Distribuovaná obnova hesel

Skupina Zabezpečná data

Technická zpráva FIT VUT v Brně

Radek Hranický, Lukáš Zobal, Vojtěch Večeřa

Technická zpráva č. FIT-TR-2017-04
Fakulta informačních technologií, Vysoké učení technické v Brně

Last modified: 2. ledna 2018

Distribuovaná obnova hesel

Radek Hranický, Lukáš Zobal, Vojtěch Večeřa

Vysoké učení technické v Brně, email:
ihranicky@fit.vutbr.cz, {xzobal00,xvecer18}@stud.fit.vutbr.cz

Abstrakt Šifrovaná data představují stále větší výzvu v oblasti digitální forenzní analýzy. Do této domény spadají jak šifrované dokumenty a archivy, tak diskové oddíly, či šifrovaný síťový provoz. Pro získání dat v otevřené podobě je nutné znát šifrovací klíč, který bývá typicky odvozen z hesla zadaného uživatelem. Pro útok na takové zabezpečení s cílem nalézt heslo dnes dominují přístupy založené na GPGPU, či využití jiných metod hardwarové akcelerace. Se vznikem stále dokonalejších technik zabezpečení dat však přestává být útok s využitím jednoho stroje prakticky použitelný a mnohdy je jediným řešením distribuovaný výpočet. Cílem této zprávy je přiblížit technologie použitelné pro distribuovanou obnovu hesel a představit prototyp řešení, kterým je distribuovaný systém Fitcrack.

1 Úvod

Při forenzní analýze digitálních dat narážíme stále častěji na média, která jsou určitým způsobem šifrována. Cílem šifrování daného média je zamezit neoprávněným osobám přistupovat k citlivým datům uvnitř. Příkladem šifrovaných médií jsou např. soubory MS Office, dokumenty PDF, archivy ZIP, RAR, či 7-Zip, nebo diskové oddíly šifrované nástroji jako TrueCrypt, VeraCrypt, apod. Pro dešifrování tavého média je potřeba znát heslo, na základě kterého je vygenerován šifrovací klíč [9, 7].

Nelze-li heslo získat tradičními metodami (např. vyzrazení majitelem), je jediným řešením *obnova hesel* (password recovery) silou, známá též jako *lámání hesel* (password cracking).

Dosavadní výzkum ukázal vysoký potenciál distribuované obnovy hesel s využitím GPGPU (General-purpose computing on graphics processing units) [10, 13]. Při výpočtu na jednom fyzickém stroji sice odpadá nutnost distribuce výpočtu a synchronizace výpočetních uzlů, tento způsob však přináší řadu omezení. Tato omezení pak limitují zejména výpočetní výkon, kterého je možné dosáhnout [6]. Pro náročnější úlohy je tedy nezbytné využít distribuovaného výpočtu.

Kim a kol. představili základní myšlenku využití modelu controller-worker, kdy experimentálně prokázali jeho využitelnost při útoku na zabezpečení dokumentů PDF 7.0 s využitím 88 fyzických uzlů obsahujících celkem 342 zařízení GPU. S využitím distribuovaného výpočtu a GPU acelerace se podařilo dosáhnout nalezení číselného hesla o délce 12 znaků za 7 dní a 15 hodin [13].

Apostal a kol. experimentovali s distribuovaným slovníkovým útokem pomocí technologií MPI a CUDA, přičemž došlo ke srovnání více možných alternativ rozdělení slovníků mezi výpočetní uzly [2]. Zou a kol. představili model distribuované obnovy hesel pro útok hrubou silou a slovníkový útok. Společně pak došli k závěru, že optimalizace výpočtu samotných kryptografických algoritmů zdaleka nestačí a je potřeba věnovat dostatek prostoru vývoji vhodných algoritmů pro distribuci výpočtu [18].

Srovnáme-li existující řešení pro obnovu hesel, pro výpočet obnovy hesel na jednom stroji pomocí GPGPU se v současnosti jeví jako nejrychlejší nástroj hashcat[1]. Jeho tým získal 1. místo na soutěži CMYC[2] ("Crack me if you can"), která proběhla v rámci konference DEFCON[3] 2012. I přes značnou popularitu a potenciálně vysoký dosažitelný výpočetní výkon tento nástroj nativně nepodporuje distribuovaný výpočet.

Pro distribuovanou obnovu hesel existuje množství softwarových řešení. Z komerčních jsou to např. ElcomSoft Distributed Password Recovery[4], Passware Kit[5], OctoPass[6], či AccessData Forensics Toolkit[7]. Tyto nástroje se však vyznačují poměrně vysokou pořizovací cenou, licence jsou typicky limitovány počtem výpočetních uzlů a ne vždy lze hovořit o univerzálním řešení. Např. software od společnosti Elcomsoft je určen pouze pro OS Windows, pro některé podporované formáty využití GPU neumožňuje vůbec, přičemž pro řadu zbývajících formátů nepodporuje technologii OpenCL, ale pouze NVIDIA CUDA, kdy nelze použít GPU od společnosti AMD, apod. Často se také jedná o řešení určená pro důvěryhodné prostředí v síti LAN a nejsou určeny pro distribuci skrze síť Internet [6].

