parametrov boli spolu kombinované, aby bolo splnené kritérium pokrytia všetkých kombinácií

vstupných domén (ACoC). Testy odhalili viacero nezávažných chýb, ktoré nemajú vplyv na základnú funkčnosť systému Fitcrack – lámanie hesiel.

Spolu s vývojármi systému Fitcrack je potrebné presnejšie zdefinovať špecifikáciu rôznych modulov a s pokračovaním vývoja systému rozširovať aj testovacie sady. Pre moduly Runner a Generator sú vo vývoji nové verzie, ktoré budú obsahovať novú funkcionálnu s ktorou testovacia sada nepočíta. Testovaciu sadu bude potrebné rozšíriť a niektoré testovacie prípady aj pozmeniť. Napríklad nové verzie budú podporovať viac typov útokov. Keďže prebehlo testovanie iba niektorých častí aplikačného rozhrania je potrebné navrhnuť a implementovať aj testy na ostatné časti. Zvlášť zaujímavá môže byť časť, ktorá rieši prihlasovanie užívateľov a ich oprávnenia prístupu k informáciám.

Literatúra

- [1] Beck, K.: *Simple Smalltalk Testing: With Patterns*. FIT ČVUT v Praze, [Online; navštíveno 07.05.2018].
URL <http://swing.fit.cvut.cz/projects/stx/doc/online/english/tools/misc/testfram.htm>
- [2] Hornický, P.: *Nástroj na testování síťových aplikací*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2012.
URL <http://www.fit.vutbr.cz/study/DP/DP.php?id=13281>
- [3] 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
- [4] Lei, J.: *Software Testing and Maintenance: Graph-Based Testing*. 2011, [Online; navštívené 06.05.2018].
URL <http://crystal.uta.edu/~ylei/cse4321/data/graph-testing.pdf>
- [5] MMA Testing Team: *Software Development Life Cycle*. 2017, [Online; navštívené 16.1.2018].
URL <http://www.althority.com/sdlc/>
- [6] Mučka, M.: *Webová aplikácia na vzdialenú správu systému Fitcrack*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2018.
- [7] Myers, G. J.; Badgett, T.; Sandler, C.: *The art of software testing*. Hoboken, New Jersey: Wiley, třetí vydání, 2012, ISBN 978-1-118-03196-4.
- [8] Patton, R.: *Testování softwaru*. Praha: Computer Press, vyd. 1 vydání, 2002, ISBN 80-7226-636-5.
- [9] Python Software Foundation: *Unit testing framework*. 2018, [Online; navštívené 14.1.2018].
URL <https://docs.python.org/3.6/library/unittest.html>
- [10] Smrčka, A.: *Testování a dynamická analýza*. FIT VUT v Brně, [Slidy k přednáškám].
- [11] Spillner, A.; Linz, T.; Schaefer, H.: *Software Testing Foundations*. O'Reilly, Čtvrté vydání, 2014, ISBN 1937538427.
- [12] University of California: *Overview of BOINC*. University of California, 2018, [Online; navštívené 14.1.2018].
URL <https://boinc.berkeley.edu/trac/wiki/BoincIntro>

Prílohy

Príloha A

Obsah CD

Priložené CD obsahuje:

- `test_generator.py` – testovacia sada modulu Generator,
- `test_assimilator.py` – testovacia sada modulu Asimilator,
- `test_runner.py` – testovacia sada modulu Runner,
- `test_api.py` – testovacia sada pre API,
- `hashcat_mock.py` – dvojník nástroja Hashcat, používaný pri testovaní,
- `hashcat_parsers.py` – funkcie a objekty používané pri syntaktickej analýze argumentov pre nástroj Hashcat,
- `api_response_models.py` – obsahuje funkcie, ktoré konvertujú údaje do formátu json, tak ako to robí API,
- `config.py` – konfiguračný súbor obsahujúci informácie o databáze, cestám k súborom a iné,
- `fc_test_library.py` – obsahuje definície funkcií a tried používaných pri testovaní
- `main.py` – postupne spúšťa všetky testovacie sady, ako to popisuje kapitola [7.1](#)
- `database/` – balíček obsahujúci súbory pracujúce s databázou, konkrétne:
 - `models.py` – obsahuje mapovanie objektov frameworku SQL Alchemy na tabuľky databázy,
 - `service.py` – funkcie pracujúce s týmito objektmi.
- README – pokyny ako nainštalovať a spúšťať testy.