VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ASOCIAČNÍ ÚTOK S NÁSTROJEM HASHCAT V DISTRIBUOVANÉM PROSTŘEDÍ

ASSOCIATION ATTACK WITH HASHCAT IN A DISTRIBUTED ENVIRONMENT

SEMESTRÁLNÍ PROJEKT
TERM PROJECT

AUTOR PRÁCE **LUKÁŠ WAGNER**
AUTHOR

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

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

147179

Ústav: Ústav informačních systémů (UIFS)
Student: **Wagner Lukáš, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Počítačové sítě

Název: **Asociační útok s nástrojem hashcat v distribuovaném prostředí**
Kategorie: Paralelní a distribuované výpočty
Akademický rok: 2022/23

Zadání:

1. Seznamte se s nástrojem Hashcat a nastudujte, jak funguje asociační útok (association attack). Prozkoumejte podobnost s režimem "single crack mode" v nástroji John the Ripper.
2. Seznamte se s architekturou a implementací systému Fitcrack.
3. Po konzultaci s vedoucím navrhněte rozšíření systému Fitcrack, které umožní realizovat asociační útok v distribuovaném prostředí.
4. Navržené řešení implementujte a otestujte.
5. Navrhněte sadu experimentů, které srovnají vaše řešení s existujícími typy útoků. Zaměřte se na faktory jako výkon, doba výpočtu, režie, škálovatelnost apod.
6. Experimenty realizujte a zhodnoťte dosažené výsledky.

Literatura:

- MARECHAL, Simon. Advances in password cracking. *Journal in computer virology*, 2008, 4.1: 73-81.
- HRANICKÝ Radek. Digital Forensics: The Acceleration of Password Cracking. *Ph.D. thesis*. Faculty of Information Technology, Brno University of Technology. 2022.
- HRANICKÝ Radek, ZOBAL Lukáš, VEČEŘA Vojtěch, MÚČKA Matúš, HORÁK Adam, BOLVANSKÝ Dávid a ŽENČÁK
- Tomáš. The architecture of Fitcrack distributed password cracking system, version 2. FIT-TR-2020-04, Brno: Fakulta Informačních technologií VUT v Brně, 2020.
- HRANICKÝ Radek, ZOBAL Lukáš, RYŠAVÝ Ondřej and KOLÁŘ Dušan. Distributed Password Cracking with BOINC and Hashcat. *Digital Investigation*, vol. 2019, no. 30, pp. 161-172. ISSN 1742-2876.

Při obhajobě semestrální části projektu je požadováno:
Body 1 až 3

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

Vedoucí práce: **Hranický Radek, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1.11.2022
Termín pro odevzdání: 17.5.2023
Datum schválení: 31.10.2022

Abstrakt

Projekt Fitcrack je distribuovaný systém pro lámání kryptografických hashů vyvíjený na FIT VUT. K lámání hesel je na výpočetních jednotkách používán nástroj Hashcat. Tento nástroj přidal v roce 2020 nový mód útoku nazývaný asociační útok. Tento útok je založený na znalosti pravděpodobného hesla, které je během útoku mnohonásobně upravováno. Cílem práce je návrh a implementace rozšíření projektu Fitcrack umožňující použití asociačního útoku a jeho rozdělení výpočetní práce v tomto distribuovaném prostředí. Asociační útok vyžaduje úpravu metod distribuce běžně používaných ostatními útoky, které jsou v práci také navrženy a implementovány. Funkcionalita implementace je později experimentálně ověřena a je učiněn závěr

Abstract

The Fitcrack project is a distributed system for cracking cryptographic hashes developed at FIT BUT. The Hashcat tool is used to crack passwords on the computational units. This tool added a new attack mode in 2020 called an association attack. This attack is based on knowledge of a likely password, which is extensively modified during the attack. The goal of this work is to design and implement an extension to the Fitcrack project, which enables the use of the association attack and solves its workload distribution in this distributed environment. Association attack requires modification of distribution methods used by other attacks. Such new methods are proposed and implemented. Implementation is later experimentally verified and conclusion is drawn.

Klíčová slova

Hashcat, asociační útok, distribuovaný systém, lámání hesel, Fitcrack

Keywords

Hashcat, association attack, distributed system, hash cracking, Fitcrack

Citace

WAGNER, Lukáš. *Asociační útok s nástrojem hashcat v distribuovaném prostředí.* Brno, 2023. Semestrální projekt. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Hranický, Ph.D.

Asociační útok s nástrojem hashcat v distribuovaném prostředí

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Radka Hranického Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Lukáš Wagner
17. května 2023

Poděkování

Chtěl bych poděkovat vedoucímu práce panu Ing. Radkovi Hranickému Ph.D. za jeho pomoc při vzniku práce. Také bych chtěl poděkovat své rodině, kamarádům a jedné kočce za psychickou podporu a kontrolní čtení práce.

Obsah

Kapitola 1

Úvod

Nástroj Hashcat je projekt s otevřeným zdrojovým kódem zaměřený na lámání kryptografických hashů. V roce 2020 nástroj přidal nový mód útoku s názvem „Asociační útok“ (*Association Attack*). Tento nový typ útoku je založen na jednom z módů nástroje John the Ripper, který byl předchůdcem nástroje Hashcat. Princip útoku je založen na znalosti pravděpodobného hesla a jeho následných úpravách. Pravděpodobné heslo představuje jedinou nápovědu k danému hashi, lze tedy využít mnohem více úprav v určeném čase, než při použití více pravděpodobných hesel. Díky této strategii je možné pravděpodobné heslo ověřit vůči velkému množství běžných lidských úprav jako je přidávání číslic a speciálních znaků na konec, použití velkých písmen, nebo odstranění samohlásek.

Projekt FitCrack vyvíjený na VUT FIT využívá programu HashCat pro distribuované lámání hesel. Projekt poskytuje webové grafické uživatelské rozhraní a řídí distribuci práce mezi více výpočetních jednotek. K tomuto účelu je použit komunikační systém BOINC, jenž umožňuje výpočetní síti komunikovat přes internet.

Cílem práce je rozšíření projektu Fitcrack implementující použití asociačního útoku a vhodná distribuce práce mezi výpočetními jednotkami. Součástí rozšíření jsou vhodné uživatelské rozhraní, procesy serveru zodpovídající za distribuci a procesy výpočetních jednotek, které zpracovávají vstupy a obstarávají samotné lámání hashů. Důvodem vzniku rozšíření je výše zmíněné přidání asociačního útoku a současná neexistence distribuovaného řešení.

První kapitola po úvodu 2 se věnuje teorii kryptografického hashe, jeho použití a vlastnostem, a popisuje některé existující útoky. Další kapitola 3 se zabývá průzkumem existujících řešení problému distribuce lámacích úloh. Zmiňuje některá řešení založená na nástroji John the Ripper i na nástroji Hashcat. Některá řešení jsou komerční a jiná s otevřeným kódem. Velká část těchto řešení byla již vývojáři opuštěna. V následující kapitole 4 je přiblížena architektura projektu Fitcrack za účelem objasnění funkčnosti vnitřních procesů, jež se přidání nového útoku může dotknout. Další kapitola 5 se zabývá návrhem implementace nového útoku, distribucí zátěže dělením hashů a dělením pravidel úprav. Tato kapitola také obsahuje sekci o testování asociačního módu v programu Hashcat za účelem určení efektivity zmíněných typů distribuce. V kapitole 6 je navržené řešení implementováno a jsou popsány jednotlivé metody implementace spolu s příklady kódu. Poslední kapitola 7 se zabývá testováním implementovaného řešení. Začíná popisem hardware a software a návrhem testů spolu s jejich cílem. Věnuje se zejména škálovatelnosti a porovnání metod distribuce, okrajově také porovnává asociační útok s útokem slovníkovým. Nakonec jsou testy zhodnoceny a je učiněn závěr. Přílohy poté obsahují obsah přiloženého média (A) a zjednodušený popis struktury projektu s upravovanými soubory (Příloha B).

Kapitola 2

Relevantní pojmy z kryptologie

Kryptografické hashe jsou důležitou součástí moderní počítačové bezpečnosti. Představují „otisky“ pevné délky dat jako jsou hesla nebo celé soubory. Jejich cílem je vytvořit metodu verifikace správnosti dat založenou na prakticky nemožné reverzibilitě. K prolomení takového hashe dochází zejména hádáním vstupu hashovací funkce a jeho bezpečnost je tedy často založená na složitosti vstupu.

2.1 Kryptografický hash a hesla

Aplikací hashovací funkce na text libovolné délky vzniká blok symbolů pevné délky nazývaný kryptografický hash. Kryptografická hashovací funkce musí splňovat tři podmínky, aby byla považována za bezpečnou [14, 4]:

- Pro daný hash je výpočetně nezvládnutelné nalézt jeho původní vstup.
- Pro daný vstup je výpočetně nezvládnutelné nalézt jiný vstup se stejným hashem.
- Je výpočetně nezvládnutelné nalézt jakékoliv dva rozdílné vstupy se stejným výsledným hashem.

Uživatelská hesla jsou ukládána v podobě svého hashe právě z důvodu těchto vlastností hashovacích funkcí. Dojde-li k úniku informací, nemá pachatel okamžitě k dispozici heslo, ale pouze jeho hash. Je tedy nucen pro jeho získání hash „prolomit“. Lámání hashů je možné protože uživatelé často volí zapamatovatelná hesla, jež bývají krátká nebo následují nějaký vzor a je tedy možné heslo „uhádnout“. Tato problematika je dále rozvedena v následující sekci 2.2.

Za účelem ztížení prolomení hesla jsou používány mechaniky kryptografické soli a pepře [4, 14, 5, 8]. Kryptografická sůl je založená na jednoduchém principu připojení textu k heslu. Sůl je bezpečně generována a ukládána spolu s hashem, často tedy dochází i k jejímu úniku. Účelem soli je zabránění prolomení hesla hledáním již známých hashů například použitím duhové tabulky (rainbow table) [10], takto také zabraňuje detekci duplicitních hesel. Pepřem lze označit rozšíření konceptu soli, který se obecně zakládá na tom že pepř není uložen s heslem. Pepř je tedy buď uložen na bezpečném místě mimo databázi, nebo je jej zapotřebí pokaždé znovu objevit. Princip znovuobjevování pepře nutí při verifikaci správného hesla iterovat přes množinu možných pepřů a prodlužuje tedy čas potřebný k autentizaci, takto ovšem zároveň značně prodlužuje čas potřebný k celkové verifikaci hesel tvořených útočníkem.

2.2 Lámání hesel v programu Hashcat

Lámáním hesel se rozumí tvorba kandidátních hesel takzvaným generátorem hesel a jejich následnou validací. Každé heslo je validováno aplikací funkce kryptografického hashe, která musí být pro tento účel útočníkovi známá, a porovnáním vygenerovaného hashe se získaným hashem původního hesla. Jednotlivé typy útoků na hash se liší v použitém generátoru. Lámání končí úspěšně nalezením správného hesla nebo neúspěšně vyčerpáním prostoru kandidátních hesel. Jelikož vyzkoušet všechna možná hesla je časově nereálné, jak přibližuje tabulka 2.1, využívá se při lámání generátorů zohledňujících některé charakteristiky, které lze nalézt v lidmi tvořenými hesly [7, 15]. Pro jednoduchost zapamatování lidé používají srozumitelná slova často nějak pro ně významná. Pokud je po uživateli vyžadováno použití speciálních znaků a čísel, je více pravděpodobné, že tyto znaky umístí na konec nebo se uchýlí k takzvanému „leetspeak“, ve kterém jsou písmena nahrazována čísly a symboly jim podobnými [15].

Jedním z nejjednodušších útoků[1] je útok silou. Jedná se o postupné abecední zkoušení všech možných kombinací symbolů. Výhodou tohoto útoku je zaručené nalezení hesla. Ovšem díky velkému množství hesel nutných validovat vůči hashi se jedná o velmi časově náročný postup. Je tedy reálně možné takto lámat hesla pouze do určité délky, běžně se uvádí zhruba do délky osmi znaků, ale toto číslo je závislé na dostupné výpočetní síle a na použité hashovací funkci. Tabulka 2.1 přibližuje dobu nalezení správného hesla útokem silou.

Počet symbolů	**Pouze číslice**	**Alfanumerické**	**Speciální symboly**
5	<1 s	<1 s	<1 s
6	<1 s	1 s	5 s
7	<1 s	1 m	6 m
8	<1 s	1 h	8 h
9	<1 s	3 dny	3 týdny
10	<1 s	7 měsíců	5 let
11	<1 s	3 roky	34 let

Tabulka 2.1: Tabulka udávající trvání útoku silou na různě dlouhá hesla a různé sady symbolů pro hashovací funkci MD5 realizovaného na grafické kartě RTX2080. Data převzata z webu `www.hivesystems.io`.

Pokročilejší verzí útoku silou je útok s maskou, který využívá omezující pravidla na množinu symbolů pro každou pozici v hledaném hesle. V nástroji Hashcat existují předdefinované běžné množiny symbolů, které jsou znázorněny v tabulce 2.2. Při spuštění lze taktéž dodefinovat až čtyři vlastní množiny. Ke tvorbě vlastních množin lze použít jednoduše chtěné znaky i zástupné symboly jiných množin, nebo také je možno symboly načíst ze souboru hashcat-charset (soubory s příponou `hcchr`).

Dalším typem útoku, založeným na lidské vlastnosti tvořit zapamatovatelná hesla, je slovníkový útok. Slovník je množina potenciálních hesel verifikovaných vůči hashi, kterou lze vytvořit z relevantních informací. Existují slovníky založené na počtu výskytů prolomených hesel ze skutečných úniků informací, jako je např slovník RockYou[2].

Použití komolících pravidel („mangling rules“) je způsob rozšíření slovníkových útoků o jednoduché změny. Tento postup byl zaveden v nástroji John the Ripper a následně přejat

[1] `https://hashcat.net/wiki/doku.php?id=hashcat#supported_attack_modes`
[2] `https://en.wikipedia.org/wiki/RockYou#Data_breach`

Maska	Symboly
?l	abcdefghijklmnopqrstuvwxyz
?u	ABCDEFGHIJKLMNOPQRSTUVWXYZ
?d	0123456789
?h	0123456789abcdef
?H	0123456789ABCDEF
?s	«space»!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
?a	?l?u?d?s
?b	0x00 až 0xff
?1	vlastní množina nastavená přepínačem -1
?2	vlastní množina nastavená přepínačem -2
?3	vlastní množina nastavená přepínačem -3
?4	vlastní množina nastavená přepínačem -4

Tabulka 2.2: Zástupné symboly znakových množin v programu Hashcat

pro Hashcat. Komolící pravidla definují jednoduché změny, jako jsou třeba převod do malých písmen, duplikování znaků a řetězců, otáčení řetězců a jiné. Hashcat popisuje útok na bázi pravidel jako jeden z nejefektivnější a nejflexibilnějších způsobů útoku, ovšem za cenu komplikované tvorby pravidel přirovnávaných k programovacímu jazyku [2]. Všechna pravidla používané nástrojem John the Ripper jsou kompatibilní s enginem pravidel v Hashcatu. Seznam pravidel Hashcat ovšem dále rozšiřuje o pravidla vlastní.

Komolící pravidla přijímá Hashcat v souboru, ve kterém každý řádek značí sekvenci úprav vedoucí k jednomu kandidátnímu heslu. Pro každé kandidátní heslo ze slovníku vznikne aplikováním všech pravidel množina nových kandidátních hesel stejné velikosti, jako je počet řádků v souboru pravidel. Pro slovník o velikosti m a soubor pravidel s n řádky vznikne $m \times n$ kandidátních hesel, je tedy zřejmé, že pravidla zvyšují výpočetní náročnost úlohy.

Pravidla jsou obvykle tvořena za účelem napodobení zvyků uživatelů při tvoření hesel. Je tedy vhodné aby pravidla vznikala z analýzy skutečných hesel uživatelů pro daný účel. Častými zvyky bývá například zakončení hesla číslem nebo velké první písmeno, jak bylo uvedeno v úvodu této sekce 2.2.

2.3 Asociační útok

Funkce asociačního útoku v nástroji hashcat je odvozen od módu *single* nástroje John the Ripper[3]. Mód single využívá nápovědy v podobě polí GECOS (General Comprehensive Operating System) tak, jak jsou použity v souboru `/etc/passwd` na Unixových systémech. Tato pole obsahují informace o uživateli zejména jeho uživatelské jméno nebo také telefonní číslo nebo emailovou adresu. Pole GECOS jsou po načtení rozdělena podle čárek a umožňují takto používat více nápověd pro jeden hash. Vstupem módu single je tedy pole nápověd a dvojtečkou oddělený hash [19].

Princip útoku je založen na lidském návyku používat pro tvorbu hesla své uživatelské jméno a jiné lehce zapamatovatelné osobní informace. V nástroji John the Ripper využívá mód single nápovědu s množstvím, v konfiguračním souboru předdefinovaných, komolících

[3]Příspěvek o novém módu útoku na fóru: https://hashcat.net/forum/thread-9534.html

pravidel pro tvorbu „virtuálního slovníku“ vysoce relevantních kandidátních hesel pro každý hash. Mangling pravidla jsou setřízená tak, že jsou prvně provedeny jednoduché změny a následně složitější operace seřazeny podle počtu úspěšně prolomených hesel[4].

Nástroj Hashcat implementuje asociační mód útoku odlišně. Nástroj vyžaduje dva vstupní soubory, podobně jako u slovníkového útoku, jeden s hashy a druhý s kandidátními hesly. Každý hash musí mít kandidátní heslo v komplementárním souboru na stejném řádku, na kterém se nachází ve svém souboru. Jinými slovy počet hashů je roven počtu slov v poskytnutém slovníku. Na rozdíl od Nástroje John the Ripper Hashcat nepoužívá v základu žádná mangling pravidla a jejich použití je ponecháno na uživateli použitím přepínače `-r` a poskytnutím souboru obsahující tato pravidla[5].

John the Ripper v tomto módu útoku používá již nalezená hesla jako jednoduché kandidáty pro ostatní hashe se stejnou kryptografickou solí, je tedy schopen rozluštit více hesel při větším množství vstupních hashů. Nástroj Hashcat tuto funkcionalitu nemá, program se tedy chová jako dávka jednoduchých slovníkových útoků s jediným kandidátem. Tento přístup zjednodušuje paralelizaci a komunikaci mezi distribuovanými výpočetními jednotkami.

[4] Pravidla lze najít spolu s komentáři vysvětlující jejich pořadí v konfiguračním souboru „john.conf“.

[5] Implementace módu je vysvětlena v příspěvku na fóru dostupném zde: https://hashcat.net/forum/thread-9534.html

Kapitola 3

Rozbor existujících řešení

Jelikož lámání hesel je s jejich délkou exponenciálně složitý úkol, není nikdy dost výpočetního výkonu [18]. A jelikož distribuce těchto úkolů je konceptuálně jednoduchá záležitost, jak je probráno v sekci 4.4, je samozřejmé, že existují řešení těchto úkolů, která této vlastnosti využívají. Bohužel podstatná část těchto projektů byla s časem opuštěna. Nemalá část projektů je také komerčních a tudíž jejich implementace není veřejná. Mnoho projektů zakládá svůj výpočet na nástroji John the Ripper[1] [18], který byl předchůdcem nástroje Hashcat [2]. John the Ripper je za účely distribuce výpočtu již navržen s implementací „Messege Passing Interface“ [16, 18]. Aplikace založené na programu John the Ripper jsou například [11, 18]:

- DJohn[2] – architektura klient server, opuštěn,
- dnetj – heterogennější než MPI, opuštěn,
- Clortho[3] – architektura klient server,
- John Crumpacker Cracker – disertační práce představující použití BOINC [6].