Zatímco komerční nástroje pro distribuovanou obnovu hesel využívají pro samotný výpočet kryptografických algoritmů proprietárních řešení, software šířený jako open-source typicky funguje jako nádstavba nad nástrojem hashcat. Zástupcem takového řešení byl např. nástroj Hashtopus[8], jehož vývoj byl ukončen a následně na něj navázal nástroj Hashtopussy[9]. Ačkoli Hashtopussy poskytuje řadu zajímavých funkcí jako např. nastavení priority úloh, či notifikace pomocí e-mailu, spouštěcí parametry nástroje hashcat je nutné nastavit ručně, což poněkud omezuje cílovou skupinu uživatelů.

Do specifické kategorie pak spadá nástroj Hashstack[10]. Tento software poskytuje velice přívětivé uživatelské rozhraní, které umožňuje mj. sledování teploty

[1] https://hashcat.net/
[2] http://contest-2012.korelogic.com/stats.html
[3] https://www.defcon.org/
[4] https://www.elcomsoft.com/edpr.html
[5] http://www.lostpassword.com/distributed-password-recovery.htm
[6] http://lastbit.com/octopass.asp
[7] http://accessdata.com/solutions/digital-forensics/forensic-toolkit-ftk
[8] https://github.com/curlyboi/hashtopus
[9] https://github.com/s3inlc/hashtopussy
[10] https://sagitta.pw/software/

a zátěže procesorů na jednotlivých uzlech. Software je ovšem dodávat pouze s hardwarem od společnosti Sagitta a není možné jej zakoupit samostatně.

Cílem této technické zprávy je analyzovat technologie použitelné pro distribuovanou obnovu hesel a navrhnout řešení typu open-source, které by eliminovalo nevýhody výše uvedených nástrojů. V sekci 2 jsou diskutovány technologie MPI, Apache Hadoop, BOINC a jejich použitelnost v této oblasti. Sekce 3 pak představuje návrh distribuovaného systému Fitcrack, kde je přiblížen princip výpočtu pomocí technologie BOINC, generování dílčích úloh a jejich distribuci s využitím adaptivního plánování. Detailnější popis systému Fitcrack je pak obsažen v přílohách. Příloha A představuje jednotlivé funkční bloky systému, jejich účel a rozhraní, které je vzájemně propojuje. Příloha B pak zahrnuje popis struktury databáze, která je využívána v serverové části systému.

2 Technologie pro distribuované výpočty

Tato sekce se zaměřuje na srovnání existujících technologií pro distribuované výpočty a jejich použitelnost pro účely obnovy hesel na více uzlech. Jako hlavní kritéria uvažujeme:

- **dostatečný výkon** - nízké hardwarové nároky pro samotnou distribuci, dostatek výpočetních prostředků pro užitečný výpočet,
- **efektivní výpočet** - minimum času stráveného komunikací a synchronizací uzlů,
- **variabilní počet uzlů** - možnost měnit podobu distribuované sítě i za běhu výpočtu,
- **dobrá škálovatelnost** - co nejblíže lineární,
- **zotavení z chyb** - možnost návratu do konzistentního stavu při výpočetní chybě (např. selhání uzlu),
- **bezpečnost** - možnost nasazení i v nedůvěryhodném prostředí (Internet).

2.1 MPI

Message Passing Interface (MPI) je specifikace protokolu a také knihovna poskytující efektivní možnost hrubého rozdělení pracovní úlohy mezi více výpočetních uzlů, přičemž každý uzel zpracovává podmnožinu vstupních dat. V případě MPI mohou být za uzly považovány počítače, jádra procesoru, či vlákna.

Kang a kol. ukázali, že technologie MPI poskytuje výborný výkon při provádění výpočetně náročných operací nad menšími objemy dat [11]. Obnova hesel je zástupcem takového typu úloh, protože k prolomení zabezpečení většiny známých formátů typicky není nutné dešifrovat všechna data, ale často pouze k výstupu příslušné hešovací funkce nalézt její vstup. I přes relativně nízký objem dat tedy tento přístup vykazuje vysokou výpočetní náročnost. Skutečnost, že Apostal a kol. použili pro své řešení založené na platformě CUDA právě technologii MPI, má tedy racionální opodstatnění [2].

V konstrastu s výše uvedeným použili Reyez-Ortiz a kol. technologii MPI pro řízené strojové učení, při kterém byly zpracovávány velké objemy dat. Použitým

hardwarovým řešením byl výpočetní cluster Beowulf[11]. Výsledkem experimentu bylo zjištění, že pro uvedenou úlohu vykazovala technologie MPI zhruba 10x vyšší výpočetní výkon než Apache Hadoop na platformě Spark[12] [15]. Vysoký výkon a skvělou škálovatelnost se nám podařilo ověřit při experimentech s integrací OpenMPI do nástroje Wrathion [14].

Pro MPI existuje více implementací, které zahrnují jak open-source řešení jako např. OpenMPI[13], nebo komerční řešení, např. Intel MPI. OpenMPI zahrnuje také řadu funkcí pro zotavení z chyb: lokální nebo distribuovan checkpointing, detekce chyby komunikační sítě, apod. Původní koncept MPI neuvažuje dynamické přidávání uzlů za běhu výpočtu. Až ve specifikaci MPI verze 2.0 byla přidána funkce `MPI_Comm_spawn()`, která vytvoří nový výpočetní proces a to i na novém uzlu. U MPI však nově přidané uzly nejsou rozpoznány automaticky a je tedy v režii programátora, aby jejich detekci implementoval - např. pomocí speciálního procesu, který periodicky kontroluje nově připojené uzly.

Ačkoli současné implementace MPI v principu podporují komunikaci přes síť Internet, neposkytují žádné další prostředky pro výpočet v nedůvěryhodném prostředí. MPI jako takové nezahrnuje prostředky pro autentizaci, či šifrování této komunikace. Jakékoli zabezpečení přenosu je tedy nutné řešit jiným způsobem [5].

MPI jako technologie pro distribuované výpočty vykazuje vysoký výkon a škálovatelnost. Je tedy vhodným prostředkem pro řešení výpočetně náročných problémů jako je např. obnova hesel. Přes tato pozitiva je však výpočet omezen na pevný počet uzlů a důvěryhodné prostředí.

2.2 Apache Hadoop

Apache Hadoop[14] představuje řešení pro distribuované operace nad velkými objemy dat. Tato technologie lze použít pro řešení problémů, které lze popsat pomocí programovacího modelu *MapReduce*: Vstupní data jsou zpracována funkcí `Map()`, která provede jejich filtrování a seřazení. Tato funkce transformuje data v podobě dvojic $(K_1, V1)$ na dvojice (K_2, V_2), kde K_1, K_2 jsou klíče a V_1, V_2 hodnoty. Následně pomocí funkce `Reduce()` dojde ke sjednocení všech hodnot se stejným klíčem K_2.

Implementace této technologie využívá svůj vlastní systém souborů Hadoop Distributed File System (HDFS), který nativně podporuje šifrování a replikaci dat. Hadoop také poskytuje techniky pro zotavení z chyb, řadu užitečných nástrojů pro analýzu dat a v neposlední řadě možnost přidávání nových uzlů i za běhu výpočtu[15].

Apache Hadoop je využíván také v oblasti forenzní analýzy digitálních dat. Chen a kol. s jehov využitím navrhli a implementovali systém Collaborative

[11] http://www.beowulf.org/
[12] https://spark.apache.org/
[13] https://www.open-mpi.org/
[14] http://hadoop.apache.org/
[15] http://hadoop.apache.org/docs/

Network Security Management System (CNSMS) sloužící pro paralelní sběr a analýzu velkého množství síťového provozu. Tento systém je schopen v reálném čase detekovat, dohledávat a sledovat důkazy o nežádoucích aktivitách jako je phishing, spam, šíření virů, červů, apod. Chen a kol. tvrdí, že CNSMS je schopen poskytnout síti i ochranu proti útokům typu DDoS [3]. Cho a kol. doporučují použití technologie Apache Hadoop pro účely forenzní analýzy digitálních dat a vytvořili řadu instrukcí a doporučení pro forenzní analýzu cloudových systémů [4].

Podpora výpočtů typu MapReduce nicméně existuje i pro technologii MPI v podobě knihovny MPI MapReduce[16] (MMR). Roussev a kol. porovnali Apache Hadoop právě s MMR, u kterého dosáhli lepší škálovatelnosti a vyššího výkonu [16]. Reyez-Ortiz a kol. také upozornili na nižší výkon technologie Hadoop oproti řešením založeným na MPI, nicméně uznali, že Hadoop nabízí jednodušší nasazení, přidávání uzlů za běhu výpočtu, řadu užitečných nástrojů a další výhody spojené především s HDFS - tedy výše uvedenou replikaci dat a další [15].

Ačkoli Apache Hadoop vykazuje nižší výkon než MPI, bohatě jej kompenzuje dalšími funkcemi, které MPI nemá. Ačkoli existují pokusy[17] o distribovaný výpočet kryptografických hešů s použitím Apache Hadoop, není obnova hesel ideálním problémem pro model MapReduce. Dalším problémem Apache Hadoop pro obnovu hesel je vyšší režie oproti MPI ze které plyne nižší výpočetní výkon.

2.3 BOINC Framework

Berkeley Open Infrastructure for Network Computing (BOINC) je platformou pro distribuované výpočty, která nativně podporuje dynamické množství uzlů propojených přes Internet [1]. Technologie byla vyvinuta laboratořemi U.C. Berkeley Space Sciences Laboratory a je šířena jako open-source řešení pod licencí GNU GPL.

Primárním účelem BOINCu je veřejné sdílení výpočetních prostředků pro účely vědeckých výpočtů v oblastech jako je meteorologie, medicína, astrofyzika a další. BOINC umožňuje dobrovolníkům poskytnout své výpočetní kapacity a podpořit tak některé z existujících projektů. Tento princip se nazývá *volunteer computing*. Druhou doménou využití BOINCu je tzv. *grid computing*, tedy zapojení množství počítačů a výpočetních středisek v geograficky oddělených lokalitách.