Komerční nástroje představují plně funkční řešení v dostatečném rozsahu, aby byly považovány za víc než jen vědecký experiment s jasným zaměřením a omezenou funkčností. Jak bylo ovšem zmíněno, komerční aplikace distribuovaného lámání hashů jsou stěží použitelné pro analýzu řešení vzhledem k jejich neveřejné implementaci. Jedná se například o programy:

- Passware Kit Forensic – Passware je komerční program pro získávání hesel, podporuje lámání přes 350 různých druhů šifrovaných souborů, ovšem nepodporuje lámání samotných hashů. V roce 2010 byl software Passware rozšířen o možnost distribuovaného lámání s architekturou klient server fungujícími na Windows i Linux distribucích. [20],
- Hashstack[4] – byla komerční aplikace od firmy Terahash založená na Hashcatu implementující distribuci zátěže použitím vlastního systému „Magistos Cluster Orchestrator“. Aplikace poskytovala plnou podporu GPU akcelerace výpočtu všech přes 375-ti formátu hashů v 6-ti módech útoku.

[1] https://openwall.info/wiki/john/parallelization
[2] https://sourceforge.net/projects/djohn/
[3] https://github.com/ccdes/clortho
[4] https://web.archive.org/web/20201108092329/https://terahash.com/#hashstack

Projekt Fitcrack, jehož rozšířením je tato diplomová práce, používá programu Hashcat pro lámání hesel. Jelikož implementace asociačního útoku se liší mezi nástroji Hashcat a John the Ripper, které jej implementují, je důležité se zaměřit na nástroje, které používají právě Hashcat. Nástroje založené na programu Hashcat s otevřeným zdrojovým kódem:

- WPA-SEC[5] – distribuované lámání WPA klíčů s použitím Hashcat nebo John the Ripper, založené na dobrovolnících,
- Kracken (Cracken)[6] – distribuovaný slovníkový útok s webovým serverem a klienty, kontejnerizace použitím Dockeru,
- CrackQ[7] – Web založený na API, kontejnerizace Dockerem, výpočetní fronty, nepodporuje asociační útok,
- Cracklord[8] – Webová aplikace umožňující distribuovat výpočet implementovaných nástrojů jako jsou Hashcat a John the Ripper, bohužel nepodporuje asociační útok,
- Hashtopolis[9] – Je aplikace distribuující zátěž Hashcatu použitím PHP serveru a Python klientů, z této aplikace Fitcrack částečně vychází.

3.1 CrackQ

CrackQ je projekt umožňující distribuci lámacích úkolů pro program Hashcat použitím front. Projekt se dělí na API server, webovou aplikaci a klientskou aplikaci v jazyce Python. Instalace je zjednodušená použitím kontejnerů Docker s nástrojem Docker-compose. K přenosu souborů a dat se využívá databáze Sqlite. Sever poskytuje REST API použitím knihovny Flask a vytváří model nad databází knihovnou SQLAlchemy a umožňuje zasílání emailů s oznámeními. Klienti využívají wrapper knihovnu libhashcat kolem programu Hashcat a nepoužívají tedy příkazový řádek. Fronta úkolů umožňuje řadit výpočetní úkoly a jednoduše přidělovat práci právě volným klientům. Projekt CrackQ ovšem neřeší dělení práce jednotlivých úkolů a jednotky ve frontě tedy bere jako celky.

Fitcrack používá velmi podobnou serverovou architekturu i se stejnými knihovnami, ovšem pro distribuci využívá další serverovou aplikaci implementující BOINC a dělení jednotlivých úkolů. Projekt CrackQ bohužel také neimplementuje asociační útok i přesto, že neřeší dělení úloh, a bylo by tedy jednoduché jeho přidání.

3.2 Hashtopolis

Hashtopolis je projekt s otevřeným kódem určený pro distribuované lámání hashů s programem Hashcat. Projekt využívá architektury klient-server, kde klienti představují výpočetní jednotky. Server je napsán jako monolitická aplikace v jazyce PHP. Server umožňuje správu úloh, uživatelů a souborů, jako jsou hashe, pravidla a slovníky. Klienti jsou aplikace v jazyce Python (Dříve také implementace v jazyce C#), které stahují data ze serveru přes protokol HTTPS a spouštějí Hashcat se specifikovanými parametry. Projekt využívá pro ukládání souborů a dat databázi MySQL [11, 9].

[5] https://wpa-sec.stanev.org/
[6] https://github.com/arcaneiceman/kraken/
[7] https://github.com/f0cker/crackq/
[8] http://jmmcatee.github.io/cracklord/
[9] https://github.com/hashtopolis

I když Fitcrack s Hashtopolis sdílí mnoho konceptů, jejich filosofie je odlišná. Fitcrack odděluje frontend webové aplikace od API za účelem možného vývoje jiných aplikací využívajících funkcionality Fitcracku. Zároveň Fitcrack odebírá nízkoúrovňovou kontrolu uživateli a přesně specifikuje jaké parametry lze jak nastavit pro konkrétní útoky a jaké soubory nahrát [11]. Díky této vlastnosti lze také implementovat distribuci specifickou asociačnímu útoku.

3.3 Implementace asociačního útoku

Přestože Hashcat využívá velké množství různých nástrojů a aplikací, žádná aplikace nepodporuje asociační mód útoku. Aplikace Hashtopolis teoreticky umožňuje použití asociačního útoku, jelikož uživatel specifikuje argumenty spouštění Hashcatu na nízké úrovni, ovšem distribuce zátěže specifická pro asociační útok není implementována. Implementace Asociačního útoku do distribuované aplikace je přínos projektu Fitcrack, ke kterému zatím neexistuje žádná alternativa. Kvůli této vlastnosti nelze analyzovat jiná řešení, ale je zapotřebí systém distribuce nově navrhnout. Při tomto návrhu bude vycházeno z možností a filosofie projektu Fitcrack a jeho již implementovaných útoků.

Kapitola 4

Projekt Fitcrack

Fitcrack[1] je distribuovaný systém pro lámání hesel vytvořený skupinou NES@FIT na Fakultě Informačních Technologií Vysokého Učení Technického v Brně [1]. Fitcrack zaručuje podporu škály hashů a útoků díky vnitřní integraci s programem Hashcat. Projekt je připraven na použití různorodých hardwarových jednotek použitím technologie OpenCL [17], která umožňuje psát a spouštět programy pro CPU, GPU, DSP, FPGA a jiné kompatibilní procesory.

4.1 Funkcionalita

Vtupem programu Hashcat je seznam lámaných hashů v textové podobě. Tento seznam je předáván souborem, který na každém řádku obsahuje právě jeden hash a jeho sůl, pokud je použita. Při použití nástroje Hashcat musí uživatel tento seznam vytvořit manuálně. Tento přístup je dostupný i v projektu Fitcrack, buď nahráním takovéhoto již existujícího souboru, nebo jeho tvorbou přímo ve formuláři uživatelského rozhraní. Projekt Fitcrack dále poskytuje abstrakci pro lámání hesel z různých zdrojů, jako jsou archivy ZIP, RAR a 7z nebo dokumenty Microsoft Officce a PDF, popřípadě také diskové oddíly šifrované použitím nástroje VeraCrypt[2] [13]. Tyto soubory systém umožňuje nahrát v celku a lámaný hash je vyextrahován použitím podsystému **XtoHashcat**, jenž je součástí implementace projektu Fitcrack. Podsystém **XtoHashcat** je podrobně popsaný v technické zprávě [13].

Tento seznam hashů je rozeslán mezi přidělené (popřípadě vybrané) výpočetní uzly. Každý výpočetní uzel běžně dostane všechny lámané hashe a část prostoru zkoušených hesel. Na každém uzlu je spouštěn Hashcat ve specifikovaném módu útoku pro výpočet přidělené části.

Výstupem výpočtu jednotlivých částí jsou nalezené páry hashů a jejich hesel, pokud jsou nějaké nalezeny. Tyto výstupy zpracovává podsystém **Assimilator**, který výsledky výpočtu postupně ukládá do databáze. Díky tomuto postupu je možné zobrazovat prolomené hashe ve webovém uživatelkém prostředí aplikace během výpočtu.

Projekt v současné době podporuje následující útoky nastavitelné z uživatelského prostředí. Tyto útoky odpovídají možnostem programu Hashcat.

- **Dictionary** – slovníkový útok, potřebuje specifikovat slovník zkoušených hesel
- **Combination** – kombinační slovníkový útok kombinující slova ze dvou slovníků

[1] https://fitcrack.fit.vutbr.cz/
[2] https://veracrypt.fr/

- **Brute force** – útok silou, je možné specifikovat masky, znakové množiny a markovské řetězce
- **Hybrid wordlist-mask** – kombinační útok připojující útok silou za slova ze slovníku
- **Hybrid mask-wordlist** – kombinační útok připojující slova ze slovníku za hesla generovaná útokem silou
- **PRINCE** – slovníkový kombinační útok chytře kombinující 1 až N slov slovníku na základě statistiky délek slov obsažených ve slovníku, nejedná se přímo o mód Hashcatu, vyžaduje podsystém tvořící slovník
- **PCFG** – útok založený na pravděpodobnostních gramatikách, využívá pravděpodobné následující symboly z trénovaného slovníku, nejedná se přímo o mód Hashcatu, vyžaduje podsystém generující hesla [12]

4.2 Architektura

Síťová architektura se skládá ze serverové a klientské části. Serverová část je zodpovědná za kontrolu a dělení úkolů mezi klienty a sama se na lámání kryptografických hashů nepodílí. Každý klient má k dispozici jednu nebo více výpočetních jednotek kompatibilních s technologií OpenCL [17]. Pokud je zastoupeno více typů OpenCL zařízení, například CPU i GPU současně, lze síť nazývat hybridní [13].

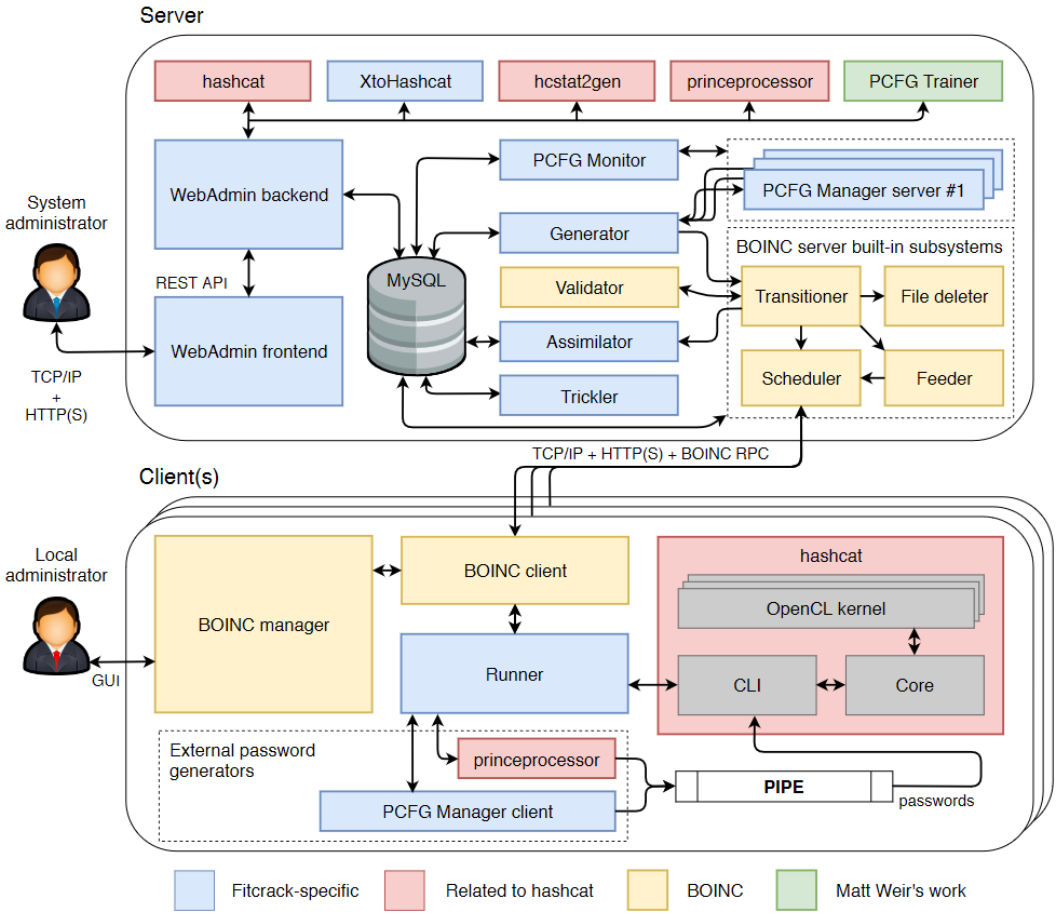
Komunikace mezi klienty a serverem probíhá použitím TCP/IP sítě, což umožňuje projektu pracovat přes internet a není tedy nutné, aby se klienti nacházeli ve stejné lokální síti, mohou se tedy nacházet i na geograficky vzdálených místech. Pro komunikaci je použit framework BOINC (Berkley Open Infrastructure for Network Computing) [3] a jeho „scheduling server protocol“ fungující přes HTTPS. Celá architektura projektu rozdělena na server a klienty i s jednotlivými podsystémy a vyznačenou komunikací mezi těmito podsystémy je zobrazena na obrázku 4.1.

4.2.1 WebAdmin

WebAdmin je název aplikace zodpovědné za uživatelskou správu projektu Fitcrack. WebAdmin se dělí na frontend a backend. Frontend je založen na javascript frameworku Vue[3] a jeho doplňkovém frameworku Vuetify[4] a poskytuje příjemné uživatelské prostředí pro ovládání projektu. Backend je napsán v jazyce Python za použití knihovny Flask[5] a implementuje Apache web server a REST API, přes které komunikuje s frontendem. Backend dále provádí operace nad databází MySQL pomocí knihovny SQLAlchemy[6], do které jsou ukládána relevantní data pro lámání. Aplikace podporuje možnost existence více uživatelů s rozdílnými rolemi za účelem upřesnění práv.

Frontend poskytuje uživateli možnost zadávat a spravovat lámací úlohy. Zobrazuje informace o stavu těchto úloh díky podsystému trickler a uvádí také informace o jednotlivých jednotkách jako jejich zatížení a paměťové místo. Frontend také zajišťuje rozhraní pro nahrávání relevantních souborů, jako jsou například soubory s pravidly, hashi, znakovými

[3] https://vuejs.org/
[4] https://vuetifyjs.com/
[5] https://flask.palletsprojects.com/
[6] https://www.sqlalchemy.org/



Obrázek 4.1: Architektura projektu Fitcrack, převzato z [13]

sadami nebo maskami. Tvorba nových úloh je zprostředkována použitím formulářů umožňujících podrobné nastavení a případné uložení nastavení pro opakované použití. Možnosti nastavení jsou založeny na parametrech nástroje Hashcat a možnostech distribuce.

Backend aplikace kromě databáze komunikuje s několika nástroji potřebnými pro různé účely [13], které jsou zobrazeny v obrázku 4.1 s naznačenou komunikací. Jedná se o:

- Hashcat - za účely ověřování formátu hashů a výpočtu počtu kandidátních hesel
- XtoHashcat - za účelem extrakce hashů ze souborů
- hcstat2gen - za účelem generování statistik pro Markovské řetězce
- princeprocessor - za účelem výpočtu počtu kandidátních hesel pro útok PRINCE
- pwd_dist - pro spočtení statistiky délek kandidátních hesel
- PCFG Trainer - pro tvorbu pravděpodobnostních gramatik

4.2.2 Generator

Generátor je proces na straně serveru zodpovědný za rozdělení úlohy na bloky mezi výpočetní jednotky podle výpočetního výkonu dané jednotky a implementované strategie pro

daný útok. Výpočetní výkon jednotek je měřen testovacími bloky rozeslanými každé jednotce vždy na začátku úlohy za účelem zjištění výkonu pro daný hash. Testovací bloky mohou být mířeny pouze na jeden formát hashe nebo na všechny podporované. Test na všechny formáty se nazývá kompletní a je vždy posílán jednotce při jejím připojení do sítě pro získání přehledu o jejím výkonu.

Úkolem generátoru je zajišťovat aby každý úkol existoval alespoň v jednom bloku a zároveň, aby každá přidělená jednotka měla k dispozici blok je-li nějaký dostupný. Pokud je tomu možné, jsou pro každou výpočetní jednotku generovány dva bloky, aby měla jednotka druhý blok okamžitě připravený ke zpracování po dokončení prvního bloku a tak se snížila režie úlohy.

Druhotnými úkoly generátoru je vypořádávat se s poruchami a odpojenými výpočetními jednotkami a likvidace rozpracované úlohy ukončené uživatelem. Pokud dojde k poruše, generátor vytvoří kopii porouchaného nebo nedokončeného bloku a označí ji pro opětovné zpracování. Pokud uživatel předčasně ukončí úlohu generátor informuje o tomto rozhodnutí jednotky a vrací se zpět do připraveného stavu.

Generátor zodpovídá za tvorbu souborů potřebných na vstupu jednotlivých jednotek. Pokaždé jsou posílány soubory s daty a nastavením obsahující informace o útoku. Některé útoky mohou vyžadovat i jiné soubory jako třeba slovníky nebo pravidla.

Ke komunikaci se zbytkem serveru používá generátor pouze databázi. Tento přístup je běžný u podprocesů systému BOINC [13].

4.2.3 Runner

Runner je adaptér mezi BOINC systémem a programem Hashcat napsaný v C++. Runner běží na straně klienta a jeho úkolem je z přijatého konfiguračního souboru vytvořit parametry, se kterými následně spouští a monitoruje Hashcat. Pokud útok vyžaduje speciální generátor hesel, je runner zodpovědný za jeho spuštění.

Runner je spouštěn klientem BOINC a vyžaduje přítomnost potřebných vstupních souborů ve stejné složce, ve které se nachází. Některé soubory jsou vyžadovány vždy, některé jsou specifické pro zvolené útoky. Dle [13] se jedná o:

- Vždy vyžadované soubory:
  - `hashcat_files_v510_1.zip` - soubory důležité pro Hashcat,
  - `config` - informace pro runner obsahující argumenty a specifika spuštění,
  - `data` - soubor s lámanými hashi.
- Specifické soubory:
  - `dict1` - slovník kandidátních hesel,
  - `dict2` - druhý slovník kandidátních hesel, například pro kombinační útok,
  - `rules` - pravidla zkomolení,
  - `markov` - markovské statistiky,
  - `preterminals` - preterminály pravděpodobnostních gramatik,
  - `grammar` - pravděpodobnostní bezkontextová gramatika.

Výstup runneru je uložen do souboru `out` ve formátu zpracovatelném serverem, který je popsán dále v 4.2.4. Záznam z běhu se ukládá do souboru `stderr.txt`.

4.2.4 Asimilátor

Asimilátor je proces na straně serveru, který se stará o zpracování výsledků vrácených z výpočetních jednotek. Podle přijatého výsledku může asimilátor změnit stav úlohy v databázi a ukončit výpočet zpracovávaných bloků. Každý typ bloku vrací odlišné informace po ukončení, proto je označen typ bloku v přijatém souboru s výsledky na prvním řádku. Bloky testující výkon jsou označeny b pro jednoduchý test a a pro kompletní. Blok lámací úlohy je označen písmenem n. Na druhém řádku je číslo značící stav výsledku zpracovaného bloku, kde číslo 0 označuje úspěšné ukončení. Chyba je označena stavem vyšším než dva, v takovém případě asimilátor označí blok pro znovuzpracování.