Výpočetní síť tvoří server/servery a množství klientů. Zatímco server zodpovídá za řízení, přiřazování a plánování výpočetních úloh, klient představuje pracovní uzel provádějící samotný výpočet. BOINC funguje na principu klient-server, kdy server je pasivní a čeká na připojení klienta, po připojení může klient požádat server o přiřazení pracovní úlohy.

Problém, či sada problémů tvoří tzv. *projekt*. V rámci projektu je vytvářeno množství pracovních úloh, které jsou následně přiřazovány klientům, které o ně požádají. Jakmile je vytvořen projekt, klienti se mohou připojit (volitelně též

[16] http://mapreduce.sandia.gov/

[17] http://en.cnki.com.cn/Article_en/CJFDTOTAL-TXBM201111026.htm

s autentizací) a participovat na výpočtu. BOINC podporuje také vytváření úloh "na míru" pro konkrétní klienty. Toho lze využít např. k rovnoměrnému rozložení zátěže, či implementace vhodné plánovací strategie pro řešení daného typu problému.

Pro řešení pracovních úloh může být potřeba jedna či více aplikací. BOINC proto zajišťuje automatickou distribuci příslušných spustitelných binárních souborů na koncové stanice. Každá aplikace je vždy dodávána pro konkrétní architekturu operačního systému daného klienta. Pokud server eviduje novou verzi některé aplikace, BOINC zajistí následnou aktualizaci též na straně klienta.

Jakmile klient obdrží binární soubory aplikací a zadání úlohy včetně vstupních dat, může zahájit výpočet. Doba výpočtu jedné úlohy závisí na implementaci daného projektu, přičemž může trvat minuty, hodiny, ale také dny. Jakmile klient dokončí výpočet, informuje o této skutečnosti server pomocí zprávy obsahující informace o řešení úlohy a výstupní soubory, jejichž obsah nemá standardizovaný formát, ale odvíjí se od specifikace a implementace řešeného projektu.

Narozdíl od MPI je BOINC přímo navržen pro distribuovaný výpočet přes síť Internet. K tomuto účelu poskytuje řadu bezpečnostních mechanismů umožňujících výpočet v nedůvěryhodném prostředí, konkrétně např. autentizaci, správu uživatelských účtů, podepisování zpráv, replikaci úloh, či kryptografii veřejným klíčem.

Klienti se v prostředí BOINC mohou připojovat i opojovat za běhu výpočtu. BOINC nabízí řadu možností pro zotavení z chyb, přičemž vždy závisí na implementaci projektu. V případě neúspěšného výpočtu některé úlohy je typicky tato úloha přiřazena jinému výpočetnímu uzlu. Konkrétní uzly také může projekt rozlišovat na důvěryhodné, nedůvěryhodné, případně vytvářet skupiny uzlů se společnými specifiky.

Klientská stanice může být v jeden moment připojena k více projektům současně. Na klientské straně lze specifikovat, které procesory a do jaké míry se zapojí do výpočtu, kdy se má výpočet provádět, apod. Případem užití může být společnost s množstvím počítačů, které v pracovní době slouží k běžné činnosti a po pracovní době začnou participovat na projektu a stanou se tak součástí virtuálního superpočítače.

Před publikováním našeho článku *On Efficiency of Distributed Password Recovery* [8] jsme nenalezli žádnou zmínku o použití technologie BOINC pro účely forenzní analýzy. Kasabov a kol. nicméně zmínili možnost použití této technologie pro obnovu hesel, přičemž jako hlavní výhody prostředí BOINC zmínili spolehlivost a robustnost takového řešení. Kasabov a kol. také zmínili nutnost implementace vhodného softwarového řešení pro server a nasazení výkonné aplikace s akcelerací pomocí GPU na straně klienta [12]. Právě na základě výše uvedeného jsme se rozhodli naše řešení pro distribuovanou obnovu hesel postavit na technologii BOINC.

Mimo samotnou obnovu hesel existoval v této souvislosti projekt DistrRTgen[18], který na základě dobrovolného sdílení výpočetních zdrojů prováděl tvorbu

[18] http://boinc.berkeley.edu/wiki/DistrRTgen

volně dostupných *rainbow tables* [17]. V současnosti však stránka projektu není funkční a projekt samotný je označen jako soukromý.

3 Systém Fitcrack

Fitcrack je distribuovaný systém pro obnovu hesel šifrovaných médií a prolomení kryptografických hešů. Díky propojení s nástrojem hashcat[19] dosahuje jak obrovské škály podporovaných formátů, tak špičkové rychlosti při použití odpovídajícího hardwaru. Výpočet je prováděn pomocí technologie OpenCL a je možné jej provozovat na všech kompatibilních CPU, GPU, FPGA, DSP, či koprocesorech. Fitcrack staví na open-source frameworku Berkeley Open Infrastructure for Network Computing (BOINC[20]), který přináší v řízení distribuovaného výpočtu značnou automatizaci při zachování flexibility systému a svobody administrátora v otázce konfigurace výpočetních uzlů a hardwarového i softwarového řešení distribuované sítě.