4.2.5 Databáze MySQL

Projekt Fitcrack používá k uložení informací relevantních pro lámání databázi MySQL. Databáze obsahuje dva typy tabulek: Tabulky frameworku BOINC a Tabulky systému Fitcrack. Tabulky vytvořené frameworkem BOINC, které jsou používány k základním funkcím distribuce, jsou upravovány BOINC procesy Transitioner, File deleter, Scheduler a Feeder. Procesy Fitcracku mají do těchto tabulek přístup pouze pro čtení. Tabulky systému Fitcrack jsou vytvořeny při instalaci Fitcracku. Tyto tabulky využívají pro sdílení informací podsystémy WebAdmin, Generator, Assimilator, Validator a Trickler.

Tabulky frameworku BOINC nejsou nijak upravované a jsou tvořeny skriptem `make_project`. Jedná se o tyto tabulky[7]:

- **platform** - cílové platformy pro kompilaci aplikace,
- **app** - aplikace a klient,
- **app_version** - verze aplikací s odkazem na její stažení a kontrolním součtem MD5,
- **user** - uživatelé, jejich e-mail, jméno, heslo a autentikátor,
- **host** - popis výpočetních uzlů přihlášených do sítě,
- **workunit** - výpočetní bloky úlohy, počet jejich zaslaní, selhání, úspěšných dokončení a výsledků,
- **result** - výsledky výpočetních bloků, potřebný čas, návratový kód a stav validace.

Tabulky přidané Fitcrackem:

- **fc_batch** - ukládá dávky úloh a jejich zadavatele,
- **fc_benchmark** - uchovává výsledky testů výpočetních uzlů pro každý hash,
- **fc_bin** - popisuje skupiny výpočetních úloh, které jsou tvořeny za účelem přehlednosti a správy,
- **fc_bin_job** - propojuje úlohy a skupiny,
- **fc_charset** - ukládá informace o souborech s množinami symbolů nahranými uživateli,
- **fc_dictionary** - ukládá informace o uložených slovnících,
- **fc_hash** - ukládá zpracovávané a prolomené hashe,
- **fc_hcstats** - ukládá informace o souborech s Markovskými statistikami,
- **fc_host** - popisuje všechny aktivní výpočetní uzly, které právě pracují na nějaké úloze,
- **fc_host_activity** - popisuje všechny právě plánované úkoly pro jednotlivé uzly,
- **fc_host_status** - popisuje on-line status uzlů,
- **fc_job** - popisuje všechny informace o úloze se zadaným módem útoku,

[7] https://boinc.berkeley.edu/trac/wiki/DataBase

- **fc_job_dictionary** - přiřazuje úloze více slovníku, například pro kombinační útok,
- **fc_job_graph** - popisuje míru postupu výpočtu zadané úlohy,
- **fc_job_status** - uchovává změny ve stavech úloh během zpracování,
- **fc_mask** - uchovává definované masky používané při útoku silou,
- **fc_mask_set** - ukládá soubory obsahující množiny masek,
- **fc_notification** - definuje notifikace zobrazované uživateli, například, když je dokončena úloha,
- **fc_pcfg_grammar** - ukládá cesty k souborům obsahující pravděpodobnostní gramatiky,
- **fc_pcfg_preterminals** - ukládá preterminály pravděpodobnostních gramatik za účelem obnovy ze selhání,
- **fc_protected_file** - popisuje uložené zašifrované soubory, ze kterých bude použitím procesu `XtoHashcat` vyjmut hash,
- **fc_role** - umožňuje definovat role uživatelů a jejich pravomoci,
- **fc_rule** - popisuje soubor s uloženými komolícími pravidly,
- **fc_server_usage** - ukládá monitorující data o serveru, jako je jeho vytížení procesoru a paměti nebo současná rychlost přenosu dat,
- **fc_settings** - popisuje nastavení serveru pro tvorbu úloh,
- **fc_template** - ukládá vzory úloh nadefinované uživateli z webového rozhraní,
- **fc_user** - uchovává všechny uživatele s přístupem do systému a jejich role,
- **fc_user_permission** - popisuje pravomoce uživatelů nad jednotlivými úlohami
- **fc_workunit** - rozšiřuje tabulku frameworku BOINC o informace specifické pro Fitcrack, jako je začátek a konec v množině kandidátních hesel.

Důležitou tabulkou pro vytvoření lámací úlohy a její komunikaci mezi serverem a výpočetní jednotkou je tabulka **fc_job**. Tato tabulka obsahuje všechny informace relevantní pro danou úlohu, které jsou za pomoci podsystému **scheduler** rozeslány výpočetním jednotkám spolu s výpočetními bloky úlohy. Tato tabulka je relačně propojena s tabulkami **fc_hash**, **fc_job_dictionary**, **fc_mask**, **fc_workunit**, **fc_pcfg_grammar** potřebnými pro útoky a dalšími tabulkami zajišťujícími kvalitu služby. Popis tabulky **fc_job**:

- **id** - primární klíč,
- **attack_mode** - číselná hodnota popisující zvolený mód útoku,
- **attack_submode** - číselná hodnota dále specifikující mód útoku,
- **hash_type** - číselná hodnota popisující typ hashe podle tabulek nástroje Hashcat[8],
- **status** - stav úlohy, rozlišující mezi připravenou, ukončenou a běžící úlohou a jejich podstavy, jak dále popisuje technická zpráva [13],
- **keyspace** - skutečný počet kandidátních hesel,
- **hc_keyspace** - počet kandidátních hesel bez vnitřní smyčky úprav, korespondující s parametry skip a limit,
- **indexes_verified** - počet zpracovaných kandidátních hesel,
- **current_index** - index kandidátního hesla, od kterého začne další výpočetní blok, v případě kombinačního útoku se jedná o index kandidátního hesla ve druhém slovníku,
- **current_index_2** - index kandidátního hesla v prvním slovníku pro kombinační útok,
- **time** - čas vzniku úlohy,

[8] https://hashcat.net/wiki/doku.php?id=example_hashes

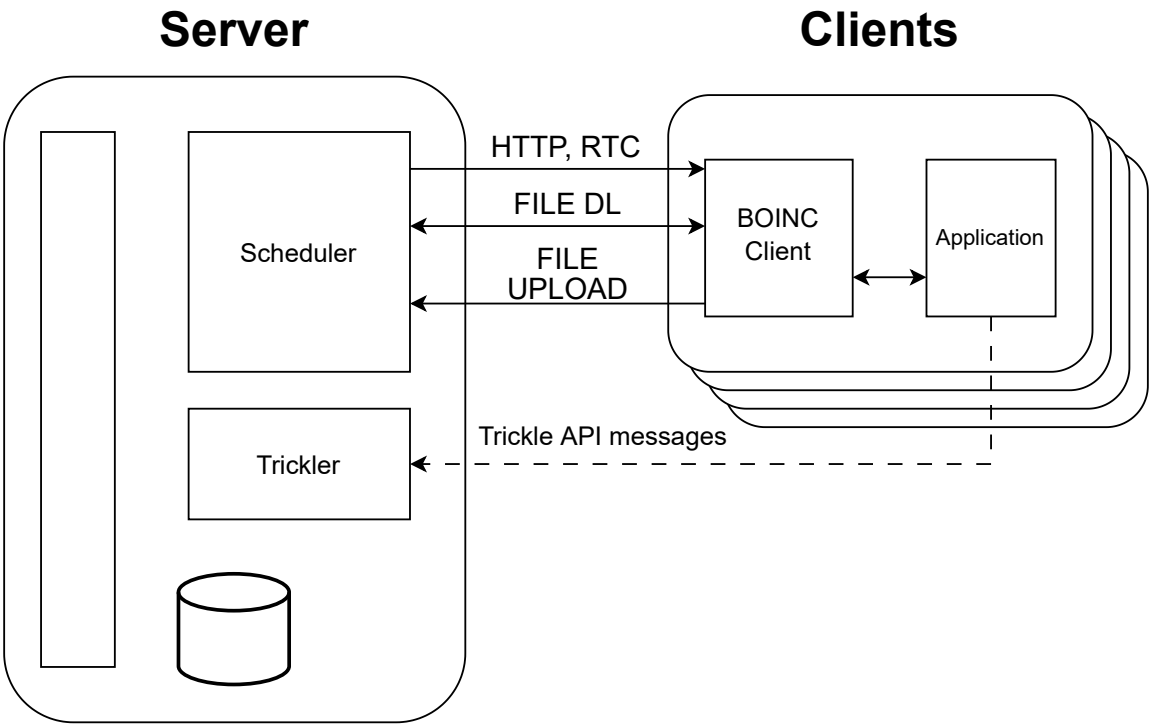
- **name** - uživatelský název úlohy,
- **comment** - uživatelský komentář k úloze,
- **time_start** - plánovaný čas začátku výpočtu,
- **time_end** - plánovaný čas ukončení výpočtu, podle zprávy o architektuře [13],
- **workunit_sum_time** - součet časů výpočetních jednotek použitý zpracování na výpočetních blokyů,
- **seconds_per_workunit** - čas určený adaptivní distribucí pro dělení výpočetních bloků,
- **charset1** - 1. množina symbolů definovaná uživatelem v hexadecimální podobě,
- **charset2** - 2. množina symbolů definovaná uživatelem v hexadecimální podobě,
- **charset3** - 3. množina symbolů definovaná uživatelem v hexadecimální podobě,
- **charset4** - 4. množina symbolů definovaná uživatelem v hexadecimální podobě,
- **rules** - název souboru s pravidly pokud jsou použita,
- **rule_left** - komolící pravidla prvního slovníku kombinačního útoku,
- **rule_right** - komolící pravidla druhého slovníku kombinačního útoku,
- **markov_hcstat** - název souboru obsahujícího markovské statistiky,
- **markov_threshold** - limit počtu stavů markovského modelu,
- **grammar_id** - databázový cizí klíč odkazující na pravděpodobnostní gramatiku, je-li zapotřebí,
- **case_permute** - příznak generátoru PRINCE módu útoku,
- **check_duplicates** - příznak přepínající kontrolu zdvojení kandidátních hesel pro útok PRINCE,
- **max_password_len** - filtr maximální délky hesla pro útok PRINCE,
- **min_password_len** - filtr minimální délky hesla pro útok PRINCE,
- **max_elem_in_chain** - limit maximální délky řetězce pro útok PRINCE,
- **min_elem_in_chain** - filtr minimální délky řetězce pro útok PRINCE,
- **generate_random_rules** - počet náhodně vytvořených pravidel pro útok PRINCE
- **deleted** - příznak označující měkké smazání úlohy,
- **kill** - příznak označující vynucené ukončení úlohy informující o nutnosti ukončení výpočtu,
- **batch_id** - databázový cizí klíč odkazující na dávku do které úloha patří, patří-li do nějaké,
- **queue_position** - pořadí ve frontě, pokud je úloha součástí dávky.

4.3 Komunikace

Komunikaci mezi serverem a klienty zajišťuje **scheduling protocol**[9] frameworku BOINC. Tento protokol funguje přes HTTP a obsahuje informace o verzi systému a chybách vzniklých během výpočtu. Důležitou součástí protokolu jsou odkazy na stažení souborů představujících vstupy a výstupy, s nimiž jsou posílány také informace o velikosti, jménu a kontrolní součet. Generování vstupních a výstupních souborů je již součástí úloh projektu Fitcrack. Počet a obsah souborů je závislý na použitém módu útoku. Počet a názvy souborů jsou definovány ve vstupních a výstupních vzorech uložených ve složce `templates`. Vzory úloh[10] používá BOINC pro specifikaci formátu vstupů a výstupů výpočetních bloků včetně potřebných argumentů příkazového řádku. Vstupní vzory definují vstupní soubory výpočet-

[9] https://boinc.berkeley.edu/trac/wiki/RpcProtocol
[10] https://boinc.berkeley.edu/trac/wiki/JobTemplates

ních bloků pro výpočetní jednotky a výstupní podobně soubory vrácené z úlohy zpět na server. Komunikace mezi serverem a klienty a naznačenými moduly je znázorněna na obrázku 4.2. Na zmíněném obrázku se nachází na levé straně server a na pravé straně skupina klientů. U obou stran jsou naznačeny podsystémy vstupujícími do komunikace, zejména podsystémy Scheduler a BOINC Client. Tyto systémy jsou součástí základní implementace systému BOINC a jejich účelem je plánování výpočtu na klientech a následný přenos souborů. Podsystém Trickler je již vlastní implementací projektu Fitcrack a přijímá zprávy o postupu výpočtu zasílaných z běžící aplikace na klientech.



Obrázek 4.2: Zobrazení komunikace mezi serverem a klienty. Trickle zprávy se zasílají postupně během výpočtu, jinak je komunikace znázorněna v pořadí průběhu zpracování jedné výpočetní části.

Pro každý mód útoku existuje vstupní vzor, který dále popisuje vstupní soubory. Za účelem testování výpočetních jednotek také existuje vstupní vzor pro tento případ. Každý vstup vždy obsahuje konfigurační soubor, útoky vždy očekávají vstupní soubor s daty obsahující jeden nebo více hashů. Jedná se o tyto vzory:

- **bench_in** - popisuje vstupní soubor pro testování jednotek, jedná se pouze o konfigurační soubor,
- **combinator_in** - vzor vstupů kombinačních útoků se dvěma slovníky,
- **dict_in** - vzor jednoduchého slovníkového útoku s jedním slovníkem,
- **dict_alt_in** - vzor slovníkového útoku bez mazání slovníků,
- **hybrid_masked_dict_in** - vzor vstupů slovníkového útoku s maskou, identický se vzorem slovníkového útoku, maska je součástí konfiguračního souboru,
- **hybrid_dict_mask_in** - vzor slovníkového útoku s maskou bez mazání slovníků,
- **markov_in** - vzor pro markovský útok,

- **mask_in** - vzor pro útok silou s maskou,
- **pcfg_in** - vzor vstupů módu útoku pravděpodobnostními gramatikami obsahující preterminály a gramatiku,
- **pcfg_rules_in** - vzor útoku pravděpodobnostními gramatikami rozšířený o soubor s komolícími pravidly,
- **prince_in** - vzor vstupů útoku PRINCE, identický se vzorem slovníkového útoku,
- **prince_rules_in** - vstupy PRINCE útoku rozšířené o soubor s pravidly,
- **rule_in** - vzor útoku se slovníkem s komolícími pravidly,
- **rule_alt_in** - vzor útoku se slovníkem s komolícími pravidly bez mazání slovníků.

Soubory se stejnými jmény sdílí formát obsahu i pro rozdílné módy útoku, vzory takto definují tyto soubory:

- **config** - konfigurační soubor, popis obsahu je uveden dále,
- **data** - soubor obsahující množinu lámaných hashů,
- **dict1** - první slovník,
- **dict2** - druhý slovník,
- **rules** - soubor s komolícími pravidly popsanými Hashcatem,
- **markov** - soubor s markovskými statistikami,
- **preterminals** - soubor s preterminály pravděpodobnostních gramatik,
- **grammar** - soubor obsahující popis pravděpodobnostní gramatiky v serializované podobě.

Konfigurační soubor je textový soubor rozesílaný výpočetním jednotkám při každé úloze. Velká část proměnných konfigurační souboru přímo odpovídá přepínačům příkazové řádky aplikací spouštěných na výpočetní jednotce. Jeho obsah je kódovaný použitím TLV („Type Length Encoding“) ve formátu:

```
|||name|type|length|value|||
```

Pole „name“ identifikuje proměnnou upravovanou řádkem konfiguračního souboru. Pole „type“ indikuje typ dat obsažených v posledním poli „value“. Možné jsou následující typy:

- **Bool** - 0 nebo 1 představující pravdivostní hodnotu
- **Char** - jedno-bytový symbol
- **String** - sekvence symbolů
- **Int** - 32-bitové číslo se znaménkem
- **UInt** - 32-bitové číslo bez znaménka
- **BigInt** - 64-bitové číslo se znaménkem
- **BigUInt** - 64-bitové číslo bez znaménka

Délka pole „value“ je značena polem „length“ a její hlavní využití spočívá v odstranění nutnosti escapování znaků. Délka udává počet bytů použitých v textovém souboru, u čísel toto znamená počet číslic, jelikož je hodnota ukládána v čitelné podobě. Pole „value“ udává hodnotu spravované proměnné. Některé útoky nevyžadují některé proměnné, tyto proměnné nejsou v takovémto případě uvedeny v souboru. Konfigurační soubor může obsahovat tyto parametry:

- **attack_mode** - parametr `-a` programu Hashcat specifikující mód útoku,
- **attack_submode** - podmód útoku často určující použití komolících pravidel,

- **hash_type** - parametr `-m` určující typ hashe v Hashcatu,
- **name** - název lámací úlohy,
- **charset1** - uživatelem definovaná množina znaků pro masku,
- **charset2** - uživatelem definovaná množina znaků pro masku,
- **charset3** - uživatelem definovaná množina znaků pro masku,
- **charset4** - uživatelem definovaná množina znaků pro masku,
- **rule_left** - komolící pravidla levého slovníku kombinačního útoku,
- **rule_right** - komolící pravidla pravého slovníku kombinačního útoku,
- **mask** - maska symbolů pro útok silou,
- **start_index** - parametr `--skip` Hashcatu značící počátek výpočetního bloku,
- **hc_keyspace** - parametr `--limit` Hashcatu značící počet neupravovaných kandidátních hesel od počátku bloku,
- **mask_hc_keyspace** - velikost množiny kandidátních klíčů pro útok silou s maskou,
- **mode** - typ bloku, určující test, celkový test nebo normální výpočet,
- **markov_threshold** - hranice zastavení markovských řetězců,
- **dict_hc_keyspace** - počet kandidátních hesel ve slovníku pro PRINCE útok,
- **case_permute** - permutace pro PRINCE útok,
- **check_duplicates** - kontrola vzniklých duplikátů pro PRINCE útok,
- **max_password_len** - maximální délka vzniklých kandidátů pro PRINCE útok,
- **min_password_len** - minimální délka vzniklých kandidátů pro PRINCE útok,
- **max_elem_in_chain** - maximální délka zřetězení při tvorbě kandidátů pro PRINCE útok,
- **min_elem_in_chain** - minimální délka zřetězení při tvorbě kandidátů pro PRINCE útok,
- **skip_from_start** - počet přeskakovaných hesel ze začátku pro PRINCE preprocesor,
- **generate_random_password** - parametr Hascatu udávající počet náhodných komolících pravidel,
- **hw_temp_abort** - hraniční teplota jednotky, při které je výpočet zrušen,
- **benchmark_dict1** - rozložení délek kandidátů ve slovníku 1 pro účel otestování výkonu,
- **benchmark_dict2** - rozložení délek kandidátů ve slovníku 2 pro účel otestování výkonu.

Výstupní vzor existuje pouze jeden a je sdílený mezi všemi útoky i testovacími úlohami. Výstupní vzor se nazývá **app_out**.

4.4 Dělení úloh za účely paralelizace

Úloha je specifikována módem útoku, vstupy a nastavením blíže specifikovanými v podsekci o databázi 4.2.5. Za účelem distribuce práce je zapotřebí vstupy úlohy vhodně rozdělit na bloky, které mohou dále zpracovávat výpočetní jednotky. Jak je zmíněno v technické zprávě Fitcracku [13, 11], bloky lze vytvářet třemi způsoby:

- **Dělením hashů** lze dosáhnout přirozeného rozdělení úlohy na více jednotek. Každá výpočetní jednotka má k dispozici část hashů a všechna kandidátní hesla. Problém s tímto přístupem je, že hashcat je schopen testovat kandidátní hashe vůči více lámaným hashům, ovšem takto mu to není umožněno a může tedy docházet zbytečně

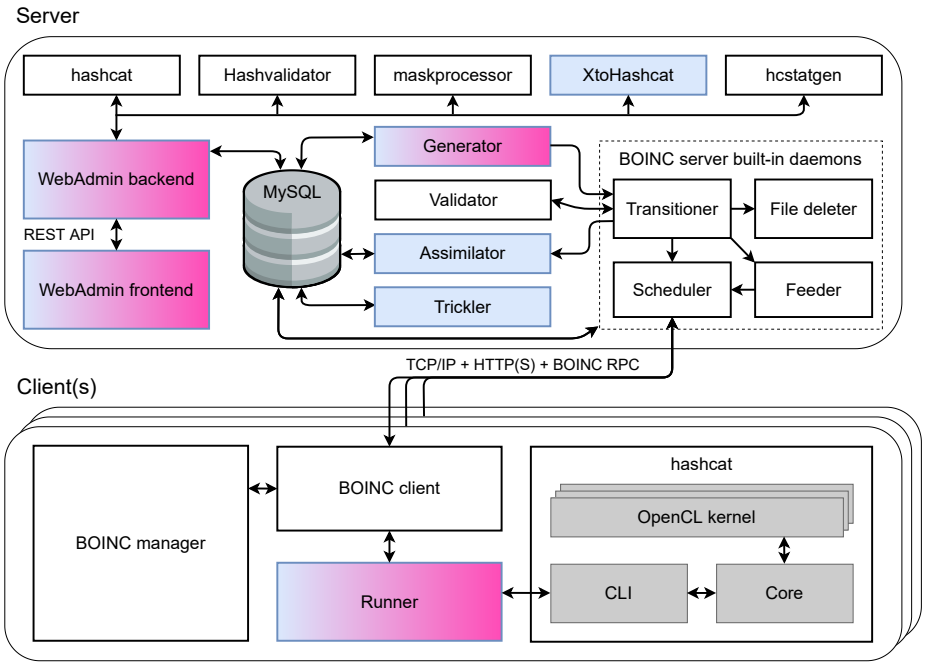
k mnohonásobnému výpočtu kandidátního hashe. Proto je tento přístup považován za neefektivní.

- **Statické dělení kandidátních hesel** je výhodné pro svou jednoduchost, díky které dosahuje nízké režie. Kandidátní hesla jsou na začátku lámání rozdělena mezi výpočetní jednotky. Tento způsob dělení se ovšem špatně vypořádává se změnami v sítí. Pokud dojde k výpadku, je celá část ztracena a musí dojít k opakování jejího výpočtu.
- **Dynamické dělení kandidátních hesel** je způsob distribuce aplikovaný v projektu Fitcrack. Tento přístup generuje výpočetní bloky postupně během procesu lámání a umožňuje tak reagovat na změny ve výpočetní rychlosti i změny v síti. Dynamické dělení začíná s bloky relativně malých velikostí, jejichž velikost následně upravuje podle rychlosti výpočetních jednotek. Ke konci prostoru kandidátních hesel začne metoda velikost výpočetních bloků opět zmenšovat, aby došlo k přesnému dokročení a byly efektivně využity všechny výpočetní jednotky a skončily tedy ideálně současně.

Kapitola 5

Návrh

Pro implementaci asociačního útoku je zapotřebí upravit část systémů. Jedná se o systémy Generátor a Runner. Dále je zapotřebí vytvořit novou Vue komponentu představující útok a jeho nastavení ve frontendu aplikace. Pro spouštění výpočtu jsou přidány nové vzorové soubory obsahující vstupní soubory a jejich popis. Přidání nového útoku může vyžadovat úpravu databáze MySQL, která ve své tabulce **fc_job** uchovává parametry útoků. Změněné systémy a jejich umístění v architektuře jsou naznačeny v obrázku 5.1.



Obrázek 5.1: Zvýraznění částí ovlivněných přidáním nového útoku, modře vlastní části projektu Fitcrack, s růžovým gradientem ovlivněné části

Asosiačnmí útok je ve své podobě velmi podobný útoku se slovníkem a často obsahuje velké množství komolících pravidel. Vzhledem k této charakteristice bude vstupní vzor velmi podobný již existujícím vzorům **rule_in** a **dict_in**, které popisuje právě slovníkový útok s pravidly a bez nich. Také frontend a jeho formuláře asociačního útoku v systému WebAdmin budou téměř identické se slovníkovým útokem nacházejícím se v komponentě **dictionary.vue** zdrojového kódu aplikace WebAdmin. Stejně tak zobrazení detailu úkolu ve

stejně pojmenované komponentě **dictionary.vue** lze z větší část převzít. Databázi MySQL není nutné upravovat, protože lze použít parametry slovníkového útoku, pouze je zapotřebí ve specifikaci vyhradit nový identifikátor sloupce **attack_mode** značící použití asociačního útoku a jeho dva podmódy ve sloupci **attack_submode** specifikující použití komolících pravidel.

Konfigurační soubor posílaný generátorem pro asociační útok obsahuje tyto parametry:

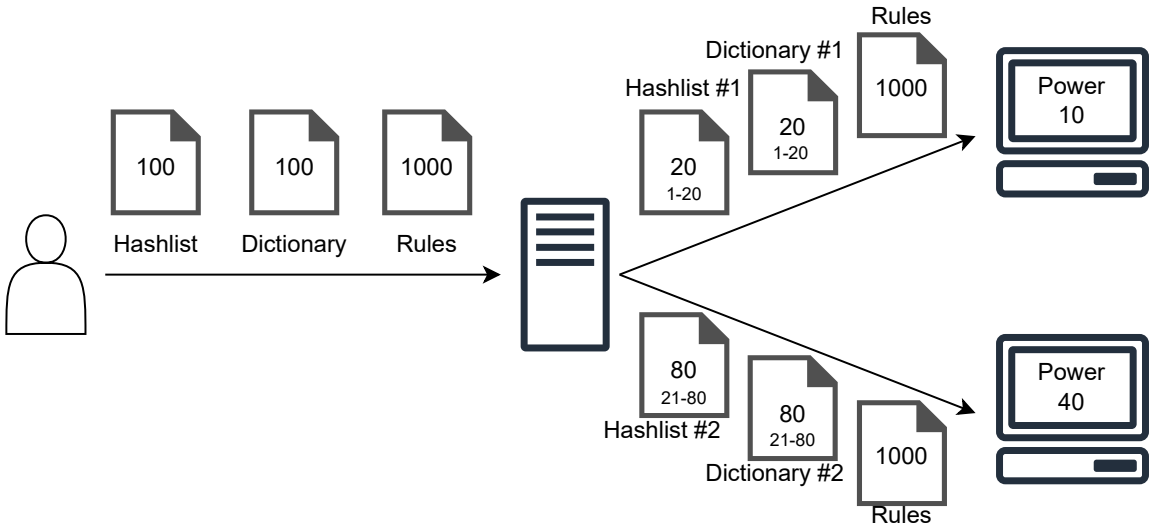
- **attack_mode** - parametr `-a` programu Hashcat, hodnota **9**, která je v programu Hashcat přidělena asociačnímu útoku,
- **attack_submode** - podmód útoku určující použití komolících pravidel,
- **hash_type** - parametr `-m` určující typ hashe v Hashcatu,
- **name** - název lámací úlohy,
- **hw_temp_abort** - hraniční teplota jednotky, při které je výpočet zrušen,
- **start_index** - parametr `--skip` Hashcatu značící počátek výpočetního bloku ve slovníku,
- **hc_keyspace** - parametr `--limit` Hashcatu značící počet neupravovaných kandidátních hesel od počátku bloku,
- **mode** - typ bloku, hodnota **n**, značící normální výpočet.

5.1 Distribuce zátěže

Jak bylo zmíněno výše v sekci 4.4, Fitcrack používá k distribuci zátěže adaptivních bloků. Tyto bloky vznikají dělením prostoru kandidátních hesel. Každé výpočetní jednotce je přidělena malá část kandidátních hesel podle jejího dříve otestovaného výkonu na daném hashi.

V případě použití slovníkového útoku, kterému je asociační útok podobný dochází k dělení slovníku na skupiny kandidátních hesel. Tento přístup je možný, protože slovníky jsou běžně dostatečně velké a množina pravidel dostatečně malá na to, aby bylo možné vyrovnávat zátěž téměř ideálně. Jelikož asociační útok používá kandidátní hesla jako nápovědy k lámaným hashům, jsou tato hesla silně svázaná se svým hashem. Distribuce slovníku by takto znamenalo zároveň distribuovat lámané hashe. Tato distribuce je znázorněna na obrázku 5.2. Takto vzniká možná nevýhoda vícenásobného lámání již prolomeného hashe, pokud se vyskytuje ve vstupu vícekrát s různou nápovědou, jak je uvedeno výše v sekci 4.4 u distribuce lámaných hashů. Tato možnost je dále prozkoumána v následující sekci 5.2.

Charakter asociačního útoku umožňuje použití dříve neprozkoumané distribuce. Jedná se o distribuci komolících pravidel, která je možná protože základní ideou asociačního útoku je použití velké množinou pravidel. Idea distribuce je znázorněna v diagramu 5.3. Celkový počet kandidátních hesel vzniklý použitím komolících pravidel je $M \times N$, kde M značí počet slov obsažených ve slovníku a N počet použitých pravidel. Z hlediska prostorové náročnosti distribuce lze toto dělení značit jako $(W \times M + U \times N/U)$, kde U označuje počet vzniklých bloků a W počet pracovních jednotek. Výskyt U v násobení symbolizuje počet nutných přenosů bloků v sítí a jeho výskyt v děliteli představuje fragmentaci souboru s pravidly. Tato distribuce je velmi podobná distribuci slovníku, kterou lze zaznačit jako $(U \times M/U + W \times N)$. Ze zápisu lze jednoduše vyčíst, že použití distribuce pravidel je podobně efektivní jako distribuce slovníků, a je tedy efektivně použitelná pro vstupy, kde počet pravidel přesahuje počet hesel. Tato situace je běžně očekávatelná u asociačního útoku a takto je umožněno distribuovat i lámání jediného hesla. Fitcrack dále implementuje distribuci celých souborů za účelem odlehčení výpočetní zátěže serveru za cenu vytížení sítě. Toto rozdělení lze popsat náročností $(W \times M + W \times N)$. Celé soubory není zapotřebí přenášet



Obrázek 5.2: Zjednodušený diagram dělení podle dvojic hashů-nápověd nezohledňující adaptivní bloky

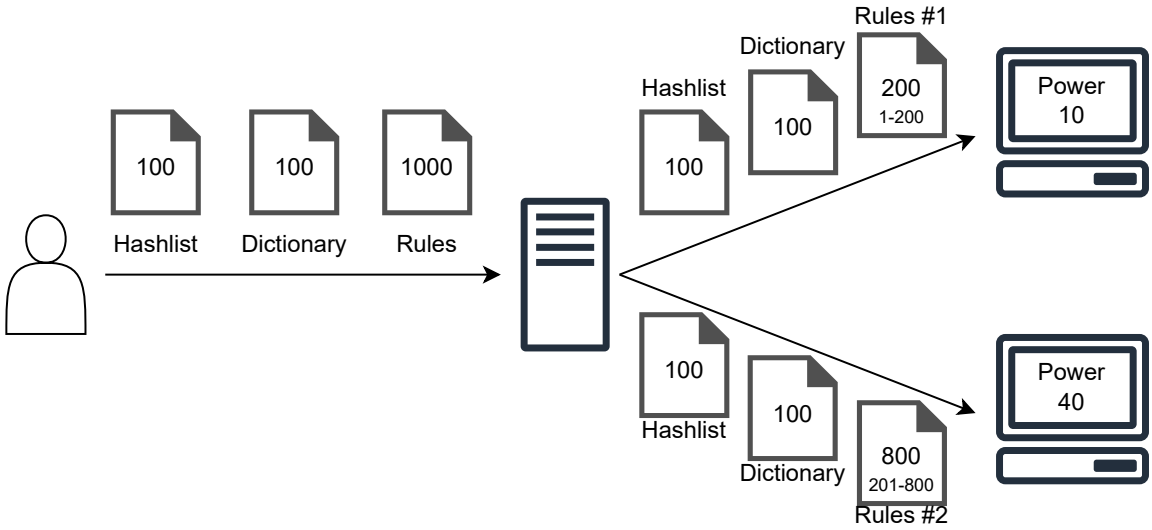
vícekrát, protože BOINC umožňuje nastavení příznaku *sticky*, který zamezuje mazání takto označených souborů a umožňuje *runneru* nestahovat soubory se stejným jménem jako již existující soubor. Této vlastnosti se používá u všech výše uvedených distribucí pro jejich specifické neměnící se soubory.

Naivní možností dělení zátěže je také předzpracování slovníku a pravidel na serveru a distribuce vzniklých kandidátních hesel. Tento přístup k distribuci ovšem zvyšuje režii serveru a zároveň také zbytečně zatěžuje síť. Prostorový zápis této distribuce je $U \times (M \times N)/U$. Při porovnání s výše zmíněnou distribucí je vidět, že naivní distribuce je kvadraticky náročná na přenos pro každou jednotku, zatímco dělení souborů je pouze lineárně náročné.

Jelikož Hashcat umožňuje použití asociačního útoku bez pravidel, není vždy možné distribuovat úlohu dělením pravidel. Proto je možné pro jednotlivé podmódy útoku implementovat odlišnou metodu distribuce nebo implementovat kombinované dělení jak hashů tak pravidel. Jednoduše lze také zavést dělení na základě porovnání délek slovníku a souboru s pravidly, které povede na použití jednoho nebo druhého systému distribuce.

5.2 Testování funkčnosti Hashcatu

Za účelem determinace možných problémů při distribuci byly nad asociačním módem v programu Hashcat provedeny jednoduché testy. Cílem testů bylo zjištění možných předávaných informací mezi jednotlivými iteracemi asociačního útoku, tedy mezi každým lámaným hashem. Za účelem odstranění možnosti uhádnutí hesla silou nebyla při testu použita žádná komolící pravidla. Vstup testů je znázorněn v tabulce 5.1. Jedná se o záměnu pořadí a zdvojení stejného lámaného hashe. Výsledkem testů bylo pouze prolomení třetího hashe, k němuž je poskytnuto správné heslo. Mezi důležité výstupy programu pro test patří počet prolomených hashů `1/4`, který napovídá, že hashcat v asociačním módu útoku nekontroluje a neošetřuje použití zdvojených hashů. Zvláštní abnormalitou ve výstupu je počet unikátních solí `4` i přesto, že zdvojený hash obsahuje stejnou sůl.



Obrázek 5.3: Zjednodušený diagram dělení podle komolících pravidel nezohledňující adaptivní bloky

kandidátní heslo	**správné heslo**	**hash a sůl**
nastrojhashcat	johntheripper	F600B7786FC68F2B9FC179D18AD0D958CA333AA7:1
johntheripper	nastrojhashcat	6AC5ABE7D5CF72AF5C3593DAD5CD64EA976150C5:2
areyoukiddingme	areyoukiddingme	9D003D229647EBDAA291768A4933245DCD4F9643:4
neniuvedenoheslo	areyoukiddingme	9D003D229647EBDAA291768A4933245DCD4F9643:4

Tabulka 5.1: Tabulka znázorňující vstupy testů na chytré předávání informací mezi iteracemi asociačního útoku.

Během testování byla objevena pravděpodobná chyba v implementaci asociačního útoku v Hashcatu. Při použití neosolených hashů indikuje program správně použití použití jedné unikátní soli. Asociační mód ovšem očekává počet unikátních solí rovný počtu záznamů ve slovníku. Nelze tedy provádět asociační útok nad více než jedním neosoleným hashem. Tato chyba pravděpodobně vznikla záměnou unikátních solí a počtu „raw hashů“. Chyba je zvláštní, protože ve zmíněném soleném testu se používá zdvojeného hashe i se solí, ovšem asociační útok je proveden bez chyby.

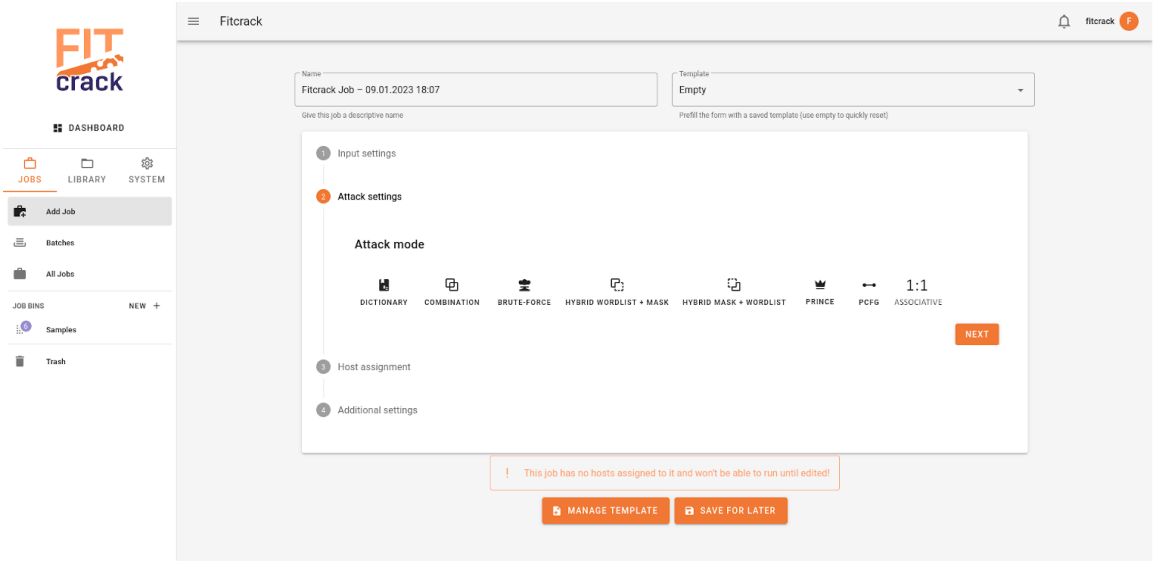
Z výsledků testů vyplývá, že lze beze ztráty efektivity pro distribuci použít dělení hashů provázaných se slovníky. Běžně je vhodné použití asociačního útoku s velkým množstvím lámaných hesel, protože pro malé množství je efektivita velmi podobná běžnému slovníkovému útoku. Je tedy možné, že distribuce hashů je dostatečná pro běžné použití, ovšem jak bylo zmíněno výše v této sekci, bylo by vhodné implementovat i distribuci komolících pravidel, pro případy, kdy počet komolících pravidel značně překračuje počet lámaných hashů.

5.3 WebAdmin frontend

Tvorba nového útoku v uživatelském rozhraní zahrnuje vytvoření ikony pro jednoduchou identifikaci typu útoku uživatelem. Ikony jsou propojeny se svými útoky jednoduchým

switch výrazem. Jako návrh ikony pro asociační útok byl zvolen text `1:1`, představující provázání hesel a nápověd k nim. Tato ikona je zobrazována na více místech v uživatelském prostředí, zejména v záložkách **Dashboard**, **Add Job** a **All Jobs**, ovšem pohledy **Dashboard** a **All Jobs** nevyžadují přímý zásah, jelikož používají pouze zmíněnou ikonu. Tato navržená ikona byla během implementace změněna na `mdi-account-details`, jelikož ikona podobná návrhu nebyla nalezena mezi Material Design Icons, které používají ostatní útoky.

Pohled **Add Job** se nachází v navigaci pod záložkou **JOBS** a tlačítkem **Add Job**. Tento pohled je zobrazen na obrázku návrhu 5.4. Aby byl nový útok zobrazen jak je naznačeno na obrázku tohoto pohledu je zapotřebí přidat útok do pole **attacks** nacházejícím se v souboru `store/job-form.js`. Tento pohled využívá komponenty představující jednotlivé útoky. Tyto komponenty obsahují nastavení specifické pro daný útok a je zapotřebí vytvořit novou komponentu představující asociační útok. Nastavení asociačního útoku odpovídá nastavení útoku se slovníkem a lze tedy většinu kódu převzít. Komponenta tedy bude obsahovat výběr slovníků, výběr pravidel a volbu módu distribuce. Samotný pohled **Add Job** vyžaduje jako jedinou úpravu import nové komponenty.