Základem distribuované výpočetní sítě je jeden, či více projektových serverů. Tyto zajišťují správu a plánování úloh, dohled nad výpočetními uzly, řízení distribuovaného výpočtu a zpracování výsledků. Klientskou část poté tvoří aplikace BOINC Client, kterou lze volitelně doplnit grafickým administračním nástrojem BOINC Manager, které umožňuje jednoduchou konfiguraci cílové stanice. Na každé stanici je možno určit, které dny a hodiny se má podílet na výpočtu a které CPU/GPU a v jaké míře bude uzel používat. Výpočet samotný provádí nástroj hashcat, který se zbytkem systému komunikuje pomocí aplikace Fitcrack Runner.

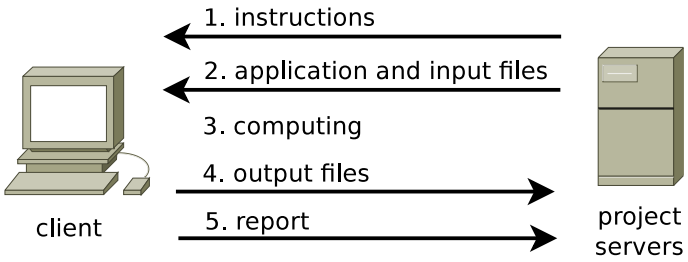
3.1 Princip výpočtu

Fitcrack funguje na principu klient-server, kde poskytovanou službou je jedna pracovní *úloha* (job, workunit). Jedna či více úloh pak tvoří jeden pracovní *balíček* (package), který představuje konkrétní útok na konkrétní šifrované médium. Serverová část běží na samostatném stroji a jejím účelem je generování úloh a správa distribuované sítě. Klientská část běží v jedné instanci na každém výpočetním uzlu a jejím účelem je řešení přiřazených úloh – tedy výpočet kryptografických hešů pomocí nástroje hashcat.

Klient po připojení k serveru (volitelně s autentizací) požádá o novou úlohu. Server na základě konfigurace a aktuálního stavu řešených pracovních balíčků vytvoří a přiřadí klientovi dílčí úlohu z některého z pracovních balíčků. Úlohy jsou tvořeny na míru konkrétním uzlům na základě výpočetní náročnosti, výkonu uzlu, konfigurace serveru a zvolené distribuční strategii. Zjednodušený průběh komunikace mezi klientem a serverem/servery ilustruje obrázek 1.

Pro připojení nového uzlu do sítě stačí na cílovou stanici nainstalovat aplikaci BOINC Client a připojit se k projektovému serveru. Veškeré další soubory

[19] https://hashcat.net/hashcat/
[20] https://boinc.berkeley.edu/



Obrázek 1. Komunikace klient-server v systému BOINC

(vstupní data, binární soubory) jsou staženy automaticky, včetně případné aktualizace na novější verze. Pro přidávání a správu běžících úloh slouží nástroj Fitcrack WebAdmin, který poskytuje správci kontrolu a dohled nad výpočty v distribuované síti.

3.2 Architektura systému

Jak serverovou, tak klientskou část lze rozdělit na množství funkčních bloků, které spolu navzájem komunikují skrze specifická rozhraní. Komunikaci klient-server definuje protokol BOINC scheduling server protocol, syntakticky vycházející z formátu XML. Jeho zprávy jsou přenášeny pomocí HTTP nebo HTTPS nad TCP/IP. Architektura systému je vyobrazena na obrázku 2. Detailní popis jednotlivých funkčních bloků a jejich rozhraní je uveden v příloze A.

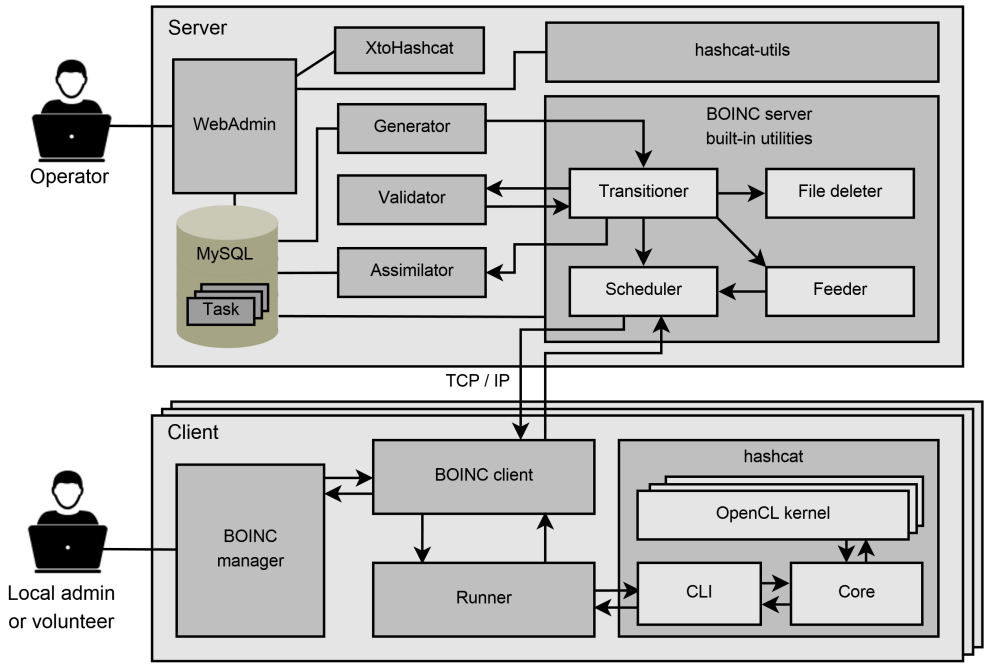
3.3 Distribuce výpočtu na základě indexů hesel