Obrázek 5.4: Uživatelské rozhraní pro tvorbu nového lámacího úkolu s naznačeným rozšířením pro asociační útok

Po vzniku výpočetního úkolu je uživatel přesměrován na pohled popisující detail úkolu se specifickými vlastnostmi daného útoku. Jedná se o pohled **Job Detail** a podobně jako u **Add Job** je zde zapotřebí vytvořit komponentu znázorňující nový asociační útok. Tato komponenta musí při zobrazování informací zohledňovat volitelnou přítomnost souboru pravidel a vždy přítomnost slovníku. Na stejném pohledu se zobrazují také informace o výpočetních blocích, které obsahují informace o svém rozsahu, který je běžně násoben s pravidly, což bude při distribuci pravidel zapotřebí změnit a naopak násobit rozsah počtem hashů. Tyto hodnoty jsou ovšem předávány z backendu, implementace tedy žádnou skutečnou změnu na frontend neprovede.

5.4 WebAdmin backend

V souboru `api/fitcrack/attacks/processJob.py` je zapotřebí připravit funkce zpracovávající vstupy nového útoku **process_job_10** a **post_process_job_10**. Tyto funkce lze opět vytvořit převzetím velké části implementace slovníkového útoku. Tyto funkce jsou volány z funkce **create_job** v souboru `api/fitcrack/endpoints/job/functions.py`, která vytváří záznam o úkolu v databázi. Pokud by bylo zapotřebí modifikovat databázi, je zapotřebí přizpůsobit se změnám i zde.

Funkce **process_job_10** je zodpovědná za tvorbu záznamu v tabulce **fc_job**, přičemž je důležité připomenout, že slovníky a pravidla již nahrány na server jsou pro účely vytvoření úlohy a hashe jsou nahrávány mimo zpracovávání specifických útoků. Druhá funkce **post_process_job_10** slouží pro tvorbu sekundárních tabulek, které potřebují primární klíč z tabulky **fc_job**. V případě asociačního útoku jsou v této funkci ukládány vztahy mezi slovníky a úkoly, které jsou v relaci N ku N.

5.5 Generátor

Podsystém generátor je zapotřebí upravit, aby byl schopen rozpoznat asociační útok ve funkci **CreateAttack**. Tato funkce dále vytváří třídu daného útoku zodpovědnou za tvorbu výpočetních bloků. Třída asociačního útoku bude definována v sousední složce **AttackModes**, jež obsahuje definice tříd implementovaných útoků. Součástí této třídy bude implementace nového módu distribuce navrženého výše v sekci 5.1.

Třída obsahuje výpočet velikosti bloku na základě výkonu jednotky a následnou tvorbu souborů specifikovaných konfiguračním souborem. Podle specifikovaného módu distribuce je zapotřebí některé vstupní soubory rozdělit a nahrát do výstupního souboru pouze určitou část. Kvůli rozdělení podle pravidel je zapotřebí, aby v takovém případě prostor klíčů odpovídal velikosti množiny pravidel a nikoliv množině slov slovníků, jak je tomu u ostatních rozdělení. Tato změna se ovšem týká nahrávání dat do databáze v backendu.

5.6 Vzorové soubory se vstupními soubory

Ve složce **templates**, která je součástí serveru, je zapotřebí vytvořit nové vzorové soubory popisující vstupní soubory výpočetních bloků. Každý vzorový soubor odpovídá jednomu způsobu distribuce souborů na jednotky, jelikož každá distribuce používá různé **sticky** soubory a soubor s pravidly není povinný. Nový vzor bude podobný vzoru útoku se slovníkem s pravidly, popřípadě bez pravidel. Bude tedy obsahovat definici existence následujících souborů:

- **config** - konfigurační soubor,
- **data** - množina hashů,
- **dict1** - množina nápověd se stejnou mohutností jako množina hashů,
- **rules** - množina pravidel, pro útok bez pravidel bude vytvořen vzor bez tohoto souboru.

5.7 Runner

Na straně klientů je zapotřebí vytvořit soubor vytvářející middleware mezi hashcatem a systémem BOINC. Účelem middleware je správné nastavení argumentů aplikace Hashcat. Pro

asociační mód útoku se jedná zejména o použití jeho přepínače `-a9`. Rozpoznání útoku z konfiguračního souboru probíhá v souboru **Attack.cpp**, který vytváří třídy představující jednotlivé útoky. Tyto třídy vytváří argumenty programu specifické pro daný útok. Nový útok tedy bude zapotřebí implementovat v těchto souborech a třídách. Třída nového útoku `AttackAssociation` bude specifikovat výše zmíněný argument módu útoku. Dále bude přidávat argumenty specifikující slovník a pravidla. Jednou zvláštností třídy bude, že se musí vyrovnat se spuštěním testovací výpočetní jednotky, která se vyznačuje velikostí 0, což běžně znamená, že je testována bez slovníku. Jelikož asociační útok má hashe silně provázané se slovníkem nelze takovouto jednotku spustit. Jednoduchým řešením tohoto problému je spouštět testovací jednotky v módu slovníkového útoku namísto útoku asociačního.

Kapitola 6

Implementace

Tato kapitola popisuje praktické a technické aspekty implementace rozšíření, jako jsou konkrétní algoritmy a útržky kódu, programovací jazyky a frameworky jednotlivých částí a problémy řešené v jednotlivých částech během vývoje. Strukturu souborů projektu, které byly v této práci upravovány lze najít v příloze B.

Kapitola popisuje zásah implementace do jednotlivých částí projektu fitcrack. Jednotlivé sekce se věnují těmto částem v pořadí v jakém se propaguje informace o úkolu, to je od webové aplikace, počínaje jejím frontendem, přes generátor po runner.

6.1 Webadmin

Jak bylo popsáno výše v podsekci 4.2.1 webová aplikace Webadmin se dělí na další dvě podčásti. „Single page“ aplikaci napsanou ve frameworku Vue.js, který používá tvorbu vlastních komponent a jejich vkládání pro tvorbu složitějšách komponent a zobrazení. A API server napsaný v jazyce Python s použitím knihoven Flask a SQLAlchemy. Nové soubory a soubory zasažené úpravami jsou znázorněny ve stromě v příloze B.3.

6.1.1 Frontend

Frontend se nachází ve složce `fitcrackFE`. Obsahuje klasickou strukturu Vue projektu s doplňkem Vue-store. Vue-store se stará o komunikaci s API serverem pomocí mutací a zaznamenává interní data aplikace v tzv. stavu. Soubor `job-form.js` obsahuje mutace a stav relevantní k vytváření nových útoků. Jelikož v rámci rozšíření nebylo zasaženo do databáze a bylo pouze využito existujících API endpointů, nebylo zapotřebí vytvářet ani měnit mutace obsažené v tomto souboru. Tento soubor obsahuje konstantní proměnnou `attacks` používanou na frontendu pro dynamické generování komponent útoků a jejich identifikace podle jejich id používaného v databázi. Hodnoty této proměnné pro asociační útok lze vidět v následujícím útržku kódu spolu s hodnotami pro slovníkový útok pro porovnání:

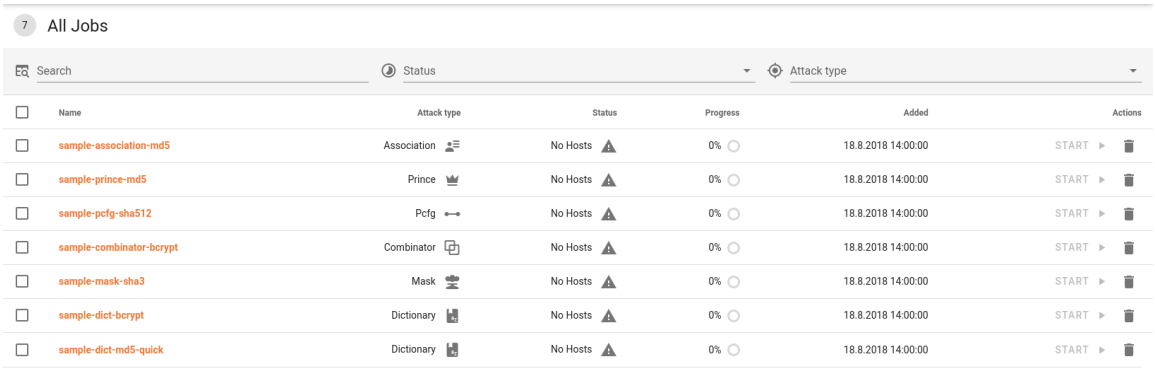
```
{handler: 'dictionary', name: 'Dictionary', id: 0, serverName: 'dict'},
{handler: 'association', name: 'Association', id: 10, serverName: 'assoc'},
```

Hodnota parametru `handler` určuje název komponent frontendu, který bude dynamicky renderován. Parametr `name` je jméno útoku zobrazované uživateli v uživatelském rozhraní. Parametry `id` a `serverName` jsou důležité pro komunikaci se serverem a databází podle nichž je útok rozpoznán v odpovědi od API serveru, popřípadě uložen do databáze.

```
export function attackIcon (handler) {
  switch (handler) {
    case 'dictionary':
      return 'mdi-dictionary'
    case 'association':
      return 'mdi-account-details'
    default:
      return 'mdi-checkbox-blank-outline'
  }
}
```

Výpis 6.1: Funkce přiřazující ikonu útoku.

Definování ikony útoku se nachází v souboru `assets/scripts/iconMaps.js` a je určována funkcí `attackIcon`, jejíž část je znázorněna ve výpisu kódu 6.1. Funkce určuje útok na základě výše zmíněné hodnoty `handler` z proměnné `attacks`. Byla zvolena ikona `mdi-account-details`, tato ikona reprezentuje osobní informace na kterých se běžně zakládají nápovědy asociačního útoku. Kombinace použití těchto hodnot a ikony pro dynamické vykreslování se nachází například v pohledu „All Jobs“, který je znázorněný na obrázku 6.1 a nevyžaduje žádné vlastní úpravy kódu.



Obrázek 6.1: Pohled All Job znázorňující dynamický impakt proměnné `attacks` a Material Design ikony

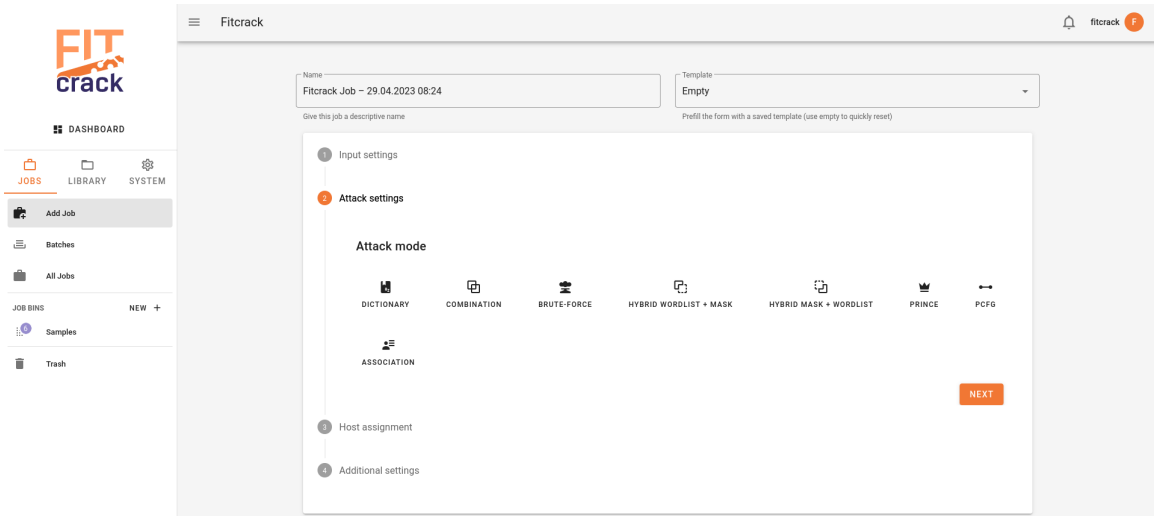
Podle této identifikace útoku je dále v tomto souboru zapotřebí doplnit specifické validace dat do funcke `validateAttackSpecificSettings()`. V případě asociačního útoku tato funkce validuje stejný počet kandidátních hesel a lámaných hashů, jak je možné vidět v útržku kódu 6.2. Jelikož fitcrack podporuje použití více slovníků naráz, je použita funkce `reduce` pro získání celkového počtu kandidátních hesel.

Komponenta formuláře zadání nového útoku jako úkolu pro systém se nachází ve složce `components/job/attacks` a je přidána do pohledu `components/job/addJobView.vue`, který generuje všechny komponenty útoků na základě výše zmíněné proměnné `attacks` a je zobrazena na obrázku 6.2. Komponenta formuláře obsahuje podkomponenty `dict-selector` a `rules-selector`, stojí za zmínku, že komponenta vstupních hashů je součástí nadřazeného pohledu a není tedy upravitelná pro jednotlivé útoky. Důležitou součástí komponenty asociačního útoku je výběr módu distribuce mezi štěpením slovníků na serveru, zasíláním celých souborů s parametry `-skip` a `-limit` a štěpením pravidel, jak je výše popsáno

```
validAttackSpecificSettings (state) {
    switch (state.attackSettingsTab) {
        case 'dictionary':
            return state.leftDicts.length > 0
        case 'association':
            return
                state.leftDicts.reduce(
                    (total, current)=>total+current.keyspace, 0
                ) == state.validatedHashes.length
    }
}
```

Výpis 6.2: Funkce ověřující validitu vstupních dat formulře útoků.

v sekci návrhu 5.1. Tato komponenta je také zodpovědná za verifikaci některých vlastností vstupů použitím události `oninput`. Kontroluje se zde přítomnost alespoň jednoho slovníku a současné vybrání dělení podle pravidel bez jakýchkoliv přidaných pravidel. Formulář s komponentami se nachází na obrázku 6.3.

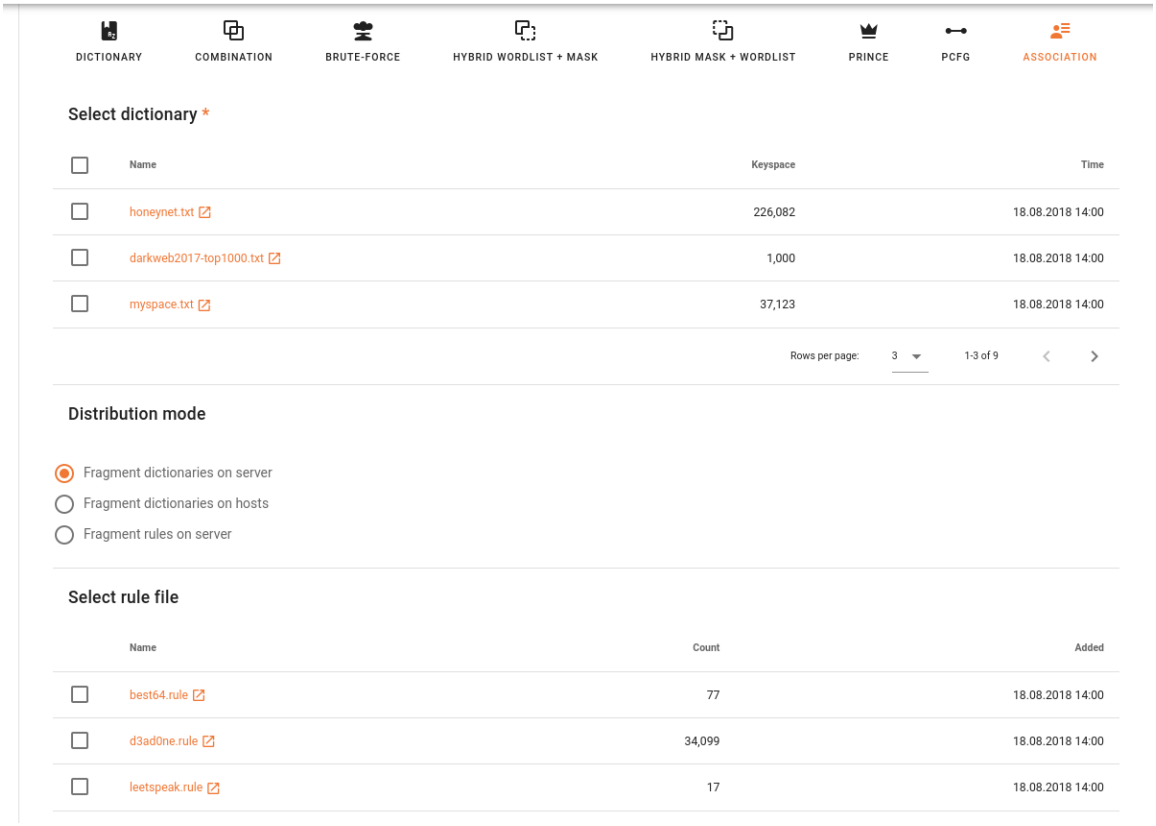


Obrázek 6.2: Pohled Add Job

Komponenta detailních informací zadaného úkolu asociačního útoku se nachází ve složce `components/jobDetail/attacks` a je přidána do pohledu `components/jobDetail/jobDetailView.vue`, který ji opět generuje na základě výše zmíněné proměnné `attacks` podle identifikátoru získaného z databáze. Zobrazuje soubory slovníků a pravidel přiřazených úkolu. Je zobrazena na obrázku 6.4

6.1.2 Backend

Backend se nachází ve složce `fitcrackAPI`, je rozdělen na dvě části `api` a `database`. Část `database` definuje modely tabulek a vztahy mezi modely, jelikož nebyla databáze upravována, neobsahuje tato část žádné změny. Část `api` se věnuje serveru komunikujícím s frontendem webadminu. Z této části je důležitý soubor `attacks/processJob.py`, který

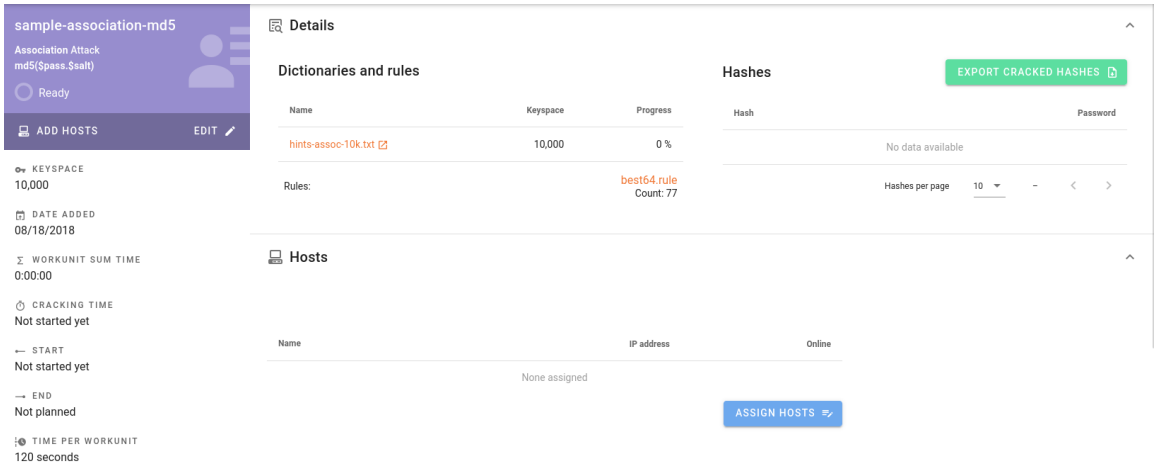


Obrázek 6.3: Pohled Add Job s formulářem asociačního útoku

obsahuje funkce zpracovávající parametry úkolu přijatého z frontendu podle *id* útoku odpovídajícího id specifikovaného ve výše zmíněné proměnné `attacks`. Tyto funkce jsou ve tvaru `process_job_{id}` a `post_process_job_{id}` a jejich implementaci pro nový asociační útok je možné vidět v následujících útržcích kódu 6.3 a 6.4. Účelem těchto funkcí je zpracované informace z frontendu uložit do databáze, se kterou komunikuje BOINC server.