Cílem automatizovaného útoku na kryptografické zabezpečení libovolného šifrovaného média je systematické zkoušení možných hesel, přičemž cílem je nalézt správné heslo.

Uvažujme, že každé heslo p je řetězcem nad abecedou Σ, tedy $p \in \Sigma^*$. Množinu $P \subset \Sigma^*$ pak nazýváme *stavovým prostorem* všech možných hesel. Mohutnost i konkrétní podoba P závisí na konkrétním typu útoku. Pro účely distribuovaného výpočtu však uvažujeme, že P je vždy konečnou uspořádanou množinou. Na základě výše uvedeného můžeme definovat funkci zvanou *generátor hesel* $g(i) : N \mapsto P$, kde $i \in \langle 0, |P| - 1\rangle$ a i nazýváme *index hesla*.

Předpokládejme jednoduchý inkrementální *útok hrubou silou*, kdy chceme generovat všechna hesla délek 1 až 3 nad abecedou $\Sigma = \{a, b, c, \ldots, z\}$. Pak platí:

$$\begin{aligned} g(0) = a,\ \ldots,\ g(25) = z \\ g(26) = aa,\ \ldots,\ g(701) = zz \\ g(702) = aaa,\ \ldots,\ g(18277) = zzz. \end{aligned} \tag{1}$$



Obrázek 2. Architektura systému Fitcrack

Distribuce výpočtu v systému Fitcrack spočívá v generování menších úloh, kde každá úloha je definována rozsahem indexů i_{min} a i_{max}, přičemž platí:

$$0 \leq i_{min} \leq i_{max} \leq (|P| - 1). \tag{2}$$

Řešením každé dílčí úlohy je pak vyzkoušení všech hesel daných generátorem $g(i)$, kde $i \in \langle i_{min}, i_{max} \rangle$. Úloha pak může skončit dvěma způsoby:

- **Jedno z hesel je správné** - Uzel v rámci úlohy nalezl správné heslo, další hesla již nezkouší a o správném heslu informuje server.
- **Žádné z hesel není správné** - Uzel vyzkoušel všechna hesla v přiděleném rozsahu indexů a o této skutečnosti informuje server.

Celý výpočet pak končí:

- **Úspěchem**, bylo-li v rámci některé z dílčích úloh nalezeno správné heslo.
- **Neúspěchem**, pokud byly dokončeny všechny dílčí úlohy a správné heslo nebylo nalezeno.

3.4 Adaptivní plánování úloh

Jak bylo zmíněno v sekci 3.3, nezbytným předpokladem distribuovaného výpočtu je vytváření dílčích úloh na základě rozdělování indexů hesel. V systému Fitcrack

jsou úlohy přidělovány progresivním způsobem. V sekci 3.1 bylo vysvětleno, že klientský uzel nejprve požádá projektový server o přidělení úlohy. Jakmile získá úlohu, provede výpočet a výsledek zašle zpět na server. Pokud klient v rámci dané úlohy nalezl heslo, server požádá ostatní klienty o ukončení výpočtu a další úlohy již nepřiděluje. Pokud klient heslo nenalezl, bude mu přidělena další úloha.

V dynamickém heterogenním prostředí uvažujeme, že jednotlivé uzly mohou vykazovat různý výpočetní výkon v závislosti na jejich hardwarových prostředcích. Systém Fitcrack také podporuje dynamické připojení, či odpojení klientů za běhu výpočtu. Kromě toho také uvažujeme, že výpočetní výkon každého uzlu se může v čase měnit, např. při zatížení uzlu jiným procesem.

Jádrem plánovacího subsystému je blok *Generator*, který je detailně popsán v příloze A. Generátor implementuje plánovací strategii, jejímž cílem je co nejefektivněji využít všechny dostupné výpočetní kapacity (tj. klientské uzly). Myšlenkou plánovací strategie je, aby na každé úloze uzly pracovaly zhruba stejnou dobu. Je zřejmé, že výkonnějším uzlům může server přidělovat složitější úlohy, tj. úlohy s větším rozsahem indexů, zatímco méně výkonným uzlům úlohy jednodušší.

Při adaptivním plánování v systému Fitcrack je nejprve třeba určit, jak dlouho by trvalo ověřit všechna zbývající hesla na všech aktivních výpočetních uzlech. Na základě tohoto času a výkonu jednotlivých uzlů následně můžeme každému uzlu přidelit rozsah indexů popisující podmnožinu stavového prostoru P. Konkrétní velikost úlohy je pak vždy určena výkonem daného uzlu. Nechť t_p je doba výpočtu (v sekundách) popisující, jak dlouho bude trvat ověření zbývajících hesel, s_i je počet hesel (množství indexů hesel) přidělených uzlu i a v_i je aktuální výpočetní výkon (rychlost) uzlu i v heslech za sekundu.

Na základě rychlosti uzlu v_i bude uzlu i přidělena část stavového prostoru P o velikosti $s_i = t_p \cdot v_i$. Rychlost v_i je určena na základě výpočtu předchozí úlohy jako $v_i = \frac{s_{prev}}{t_{prev}}$, kde s_{prev} je velikost (počet indexů) předchozí úlohy přidělené uzlu i a t_{prev} je skutečný čas výpočtu této úlohy (v sekundách). Pokud uzel i dosud nedokončil žádnou úlohu, je v_i určena pomocí speciální úlohy typu *benchmark*, jejímž cílem je určit výkon uzlu.

Zbývající čas t_p není možno určit pouze na základě výkonu jednotlivých uzlů. Příliš nízké, či příliš vysoké hodnoty mohou mít negativní vliv na efektivitu výpočtu. Platí, že:

- Nižší hodnota t_p znamená přidělování více menších úloh a je tak vhodnější pro méně stabilní prostředí, kdy očekáváme, že klienti se budou častěji připojovat/odpojovat, či očekáváme možnost selhání výpočtu některé z úloh. Nižší t_p ovšem přináší také celkově vyšší režii pro komunikaci mezi serverem a klienty.
- Vyšší hodnota t_p představuje menší režii a větší procento času strávené samotným výpočtem. V heterogenním prostředí však vyšší t_p může vést k méně efektivní distribuci výpočtu, zejména na konci stavového prostoru. Může tedy nastat stav, kdy např. z 20-ti uzlů bude poslední hodinu počítat pouze 10, protože již nezbývá žádný stavový prostor pro tvorbu úloh, které by bylo možné přidělit zbývajícím uzlům.

Pro dosažení efektivní distribuce úloh definujme funkci $proctime(t_J, s_R, k)$, která adaptivně vypočte očekávaný čas výpočtu t_p do konce stavového prostoru hesel. Parametr t_J představuje dosavadní dobu výpočtu, s_R je počet zbývajících indexů (hesel) a k je počet aktivních uzlů, které se mohou podílet na výpočtu. V souvislosti s výše uvedeným je nutné poznamenat, že parametry t_J, s_R and k se mohou v čase měnit. Funkce *proctime* je implementována dle algoritmu 1. Na základě odhadovaného zbývajícího času t_p, dostane každý uzel přidělenou část stavového prostoru o velikosti $s_i = v_i \cdot t_p$. Zbývající část stavového prostoru je tedy distribuována na základě aktuálního výkonu jednotlivých uzlů. V optimálním případě dokončí všechny uzly svůj výpočet za odhadnutou dobu t_p.

Algoritmus 1: Adaptivní výpočet t_p

Input: t_J, s_R, k
Output: t_p

```
1:  v_sum = 0
2:  forall client_i ∈ {0, ..., k} do
3:      if client_i is active then
4:          v_i = s_prev / t_prev
5:          v_sum = v_sum + v_i
6:  t_p = (s_R / v_sum) · α
7:  if t_p < t_pmin then
8:      t_p = t_pmin ;                          // minimální čas
9:  else if t_J > t_pmax then
10:     t_p = min(t_p, t_pmax) ;                // maximální čas
11: else
12:     t_p = min(t_p, t_J) ;                   // menší úlohy
13: return t_p
```

Řádky 2 až 5 algoritmu 1 slouží k výpočtu celkové rychlosti všech aktivních uzlů. Dále bychom uvažovali, že dobu t_p vypočteme jako $t_p = \frac{s_R}{v_{sum}}$. Na řádku 6 však dochází ještě k násobení parametrem α, kterému říkáme *koeficient distribuce* a definujeme jej jako reálné číslo v rozsahu 0 až 1. Právě násobením výsledného času tímto koeficientem určujeme, že mezi aktivní uzly *nebudeme* přidělovat celý zbytek stavového prostoru P, ale pouze jeho *část*.

Sémantika koeficientu α je zřejmá. Pokud např. $\alpha = 0,1$, pak bude mezi aktivní uzly rozděleno právě 10% stavového prostoru P. Důvod, proč nepřidělujeme celý stavový prostor, vychází z dynamického chování, které u uzlů očekáváme. Tímto chováním uvažujeme změnu výkonu uzlů v čase, připojování nových uzlů, či odpojování existujících, apod. Pokud by v průběhu výpočtu došlo k připojení nových uzlů, bez vynásobení času koeficientem α by nezbýval žádný stavový prostor pro úlohy, které bychom těmto uzlům mohli přidelit, což by vedlo k vý-

raznému poklesu efektivity výpočtu. Díky řádku 6 však po většinu času existuje prostor pro tvorbu nových úloh.