Funkce `process_job_10` vytváří v databázi záznam samotného útoku a nastavuje důležité parametry jako název souboru s komolícími pravidly, je-li přítomen, podtyp útoku, značící právě přítomnost pravidel, počet kandidátních hesel a počet celkových hesel po aplikaci komolících pravidel. Pro distribuci dělením pravidel je zde nastavována proměnná `hc_keyspace` na hodnotu rovnou počtu komolících pravidel namísto běžné hodnoty počtu nápověd ve slovníku. Tato změna je provedena, protože podle této hodnoty se generátor rozhoduje zda je úloha ukončena a mohlo by dojít k předčasnému ukončení. Tato změna zároveň ovšem produkuje vedlejší problémy s výpočtem celkového rozsahu úlohy, která je běžně počítána vynásobením `hc_keyspace` s velikostí množiny pravidel, tento problém je řešen upravením hybridní vlastnosti `keyspace` modelu `FcJob` v souboru `database/models.py`. V této vlastnosti je přidán speciální případ asociačního útoku s dělením podle pravidel, při kterém funkce násobí hodnotu `hc_keyspace` s počtem hashů úkolu. Ve stejném souboru byla také zcela analogicky upravena hybridní vlastnost `start_index_real` modelu `FcWorkunit`, která je zobrazována mezi informacemi výpočetních bloků na frontendu a představuje počátek výpočetního bloku.

Funkce `post_process_job_10` je zodpovědná za vytváření záznamů v tabulkách závislých na záznamu v hlavní tabulce úkolu, jinými slovy vytváří záznamy v tabulkách, které



Obrázek 6.4: Pohled Job Detail s asociačním útokem

potřebují cizí klíč úkolu. V případě asociačního útoku se jedná o tabulku slovníků přidělených úkolu, kterých může být více, i když této funkcionality není doporučeno využít, protože nelze jednoduše ovlivnit v jakém pořadí budou slovníky čteny.

Další úpravy backendu zahrnují soubor `job/functions.py`, v němž ve funkci `computeCrackingTime` byl přidán výpočet počtu kandidátních hesel po použití pravidel pro asociační útok. Tato funkce je navázaná na API adresu `/crackingTime`, která je volána při tvorbě úkolu na frontendu, aby dala uživateli představu o době trvání úkolu, který právě tvoří. Ve stejném souboru byla také upravena funkce `kill_job`, tak aby vymazání již vyčerpaných slovníků proběhlo i u asociačního útoku. Stejná změna proběhla v akci `get` třídy `OperationWithJob` ze souboru `job/job.py` pro operaci restartování výpočetního úkolu, která tuto funkci volá při specifikované operaci `kill` nazývané také „purge“.

6.2 BOINC Server

Rozšíření serveru BOINC je implementováno v jazyce C++ a jeho integrace do balíčku BOINC je umožněna souborem `fitcrack_changes_in_boinc.patch` udávajícím změny v Makefilu nacházejícím se v hlavním adresáři projektu v podsložce `installer`. Za účelem odlišení asociačního útoku s pravidly a bez pravidel jsou vytvořeny dvě třídy módů útoku, které jsou podle informací z databáze použity třídou generátoru.

6.2.1 Generator

Diagram 6.5 znázorňuje strukturu generátoru a vyznačuje upravené třídy modře a nové třídy asociačního útoku fialově. Základová třída generátoru `AbstractGenerator` je rozšířena o ukládání názvů „sticky“ souborů, za účelem jejich pozdějšího odstranění z výpočetních jednotek po skončení výpočetního úkolu. Jak je zmíněno níže v sekci o vzorových souborech 6.2.2, jedná se zejména o soubor s pravidly pokud se práce dělí podle slovníků a o soubory s hashi a kandidátními hesly pokud se práce dělí podle pravidel, v případě využití parametrů `--skip` a `--limit` jsou *sticky* všechny soubory.

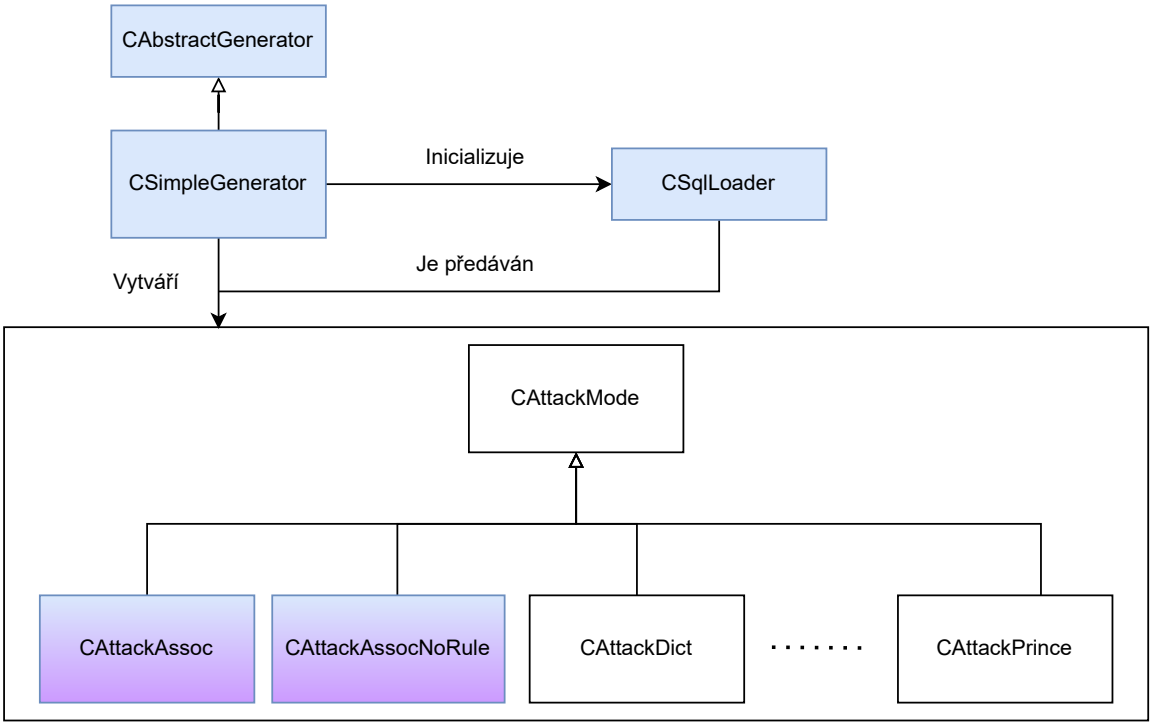
Třída `SimpleGenerator` je zodpovědná za vytvoření správné třídy představující mód útoku a je tedy rozšířena tak, aby zahrnovala výše zmíněné módy asociačního útoku s pravidly, představovaný novou třídou `CAttackAssoc`, a bez pravidel, představovaný třídou

```
# association attack
def process_job_10(job):
    job['attack_settings']['attack_submode'] = 0
    job['attack_name'] = 'association'
    job['hc_keyspace'] = 0

    for dictObj in job['attack_settings']['left_dictionaries']:
        dict = FcDictionary.query.filter(
                FcDictionary.id == dictObj['id']
            ).first()
        if not dict:
            abort(500, 'Wrong dictionary selected.')
        if not os.path.exists(os.path.join(DICTIONARY_DIR, dict.path)):
            abort(500, 'Dictionary does not exist.')
        job['hc_keyspace'] += dict.keyspace

    ruleFileMultiplier = 1

    if job['attack_settings']['rules']:
        rules = FcRule.query.filter(
                FcRule.id == job['attack_settings']['rules']['id']
            ).first()
        ruleFileMultiplier = rules.count

        if ruleFileMultiplier == 0:
            ruleFileMultiplier = 1

        if not rules:
            abort(500, 'Wrong rules file selected.')
        if not os.path.exists(os.path.join(RULE_DIR, rules.path)):
            abort(500, 'Rules file does not exist.')

        job['attack_settings']['attack_submode'] = 1
        job['rules'] = rules.name

    job['keyspace'] = job['hc_keyspace'] * ruleFileMultiplier

    # in case of rule distribution hashcat keyspace is defined by rules
    if job['attack_settings']['distribution_mode'] == 2:
        job['hc_keyspace'] = ruleFileMultiplier

    return job
```

Výpis 6.3: Funkce zpracovávající informace z frontendu do databáze nacházející se na API serveru.

```
def post_process_job_10(data, db_job):
    for dict in data['attack_settings']['left_dictionaries']:
        jobDict = FcJobDictionary(
            job_id=db_job.id,
            dictionary_id=dict['id']
        )
        db.session.add(jobDict)
```

Výpis 6.4: Funkce zpracovávající informace z frontendu do vedlejších tabulek databáze nacházející se na API serveru.



Obrázek 6.5: Diagram znázorňující třídy generátoru a vztahy mezi nimi s vyznačenými úpravami modře a novými třídami asociačního útoku fialově.

`CAttackAssocNoRule`. Teprve tyto třídy jsou zodpovědné za tvorbu a pojmenování souborů nahrávaných na výpočetní jednotku. To také znamená, že implementace distribuce zátěže a štěpení souboru se nachází právě v těchto třídách.

Nově vytvořené třídy `CAttackAssoc` a `CAttackAssocNoRule` jsou důvodem pro úpravu „patch“ souboru makefilu, aby byl linker schopný správně složit spustitelný server. Tato úprava je vyžadována protože hlavní část programu serveru BOINC je součástí samostatného git balíčku BOINC a fitcrack implementuje úpravy původní aplikace.

Třída `CAttackAssoc` implementuje zejména metodu `makeWorkunit`, tuto funkci lze logicky rozdělit do několika částí. První část je oddělena do podfunkce `generateWorkunit`, která vytváří instanci třídy `CWorkunit` představující výpočetní blok pro danou výpočetní jednotku. Součástí této podfunkce je výpočet „keyspace“, tj. počtu hesel, které budou v tomto bloku vyzkoušeny. Podfunkce `generateWorkunit` také spravuje současný slovník

a pozici posledního přečteného hesla v něm. Další částí metody `makeWorkunit` je tvorba konfiguračního souboru a generování jmen souborů. Jména souborů, která se mění kvůli fragmentaci původního souboru, obsahují časovou značku a sekvenční číslo inkrementující se s každým souborem, zatímco jména *sticky* souborů obsahují pouze identifikační jméno úkolu. Do konfiguračního souboru je nahrána velikost posílaného slovníku, vybraného ve výše zmíněné podfunkci. Pokud je úkol distribuován použitím parametrů `--skip` a `--limit`, obsahuje konfigurační soubor odpovídající hodnoty počátečního indexu a počtu hesel ke zkoušení v zasílaném sloučeném slovníku. V další části jsou zároveň řešeny hesla a lámané hashe, protože jsou tato data provázána u asociačního útoku. Pokud jsou tato data fragmentována na serveru je načten současný slovník a je z něj přepsán daný počet hesel, poté je také stejný počet hashů zapsán z databáze do souboru. Již vyzkoušené hashe jsou přeskočeny neoptimálně po řádcích podle počtu získaného z metody `getStartIndex` proměnné `m_workunit`, který je nastavován v podfunkci ve zmíněné první části při tvorbě třídy `CWorkunit`. Výjimkou provázání hashů a nápověd je výpočetní blok testování výkonu, který je identifikován v systému nepoužitím žádného hesla, toto by v případě asociačního útoku znamenalo nezaslání ani žádného hashe, což by vytvořilo chybu hashcatu. Proto je v případě bloku testování výkonu zasílán soubor se všemi hashi. Pokud nejsou hashe a nápovědy fragmentovány na serveru jsou slovníky sloučeny do jednoho souboru a výpočetním jednotkám jsou zasílány všechny hashe a nápovědy. Poslední částí před odesláním práce jednotce je zapsání souboru pravidel, pokud nejsou pravidla štěpena jsou opět jednoduše všechna zapsána do souboru a poslána jednotce ve formě *sticky* souboru. Pokud jsou pravidla štěpena jsou již použitá pravidla přeskočena opět po řádcích podle počtu z metody `getStartIndex` a je jich zapsán počet odpovídající počtu z metody `getHcKeyspace` nastavovaný také v první části na hodnotu, která symbolizuje počet hesel k projití podle síly jednotky. Jedná se o stejnou proměnnou používanou při fragmentování slovníků, ovšem pro případ štěpení pravidel obsahuje informace o počtu pravidel, jak je uvedeno výše v sekci 6.1.2. V posledním kroku vytváření výpočetního bloku jsou soubory registrovány spolu se vzorovým souborem odpovídajícím metodě distribuce na serveru BOINC.

Jelikož generátory běžně nefragmentují hashe, bývají načítány pouze neprolomené hashe. Prolomené hashe nemá běžně smysl posílat a je takto šetřeno prostorem. Jelikož asociační útok vyžaduje fragmentaci hashů spolu se slovníkem, při načítání pouze neprolomených hashů by byl ztracen index v hashích. Aby bylo možné použít stejný index pro hashe a slovníky, byla implementována nová funkce `loadJobAllHashes` ve třídě `CSqlLoader` načítající všechny hashe nezávisle na jejich stavu prolomení. Tato funkce je volána z modelu `CJob` metodou `loadHashes`, ve které je zkontrolováno, zda nejsou již všechny hashe načteny a čtení z databáze se provede pouze pokud zatím nejsou. Tato funkce také načítá běžně používané neprolomené hashe, které vytváří dotaz v databázi při každém volání této funkce.

6.2.2 Vzorové soubory se vstupními soubory

Vzorové soubory udávají přenášené soubory mezi BOINC serverem a výpočetními jednotkami. Tyto soubory specifikují pro výpočetní jednotky, které soubory jsou označené jako *sticky* a *no_delete*. Takovéto soubory jednotka nemaže po dokončení jednoho bloku úkolu, ale čeká na příkaz ze serveru pro jejich smazání. Tento příkaz je zasílán po dokončení celého úkolu všem jednotkám. Účelem nemazání souborů je, zamezení zbytečného vytěžování sítě, jelikož některé soubory se mezi bloky úkolu nemění. Pokud výpočetní jednotka dostane v konfiguračním souboru odkaz na soubor jehož jméno již existuje nestahuje soubor znovu.

Vzorové soubory jsou ve formátu XML, každý soubor je uveden pod elementem `file_info`, který obsahuje numerické označení souboru a případně elementy specifikující označení *sticky*. Kromě značek souborů vzor také obsahuje informace o referenčním jméně stahovaných souborů pro výpočetní vztažených na výše zmíněné numerické označení. Jelikož každý mód distribuce předpokládá neměnné jiné soubory a zároveň použití souboru pravidel je volitelné existuje pro asociační útok 5 vzorových souborů. Příklad vzorového souboru pro asociační útok s pravidly a distribuci podle slovníku bez použití argumentů `--skip` a `--limit` se nachází v útržku kódu 6.5.

Informace obsažené ve vzorovém souboru pro asociační útok s použitím komolících pravidel a distribucí zátěže použitím fragmentací slovníků kandidátních hesel spolu s lámanými hashi jsou obsaženy v tabulce 6.1. Jelikož tento způsob distribuce štěpí soubory slovníků a s nimi provázané hashe na serveru do podsouborů nelze pro výpočetní jednotky tyto soubory označit jako *sticky*. Alespoň soubor pravidel zůstává v tomto případě neměnný, ten je tedy označen *sticky* a stahuje se pouze jednou.

assoc_dict_split_in			
soubor	*id*	*sticky*	*no_delete*
config	0	-	-
data	1	-	-
dict1	2	-	-
rules	3	ano	ano

Tabulka 6.1: Informace o přenášených souborech ze vzorového souboru distribuce podle slovníku

Vzorový soubor pro asociační útok s pravidly ovšem bez štěpení na serveru, tedy s použitím argumentů `--skip` a `--limit`, je znázorněn tabulkou 6.2. Jelikož na serveru nedochází k žádnému štěpení je možné všechny důležité soubory lámání označit jako *sticky*. Tento přístup je tím výhodnější čím méně je používáno výpočetních jednotek, jelikož každá jednotka stahuje stejná duplicitní data, z nichž část bude pravděpodobně spočtena na jiné jednotce a nebude tedy na této zapotřebí.

assoc_dict_alt_in			
soubor	*id*	*sticky*	*no_delete*
config	0	-	-
data	1	ano	ano
dict1	2	ano	ano
rules	3	ano	ano

Tabulka 6.2: Informace o přenášených souborech ze vzorového souboru distribuce použitím argumentů `--skip` a `--limit`

Pro asociační útok distribuovaný štěpením komolících pravidel je vzorový soubor popsán tabulkou 6.3. Neměnící se soubory jsou přesně obráceně oproti štěpení slovníků, tedy slovník a soubor s hashi je označen *sticky*, zatímco soubor pravidel ne.

Jelikož přítomnost komolících pravidel je volitelná, jsou vzorové soubory zdvojeny a napodruhé z nich je odebrán soubor pravidel. Toto zdvojení se netýká distribuce štěpením podle pravidel z logických důvodů. Vzorové soubory těchto módů jsou pro úplnost uvedeny v tabulkách 6.4 a 6.5. Tyto módy odpovídají pro porovnání vzorům z tabulek 6.1 a 6.2.

```
<input_template>
    <file_info>
        <number>0</number>
    </file_info>
    <file_info>
        <number>1</number>
    </file_info>
    <file_info>
        <number>2</number>
    </file_info>
    <file_info>
        <sticky/>
        <no_delete/>
        <number>3</number>
    </file_info>
    <workunit>
        <file_ref>
            <file_number>0</file_number>
            <open_name>config</open_name>
        </file_ref>
        <file_ref>
            <file_number>1</file_number>
            <open_name>data</open_name>
        </file_ref>
        <file_ref>
            <file_number>2</file_number>
            <open_name>dict1</open_name>
        </file_ref>
        <file_ref>
            <file_number>3</file_number>
            <open_name>rules</open_name>
        </file_ref>
    </workunit>
</input_template>
```

Výpis 6.5: Konkrétní příklad používaného vzorového souboru pro asociační útok s pravidly a fragmentací slovníku.

assoc_rule_split_in			
soubor	*id*	*sticky*	*no_delete*
config	0	-	-
data	1	ano	ano
dict1	2	ano	ano
rules	3	-	-

Tabulka 6.3: Informace o přenášených souborech ze vzorového souboru distribuce podle pravidel

assoc_no_rule_in			
soubor	*id*	*sticky*	*no_delete*
config	0	-	-
data	1	-	-
dict1	2	-	-
rules	-	-	-

Tabulka 6.4: Informace o přenášených souborech ze vzorového souboru distribuce podle slovníku bez použití komolících pravidel

6.3 Výpočetní jednotka a Runner

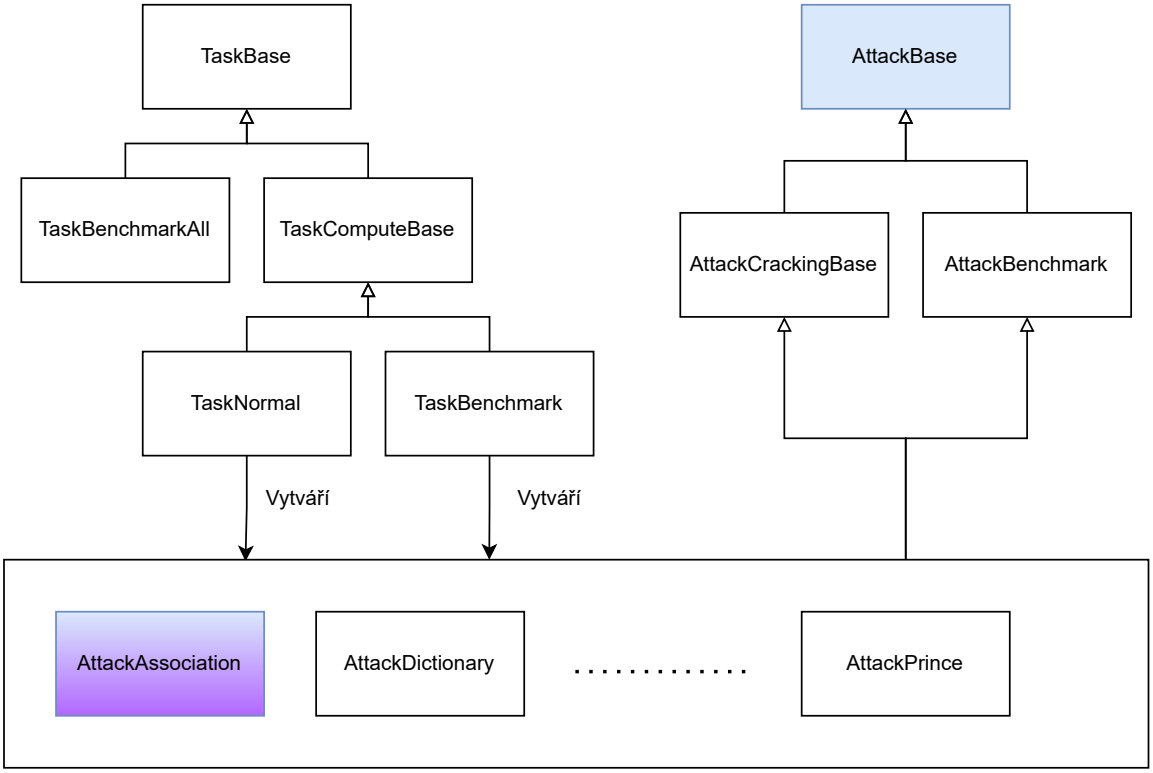
Na výpočetních jednotkách je za použití projektu BOINC spouštěna aplikace runner, která je zcela vlastní implementací middleware nad Hashcatem projektu Fitcrack. Runner není zodpovědný za stahování souborů, pouze za konstrukci parametrů a spouštění programu Hashcat. Struktura některých důležitých tříd je znázorněna na obrázku 6.6 s vyznačenými úpravami.

V souboru `Attack.cpp` jsou vytvářeny třídy různých útoků podle identifikačního čísla útoku z konfiguračního souboru. Tato funkce je „friend“ funkcí třídy `AttackBase` a tak je v diagramu 6.6 tato změna zaznačena v této třídě. Zde je přidána nová třída pro asociační útok `AttackAssociation` a jelikož její identifikátor „10“ zasahuje do dvouciferných čísel, funkce detekce útoku je předělána z porovnávání pouze prvního znaku na porovnávání celých řetězců identifikačního čísla přečteného z konfiguračního souboru.

Zmíněná nová třída se nachází v souboru `AttackAssociation.cpp` a jejím účelem je konstrukce parametrů programu hashcat z konfiguračního souboru. Tato funkce tedy přidává soubor s pravidly, je-li přítomen a slovník s nápovědami. Zvláštností této třídy je, že musí předefinovat identifikační číslo módu útoku z „10“ na „9“, které odpovídá asociačnímu útoku v parametru `-a` Hashcatu, číslo „9“ již ve Fitcracku náleží útoku PCFG. Pokud je výpočetní blok určený pro testování výkonu jednotky, je mód útoku parametrem `-a` znovu předefinovaný tentokrát na „0“ odpovídající slovníkovému útoku. Tento přístup je nutný, protože asociační útok předpokládá stejný počet nápověd a hashů, ovšem testovací bloky jsou identifikovány nulovým počtem nápověd (slov ve slovníku), což by odpovídalo také nulovému počtu hashů vytvářející chybu spuštění Hashcatu. Tato změna je zároveň možná, protože slovníkový útok je velmi podobný asociačnímu útoku a jejich výkon v hashích za sekundu by měl být stejný.

assoc_no_rule_alt_in			
soubor	*id*	*sticky*	*no_delete*
config	0	-	-
data	1	ano	ano
dict1	2	ano	ano
rules	**-**	-	-

Tabulka 6.5: Informace o přenášených souborech ze vzorového souboru distribuce použitím argumentů `-skip` a `-limit` bez použití komolících pravidel



Obrázek 6.6: Některé důležité třídy aplikace runner s vyznačenými třídami podstupujícími úpravu modře a novou třídou fialově.

Kapitola 7

Experimenty

Asociační útok se přístupem k lámání velmi liší od ostatních útoků a nemá tedy příliš smysl je porovnávat, přesto je alespoň porovnán se slovníkovým útokem, aby byl přiblížen rozdíl v rychlosti průchodu hashů, při použití jediného kandidátního hesla. Dalo by se říci, že asociační útok je vlastně dávka jednotlivých lámacích útoků a jeho síla tedy spočívá zejména v pohodlnosti.

V této kapitole jsou testovány zejména rozdílné metody distribuce, aby byla předvedena jejich užitečnost v různých situacích. Distribuce slovníků by měla být lehce vhodnější pro větší slovníky a menší soubory s pravidly, zatímco distribuce podle pravidel by měla naopak být lehce výhodnější pro menší slovníky s více pravidly.

7.1 Hardware

Testy byly provedeny na notebooku značky Dell modelu „Latitude E7450“. Uvedené experimenty byly prováděny se současně nejnovější verzí projektu Fitcrack 2.4.0. Uvedený notebook zároveň sloužil jako server a jako klient. Přičemž při běhu výpočtu byly všechny procesy na popředí ukončeny. Notebook má nasledující parametry:

- **Operační systém** – Linux Debian x86_64
- **CPU** – Intel i7-5600U 2.6 GHz
- **GPU** – Intel HD Graphics 5500
- **RAM** – 8 GB DDR3L 1600 MHz

7.2 Návrh experimentů

Pro účely testování byl vytvořen jednoduchý skript v jazyce Python, který generuje specifikovaný počet náhodných hesel o specifikované délce spolu s jejich otiskem vytvořeným pro několik různých hashovacích funkcí. Efektivně tak vytváří soubory s hashi a slovníky pro použití Hashcatem nebo Fitcrackem. Jsou vytvářeny Hashe MD5[1], SHA1[2], SHA256[3]

[1] https://www.rfc-editor.org/info/rfc1321
[2] https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf
[3] https://federalregister.gov/a/02-21599

a Bcrypt[4], tyto hashe jsou uvedeny v pořadí podle své výpočetní náročnosti. Experimentálně bylo ověřeno, že MD5 ani SHA1 nejsou vhodné hashe pro experimenty, jelikož jejich výpočetní náročnost je příliš nízká a vyžadují tedy obrovské množství hashů pro znatelné rozdíly. Ovšem již při použití 10 000 hashů se začíná Webadmin Frontend velmi znatelně zasekávat při zadávání nové úlohy, pravděpodobně kvůli validaci vstupů.

Za účely testování tedy byly použity hashovací funkce SHA256 a Bcrypt. SHA256 představuje relativně jednoduchou hashovací funkci. což umožňuje použití velkého množství hashů a v krátkém čase demonstrovat distribuci přes hashe s nápovědami. Bcrypt je naopak složitější šifra při použití více kol a je vhodná zejména pro ukázku distribuce jediného hesla s více pravidly aby pravidel nemuselo být zbytečně mnoho. Na Bcrypt hashích je asociační útok také předváděn v příspěvku na fóru, který zmiňuje vznik nového útoku[5].

Za účelem experimentování se škálovatelností je vytvořen kontrolní test bez použití projektu Fitcrack a pro simulaci distribuce je použito nastavení preferované doby výpočetních bloků u vytvořeného úkolu. Takto lze měnit počet vytvořených výpočetních bloků a zkoumat režii distribuce. Fitcrack Webadmin poskytuje celkový čas a čas strávený výpočtem na klientech. Při odečtu těchto hodnot vyjde čistá režie, část režie je ovšem ztracena překryvem s výpočtem. Tento výpočet lze provést pouze s jedním klientem, protože při použití více klientů může být čas strávený součtem jednotek větší než čas celkový. Tento experiment představuje škálovatelnost pro více klientů i přes použití pouze jednoho, jelikož více klientů pouze ovlivňuje rozdílnou velikost výpočetních bloků podle jejich výkonu a režie zůstává stejná, limit škálovatelnosti tedy nastává, když je některý výpočetní blok vždy dokončen dříve než je generátor schopen vygenerovat nový.

Pro účely testování rozdílných módů distribuce je zapotřebí spouštět úlohu se stejnými parametry rozdílnou pouze právě v módu distribuce. Teoreticky by měly všechny módy distribuce dosahovat stejné rychlosti lámání s tím, že distribuce celých slovníků bude trpět počáteční režií, ale limit škálovatelnosti bude vyšší, distribuce slovníků na serveru naopak bude s nižší počáteční režií, ovšem s nižším limitem škálovatelnosti. Distribuce pravidel na serveru je podobná distribuci slovníků na serveru, pouze umožňuje lépe dělit úlohy, kde je více pravidel než hashů. Distribuce na serveru také využívá více výpočetního výkonu serveru a méně zatěžuje síť než distribuce na klientech, ovšem těmto hodnotám se experimenty nevěnují.

7.3 Kontrolní vzorek samotného Hashcatu

Po krátké analýze projektu Fitcrack byl extrahován přesný příkaz spouštění Hashcatu na klientech v programu runner. Hashcat je spouštěn z metody `launchSubprocess()` v souboru `ProcessLinux.cpp` poté, co jsou sestavěny všechny argumenty příkazového řádku podle konfiguračního souboru ve funkcích jednotlivých útoků. Hashcat je spouštěn následujícím způsobem:

```
hashcat -a9 -m [hash-type] [hashes] --optimized-kernel-enable
    --hwmon-temp-abort 90 --restore-disable --potfile-disable
    --logfile-disable --status-timer=10 --quiet --status --status-json
    --outfile passwords.txt --outfile-format=1,3
    --rules-file [rules] [hints]
```

[4] https://www.usenix.org/legacy/events/usenix99/provos/provos_html/node1.html
[5] https://hashcat.net/forum/thread-9534.html

S těmito parametry byl Hashcat spuštěn nad stejnými daty mimo prostředí Fitcrack a BOINC s nástrojem `time`. Takto je možné získat kontrolní vzorek pro experimenty z nedistribuovaného případu. Je důležité podotknout, že u kontrolního nedistribuovaného případu je očekávána menší doba výpočtu způsobená chybějící režií distribuce. Účelem existence tohoto vzorku je zdůraznění zmíněné režie.

Pro získání kontrolního vzorku bylo porovnáno použití asociačního útoku a slovníkového útoku v samotném hashcatu. Oba módy byly spuštěny vícekrát za účelem získání průměrné doby výpočtu pro každý experiment. Pro první měření byla použita data s 10 000 hashi SHA256, pěti-číselným pepřem a pravidly pro 5 čísel a pro druhé měření byla použita složitější šifra bcrypt s 500 hashi a pěti koly bez použití pravidel, později je dopad použití pravidel naznačen.

První vzorek je naznačen v tabulce 7.1, je zvláštní, že v tomto případě dosahuje asociační útok nižších rychlostí než útok slovníkový, i když by měl být schopný dříve ukončit výpočet u každého hashe. Nicméně důležitým poznatkem z této anomálie je, že nízká rychlost asociačního útoku v projektu Fitcrack nebude pro tento hash způsobena chybou v implementaci rozšíření.

SHA256	**Asociační**	266.4	262.5	261.4	252.5	249.4	250.4	248.4	251.4	**255.3**
	Slovníkový	141.4	112.2	118.2	113.2	117.3	117.3	113.2	112.2	**118.1**

Tabulka 7.1: Sekundy výpočtu SHA256 hashů zkoumaných útoků v prostoru hesel o velikosti 1 111 110 000 s použitím pravidel.

Následující tabulka 7.2 používá stejného počtu hashů, ovšem tentokrát s rozdílnými solemi. Tento případ je uveden aby podtrhl skutečnou optimalizaci asociačního útoku, která spočívá v tom, že není zapotřebí slova ze slovníku zkoušet vůči všem solím. Aby byla vyvážena doba trvání experimentu je používáno 10-krát méně pravidel.

SHA256	**Asociační**	33.0	33.0	32.0	32.0	36.0	36.2	31.0	37.1	**33.8**
	Slovníkový	1089.6	1040.3	999.2	1003.2	1004.2	994.1	1004.3	998.8	**1016.7**

Tabulka 7.2: Sekundy výpočtu SHA256 hashů zkoumaných útoků v prostoru hesel o velikosti 111 111 000 s použitím pravidel a náhodných solí o délce 3.

Tabulka 7.3 obsahuje hashe Bcrypt, na kterých je ukazována síla asociačního útoku v příspěvku na fóru pojednávajícím o přidání útoku[6]. V tomto případě je opět asociační útok rychlejší, toho je dosaženo předčasným ukončením zbytečných výpočtů po nalezení správného hesla, protože hashe Bcrypt vždy obsahují vygenerovanou sůl. Tento test byl dále uskutečněn s použitím 100 a 1000 pravidel, přestože správná hesla jsou již obsažená ve slovníku. Tento následující test vedl na objevení další zvláštnosti v chování asociačního útoku. Použitím pravidel se samozřejmě zvětšil prostor kandidátních hesel stokrát a tisíckrát, což také vedlo k úměrnému zvýšení doby výpočtu nejen slovníkového, ale i asociačního útoku, přestože hned první pravidlo bylo pravidlo identity a umožňovalo tedy předčasné ukončení.

7.4 Výkon oproti slovníkovému útoku

Pro porovnání asociačního útoku byl zvolen útok slovníkový, který je asociačnímu konceptuálně a funkčně velmi podobný. Aby bylo možné módy útoku porovnat, jsou oba dva také

[6] https://hashcat.net/forum/thread-9534.html

Bcrypt	**Asociační**	6.8	7.0	6.8	7.0	7.0	7.1	7.0	6.9	**7.0**
	Slovníkový	96.0	95.9	94.9	94.9	98.1	96.0	94.9	95.9	**95.8**

Tabulka 7.3: Sekundy výpočtu Bcrypt s pěti koly hashů zkoumaných útoků v prostoru hesel o velikosti 500 bez použití pravidel.

testovány výše v sekci 7.3. Jelikož asociační útok nepřináší žádný nový způsob tvorby hesel nemá smysl testovat jeho úspěšnost v nalezení takových hesel, jak tomu může být například u útoků PCFG a PRINCE.

Za účelem jednoduchého porovnání byl proveden experiment s 10 000 hashi SHA256 a 100 000 pravidly na asociačním i slovníkovém úkolu. Byly zvoleny stejné módy distribuce, tj. distribuce slovníků na serveru. Preferovaná doba trvání výpočetního bloku byla nastavena na 60 sekund. Výsledky je možné vidět v tabulce 7.4. Z výsledků je vidět, že efektivita obou útoků je velmi podobná svým sekvenčním ekvivalentům.

SHA256 1B ks	**Asociační**	**čas celkem**	13:28	11:06	10:39	13:43	14:11	**12:25**
		čas výpočtu	6:51	4:47	4:45	6:29	6:22	**5:51**
		počet dílů	10	8	8	10	10	**9**
		efektivita	62%	89%	89%	66%	67%	**73%**
	Slovníkový	**čas celkem**	10:23	10:14	10:34	10:17	10:52	**10:28**
		čas výpočtu	3:41	2:43	2:47	2:46	2:50	**2:57**
		díly	7	7	7	7	7	**7**
		efektivita	53%	72%	71%	71%	69%	**67%**

Tabulka 7.4: Tabulka výsledků porovnání slovníkového a asociačního útoku.

Pro porovnání v podmínkách vhodných pro asociační útok byl proveden experiment s 10 000 hashi SHA256 s tří-cifernou solí. Bylo použito 1 000 pravidel za účelem zvětšení prostoru hesel a prodloužení trvání experimentu, aby byly relativně zastíněny vnější vlivy na výkon jednotky. Výsledky experimentu je možné vidět v tabulce 7.5. Jak je vidět velké množství unikátních solí je vhodné pro asociační útok a v takovém případě je rychlejší než útok slovníkový. Efektivita je měřena vůči nedistribuovanému řešení, které není zmíněno výše. Pro tento případ s těmito argumenty vychází sekvenční řešení asociačního útoku průměrně na 7 sekund a slovníkového útoku na 98 sekund. Asociační útok je tak rychlý, že v tomto případě trpí režií a tak má velmi nízkou efektivitu distribuce. I přes tuto nízkou efektivitu je stále rychlejší než útok slovníkový.

SHA256 10M ks 3 číslice soli	**Asociační**	**čas celkem**	8:54	8:42	8:27	8:49	8:58	**12:25**
		čas výpočtu	0:42	0:39	0:40	0:40	0:43	**0:51**
		počet dílů	6	6	6	6	6	**6**
		efektivita	17%	18%	18%	18%	16%	**73%**
	Slovníkový	**čas celkem**	10:41	10:54	10:56	11:08	11:02	**10:56**
		čas výpočtu	2:06	2:01	2:05	2:13	2:06	**2:06**
		díly	7	7	7	7	7	**7**
		efektivita	78%	81%	78%	74%	78%	**78%**

Tabulka 7.5: Tabulka výsledků porovnání slovníkového a asociačního útoku se solenými hashi SHA256.

7.5 Škálovatelnost a režie

V tabulce 7.6 jsou vidět hodnoty experimentu s režií vznikající tvorbou bloků. Počet tvořených bloků je regulován požadovaným časem na blok v nastavení úkolu. Test používá stejná základní data jako většina ostatních experimentů. Jedná se o 10 000 SHA256 hashů s 100 000 pravidly.

SHA256 1B ks	**60 s/wu**	**čas celkem**	13:28	11:06	10:39	13:43	14:11	**12:25**
		čas výpočtu	6:51	4:47	4:45	6:29	6:22	**5:51**
		počet dílů	10	8	8	10	10	**9**
		efektivita	62%	89%	89%	66%	67%	**73%**
	120 s/wu	**čas celkem**	13:10	13:12	13:08	13:09	13:08	**13:09**
		čas výpočtu	4:36	4:34	4:36	4:34	4:39	**4:36**
		počet dílů	6	6	6	6	6	**6**
		efektivita	%	%	%	%	%	**%**
	3600 s/wu	**čas celkem**	8:42	8:35	8:08	8:20	8:24	**:**
		čas výpočtu	4:04	4:14	4:07	4:09	4:08	**4:08**
		počet dílů	3	3	3	3	3	**3**
		efektivita	%	%	%	%	%	**%**

Tabulka 7.6: Tabulka výsledků porovnání slovníkového a asociačního útoku.

Jelikož SHA256 je relativně slabý hash, výpočet výše zmíněného experimentu byl relativně krátký na demonstraci režie, ovšem již v tomto počtu hashů byl formulář tvorby úkolů velmi zasekaný. Proto byl ještě vytvořen následující experiment s 500 hashi Bcrypt o čtyřech kolech a 1 000 pravidly pro další prodloužení výpočtu. Výsledky je možné vidět v tabulce 7.7. Je velmi zvláštní, že střední dělení bloků dosahuje konzistentně nejrychlejších výsledků. Předpokladem testu bylo, že nejrychlejší bude experiment s nejmenším počtem bloků, jelikož nevzniká velká režie. Z výsledků experimentu nejsem schopen učinit smysluplný závěr.

Bcrypt 500k ks	**60 s/wu**	**čas celkem**	8:02	8:09	8:14	8:12	8:08	**8:08**
		čas výpočtu	6:18	6:27	6:25	6:19	6:21	**5:51**
		počet dílů	8	8	8	8	8	**8**
	120 s/wu	**čas celkem**	5:50	6:27	5:58	6:11	6:08	**:**
		čas výpočtu	4:35	4:49	4:36	4:35	4:39	**:**
		počet dílů	4	4	4	4	4	**4**
	3600 s/wu	**čas celkem**	7:14	6:44	6:54	7:16	6:59	**7:01**
		čas výpočtu	5:54	5:42	5:39	5:44	5:40	**5:44**
		počet dílů	1	1	1	1	1	**1**

Tabulka 7.7: Tabulka výsledků porovnání slovníkového a asociačního útoku s použitím šifry Bcrypt.

Aby byla demonstrována počáteční režie úkolu, která dále neovlivňuje škálovatelnost, byl proveden krátký experiment s jedním hashem bez pravidel. Experiment ukázal, že takováto úloha průměrně trvá přesně minutu a půl, zatímco čas strávený na výpočetní jednotce se pohybuje kolem deseti sekund. Z tohoto experimentu a experimentů výše vyplývá, že režie je ovlivněna velikostí souborů, které byly pro případ SHA256 největší a pro poslední

krátký test zcela minimální. Počáteční režie vzniku úlohy a počátku jejího plánování je pak minimální, do jedné minuty.

7.6 Dělení podle pravidel

Experimenty s módy distribuce byly provedeny s běžným nastavením. V rámci experimentu s hashovací funkcí SHA256 bylo použito 10 000 hashů se 100 000 pravidly. Preferovaná doba výpočtu bloku byla nastavena na 60 sekund. Výsledky experimentu jsou zaznamenány v tabulce 7.8. Z výsledků vyplývá, že distribuce pravidel je porovnatelná s distribucí dělením slovníků na serveru. Pozorovatelnou zvláštností je poněkud malý počet bloků při dělení na hostech použitím `--skip` a `--limit`. U ostatních módů distribuce začíná generátor malými bloky dokud se nedopracuje na velikost trvající zhruba zadanou preferovanou minutu. Ovšem při dělení na klientech je hned první díl dvouminutový a druhý díl je prostě zbytek. Dle očekávání jsou všechny módy distribuce relativně porovnatelné a preference je tudíž na uživateli.

SHA256 1B ks	**On Server**	**čas celkem**	13:28	11:06	10:39	13:43	14:11	**12:25**
		čas výpočtu	6:51	4:47	4:45	6:29	6:22	**5:51**
		počet dílů	10	8	8	10	10	**9**
		efektivita	62%	89%	89%	66%	67%	**73%**
	On Hosts	**čas celkem**	11:29	11:32	11:01	8:46	8:49	**10:44**
		čas výpočtu	3:46	3:43	4:17	3:46	3:52	**3:53**
		díly	2	2	2	2	2	**2**
		efektivita	100%	100%	99%	100%	100%	**100%**
	By Rules	**čas celkem**	10:28	9:52	9:54	10:45	10:25	**10:17**
		čas výpočtu	5:18	5:14	5:18	6:07	5:14	**5:26**
		díly	7	7	7	9	7	**7**
		efektivita	80%	81%	80%	69%	80%	**78%**

Tabulka 7.8: Výsledky experimentu s módy distribuce s hashem SHA256 s miliardou kandidátních hesel. Pro každý mód distribuce je uvedena reálná doba trvání v minutách a sekundách, doba výpočtu na klientech v minutách a sekundách, počet bloků vzniklých distribucí, efektivita oproti nedistribuovanému hashcatu.

Za účelem demonstrace použitelnosti distribuce podle pravidel byl proveden test s jedním hashem Bcrypt o 4 iteracích a 100 000 pravidly. Účelem je demonstrovat, že při použití velké množiny pravidel nelze přesně dělit práci mezi výpočetní jednotky, obzvláště pak v případech, kdy je množství hashů řádově porovnatelné s množstvím výpočetních jednotek. Výsledky demonstrace jsou uvedeny v tabulce 7.9. Lze vypozorovat, že dělení na jednotky vytváří režii, ovšem jak bylo účelem ukázat, dělení podle pravidel přináší novou možnost dělení pro případy kde tomu jinak nebylo možno.

7.7 Výsledky

Z experimentů vyplývá, že asociační útok je ve své distribuci podobně efektivní jako útok slovníkový. Tento závěr potvrzuje předpoklad vzniklý z toho, že se jedná téměř o stejné útoky. Zároveň bylo předvedeno, že rozdíl v metodách distribuce na serveru je zanedbatelný a záleží především na velikosti souborů a uvážení uživatele, který použít. Při tomto expe-

Bcrypt	**On Server**	**počet dílů**	1	1	1
		čas na díl	1:29	1:58	3:10
100k ks	**On Hosts**	**počet dílů**	1	1	1
		čas na díl	1:29	1:30	1:31
60s/wu	**By Rules**	**počet dílů**	13	14	14
		čas na díl	0:17	0:15	0:18

Tabulka 7.9: Demonstrace použití distribuce pravidly na jednom hashi (malá množina) s velkou množinou hesel.

rimentu také vyšlo najevo, že mód distribuce používající `--skip` a `--limit` nerespektuje správně čas preferovaný pro výpočetní blok. Tato chyba zamezuje kompletnímu porovnání módů distribuce, jelikož tento mód bude generovat menší počet bloků a tím pádem bude mít menší režii.

Výkon a jeho přirovnání k ostatním útokům nemá smysl u asociačního útoku zkoumat. Přesto je ukázáno, že asociační útok dosahuje většího výkonu než útok slovníkový pouze pro hashe s mnoho různými solemi, které by slovníkový útok musel zkoušet ke každému záznamu ve slovníku. Bohužel z experimentů vyplývá, že asociační útok neukončuje předčasně průchod pravidly i když tuto možnost technicky má.

Kapitola 8

Závěr

V rámci diplomové práce byla nastudována architektura projektu Fitcrack vyvíjeného na Fakultě informačních technologií Vysokého učení technického v Brně. Za účelem implementace nového módu útoku byla zkoumána funkčnost tohoto útoku v aplikaci Hashcat, která je na pozadí projektu zodpovědná za výpočetní část lámání kryptografických hashů. Idea asociačního útoku pochází již z nástroje John the Ripper, ze kterého Hashcat vychází při své implementaci. Během testování byla v současné verzi Hashcatu objevena chyba pro asociační mód, která vyžaduje použití unikátních solí pro každý záznam ve slovníku, což kompletně znemožňuje použití nesolených hashů. Tato chyba pravděpodobně nastala záměnou mezi solemi a záznamy ve slovníku. Chyba ovšem nezamezuje implementaci zařazení útoku mezi módy Fitcracku.

Bylo navrženo rozšíření Fitcracku a jeho zásah do jednotlivých systémů. Z technické zprávy projektu byly nastudovány možnosti distribuce zátěže a byl navržen nový systém distribuce využívající dělení komolících pravidel mezi výpočetní jednotky.

Navržené rozšíření projektu bylo implementováno i přes některé problémy, které se během implementace objevily, zejména problémy s testováním výkonu jednotek pro asociační útok, který vyžaduje nenulový „keyspace“, kterým se běžně testovací úkoly vyznačují a jsou podle něj rozpoznávány, problémům s implementací distribuce fragmentací pravidel a občasným problémům s aktualizacemi projektu, jež způsobovaly obskurní chyby v různých částech projektu.

Účel útoku spočívá hlavně v pohodlné možnosti tvorby dávky a nelze jej tedy jednoduše přirovnávat k jiným útokům. Experimenty byly navrženy alespoň pro jednoduché porovnání rychlosti průchodu velkého množství hashů mezi asociačním útokem a slovníkovým útokem a pro porovnání jednotlivých módů distribuce na jedné výpočetní jednotce. Jak bylo teoretizováno z experimentů vyplynulo, že distribuce asociačního útoku je podobně efektivní jako distribuce útoku slovníkového. Bylo ukázáno, že optimalizace asociačního útoku spočívá v přeskakování nepotřebných solí, ovšem nedisponuje předčasným ukončením použití komolících pravidel. Zároveň bylo ověřeno, že nově navržený mód distribuce dělením pravidel na serveru je podobně efektivní jako dělení slovníků serverem a jeho použití je tedy doporučeno v případě řádově většího počtu pravidel než hashů.

Další pokračování v tomto rozšíření projektu Fitcrack vidím například v možnosti zadávat slovník manuálně přímo při tvorbě úkolu ve formuláři útoku. Samozřejmě projekt Fitcrack samotný je velmi rozsáhlý a umožňuje implementaci více různých rozšíření, na některé tyto možnosti jsou již vypsány jiné diplomové práce.

Literatura

[1] *Fitcrack* [online]. [cit. 2022-12-03]. Dostupné z: https://fitcrack.fit.vutbr.cz/.

[2] *Hashcat Wiki* [online]. [cit. 2022-12-03]. Dostupné z: https://hashcat.net/wiki/.

[3] Anderson, D. P. BOINC: A system for public-resource computing and storage. In: *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. Pittsburgh, PA, USA: [b.n.], November 2004, s. 4–10. DOI: 10.1109/GRID.2004.14. ISBN 0-7695-2256-4.

[4] Aumasson, J.-P. *Serious cryptography*. San Francisco, CA: No Starch Press, listopad 2017.

[5] Craig. *Securing Passwords with Salt, Pepper and Rainbows* [http://www.barkingiguana.com/2009/08/03/securing-passwords-with-salt-pepper-and-rainbows/]. 2009 [cit. 2023-3-10].

[6] Crumpacker, J. R. *Distributed password cracking*. 2009. Diplomová práce. Naval Postgraduate School, Monterey, California, USA.

[7] Florencio, D. a Herley, C. A large-scale study of web password habits. In: *Proceedings of the 16th international conference on World Wide Web*. ACM, Květen 2007. DOI: 10.1145/1242572.1242661. Dostupné z: https://doi.org/10.1145/1242572.1242661.

[8] Grassi, P. A., Fenton, J. L., Newton, E. M., Perlner, R. A., Regenscheid, A. R. et al. *Digital identity guidelines: authentication and lifecycle management*. červen 2017. Dostupné z: https://doi.org/10.6028/nist.sp.800-63b.

[9] Hashtopolis. *Hashtopolis Wiki*. [cit. 2023-05-10]. Dostupné z: https://github.com/hashtopolis/server/wiki.

[10] Horálek, J., Holík, F., Horák, O., Petr, L. a Sobeslav, V. Analysis of the use of Rainbow Tables to break hash. *Journal of Intelligent & Fuzzy Systems*. IOS Press. leden 2017, sv. 32, č. 2, s. 1523–1534. DOI: 10.3233/jifs-169147. Dostupné z: https://doi.org/10.3233/jifs-169147.

[11] Hranický, R. *Digital Forensics: The Acceleration of Password Cracking*. Brno, CZ, 2022. Disertační práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/phd-thesis/890/.

[12] Hranický, R., Lištiak, F., Mikuš, D. a Ryšavý, O. On Practical Aspects of PCFG Password Cracking. In: *Data and Applications Security and Privacy XXXIII*.

Springer International Publishing, 2019, s. 43–60. DOI: 10.1007/978-3-030-22479-0_3. Dostupné z: https://doi.org/10.1007/978-3-030-22479-0_3.

[13] Hranický, R., Zobal, L., Večeřa, V., Múčka, M., Horák, A. et al. *The architecture of Fitcrack distributed password cracking system, version 2* [online]. [cit. 2022-12-03]. Dostupné z: https://fitcrack.fit.vutbr.cz/files/doc/TR_Fitcrack_architecture.pdf.

[14] Katz, J. a Lindell, Y. *Introduction to modern cryptography*. 2. vyd. Boca Raton, FL: CRC Press, listopad 2014. Chapman & Hall/CRC Cryptography and Network Security Series.

[15] Li, W. a Zeng, J. Leet Usage and Its Effect on Password Security. *IEEE Transactions on Information Forensics and Security*. 2021, sv. 16, s. 2130–2143.

[16] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*. červenec 2012. Dostupné z: https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[17] Munshi, A. The OpenCL specification. In: *Proceedings of the 21st IEEE Hot Chips Symposium (HCS)*. Standford, CA, USA: [b.n.], August 2009, s. 1–314.

[18] OpenWall. *John the Ripper* [online]. [cit. 2022-12-03]. Dostupné z: https://www.openwall.com/john/.

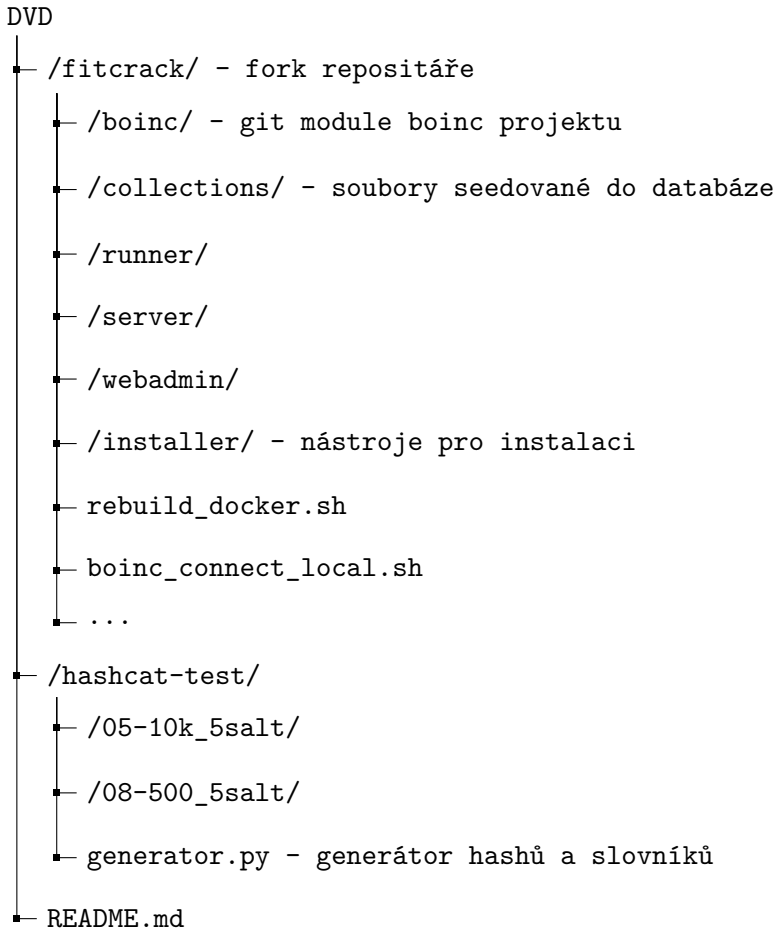
[19] OpenWall. *John the Ripper's cracking modes* [online]. [cit. 2022-12-03]. Dostupné z: https://www.openwall.com/john/doc/MODES.shtml.

[20] Passware, Inc.. *Passware Kit Forensic*. February 2010 [cit. 2023-05-03]. Dostupné z: https://www.passware.com/kit-forensic/.

Příloha A

Obsah přiloženého paměťového média

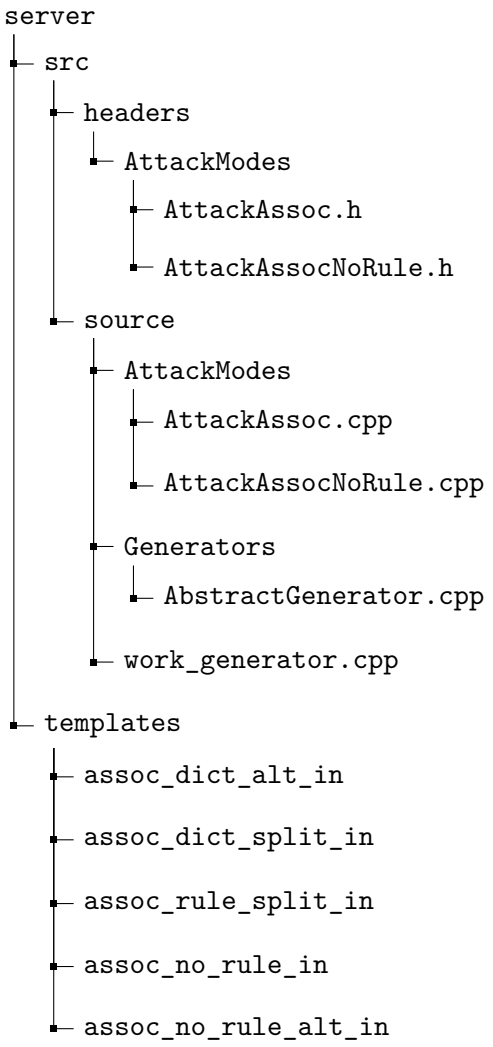
Následující příloha B popisuje strukturu souborů jednotlivých aplikací projektu Fitcrack, proto je v této příloze struktura těchto aplikací dále nerozvinuta.



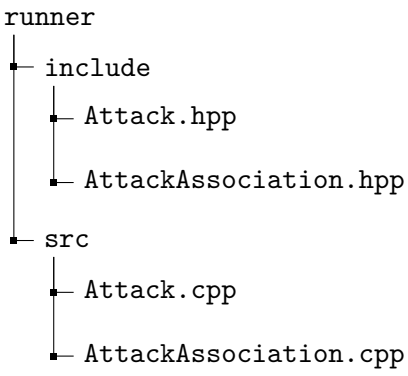
Obrázek A.1: Struktura souborů BOINC serveru

Příloha B

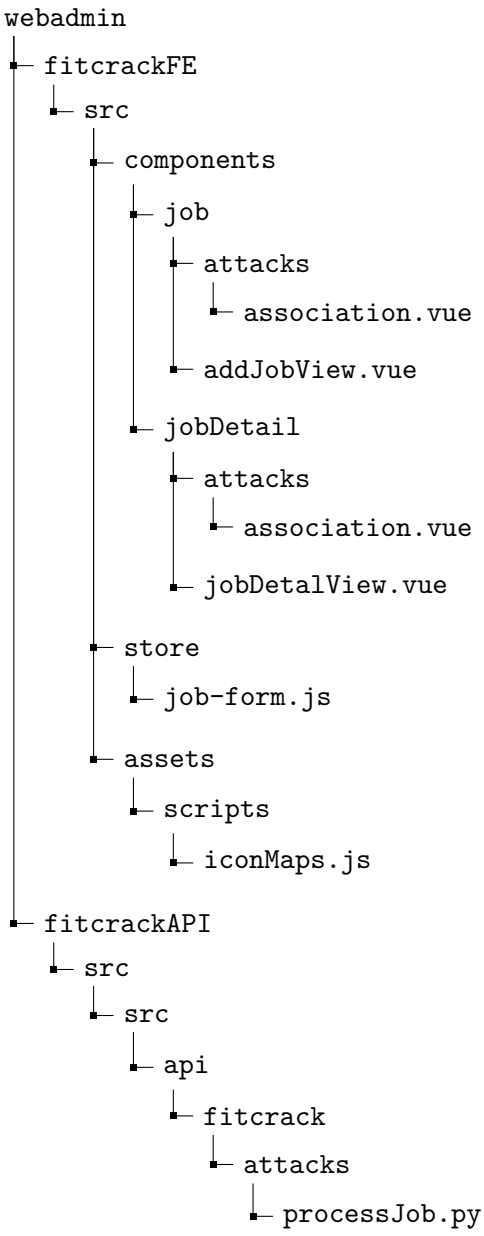
Struktura ovlivněných souborů projektu



Obrázek B.1: Struktura souborů BOINC serveru



Obrázek B.2: Struktura souborů middlewaru *runner*



Obrázek B.3: Struktura souborů webové aplikace *Webadmin*