Na řádcích 7 až 12 algoritmu 1 můžeme vidět, že výsledný čas t_p je také omezen parametry t_{pmin} a t_{pmax}. Parametr t_{pmin} udává minimální čas výpočtu dílčí úlohy, pro který (s výjimkou posledních úloh) výpočet považujeme za efektivní. Podobně t_{pmax} udává maximální čas provádění jedné dílčí úlohy, kterého chceme dosáhnout. Hodnotu koeficientu α a parametrů t_{pmin}, t_{pmax} určuje administrátor systému. Na základě dosavadních experimentů doporučujeme pro středně velké sítě (v řádu desítek uzlů) nastavit t_{pmin} alespoň na 1 minutu a t_{pmax} zhruba na 1 hodinu.

4 Závěr

V této zprávě byly diskutovány odlišné technologie pro implementaci distribuovaných výpočtů při obnově hesel na více uzlech. Ačkoli MPI vykazuje ze všech nejmenší režii a nejvyšší výkon, nepodporuje nativně dynamické přidávání nových uzlů za běhu výpočtu, zotavení z chyb, ani prostředky pro běh v nedůvěryhodném prostředí jako je síť Internet. Apache Hadoop tyto nedostatky kompenzuje, přičemž tato kompenzace je však vykoupena nižším výkonem. Hadoop je nicméně omezen na výpočet úloh, které lze definovatm modelem MapReduce. Framework BOINC poskytuje mechanismy pro běh v nedůvěryhodném prostředí a poskytuje jak prostředky pro dynamickou změnu počtu výpočetních uzlů, tak pro zotavení z chyb. Ačkoli primární doménou využití je tzv. volunteer computing, lze jej použít i pro privátní projekty a přenos citlivých dat. Také s přihlédnutím na skutečnost, že zatím neexistuje jeho evidované použití pro účely obnovy hesel, rozhodli jsme se pro tvorbu našeho řešení použít právě BOINC.

Spojením nástroje hashcat, technologie BOINC a několika dalších přidaných funkčních bloků jsme vytvořili koncept systému Fitcrack. Tento systém slouží k distribuované obnově hesel jak v síti LAN, tak v Internetu. Fitcrack je schopen vyrovnat se s proměnlivým počtem uzlů, výpadkem uzlu, i chybě při výpočtu. V této zprávě jsme představili jeho architekturu, princip výpočtu a techniku distribuce výpočtu na základě rozdělení indexů hesel. V neposlední řadě byl představen algoritmus pro adaptivní plánování úloh, který řeší nerovnoměrný výkon jednotlivých uzlů a jeho kolísání za běhu výpočtu. Detailní popis jednotlivých funkčních bloků a struktura databáze serveru se nachází v příloze.

Tato technická zpráva může sloužit jak uživatelům, tak vývojářům systému Fitcrack pro jeho další rozvoj a zdokonalování. Může také poskytnout inspiraci při hledání dalších způsobů realizace obnovy hesel v distribuovaném prostředí.

Odkazy

[1] D. P. Anderson. „BOINC: a system for public-resource computing and storage“. In: *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. Lis. 2004, s. 4–10.

[2] D. Apostal, K. Foerster, A. Chatterjee a T. Desell. „Password recovery using MPI and CUDA“. In: *2012 19th International Conference on High Performance Computing*. Pros. 2012, s. 1–9. DOI: 10.1109/HiPC.2012.6507505.

[3] Z. Chen, F. Han, J. Cao, X. Jiang a S. Chen. „Cloud computing-based forensic analysis for collaborative network security management system“. In: *Tsinghua Science and Technology* 18.1 (ún. 2013), s. 40–50. DOI: 10.1109/TST.2013.6449406.

[4] C Cho, S Chin a Kwang Sik Chung. „Cyber forensic for hadoop based cloud system“. In: *International Journal of Security and its Applications* 6.3 (2012), s. 83–90.

[5] Mehnaz Hafeez, Sajjad Asghar, Usman Malik, Ahmadand ur Rehman a Naveed Riaz. „Survey of MPI Implementations“. In: *Digital Information and Communication Technology and Its Applications*. Sv. 167. Berlin, Germany: Springer Berlin Heidelberg, 2011, s. 206–220. ISBN: 978-3-642-22027-2.

[6] Radek Hranický. *Digitální forenzní analýza s použitím distribuovaného prostředí*. Teze disertační práce. Fakulta informačních tehnologií VUT v Brně. 2016.

[7] Radek Hranický, Martin Holkovič, Petr Matoušek a Ondřej Ryšavý. „On Efficiency of Distributed Password Recovery“. In: *The Journal of Digital Forensics, Security and Law* 11.2 (2016), s. 79–96. ISSN: 1558-7215. URL: http://www.fit.vutbr.cz/research/view_pub.php.cs?id=11276.

[8] Radek Hranický, Martin Holkovič, Petr Matoušek a Ondřej Ryšavý. „On Efficiency of Distributed Password Recovery“. Angl. In: *The Journal of Digital Forensics, Security and Law* 11.2 (2016), s. 79–96. ISSN: 1558-7215. URL: http://www.fit.vutbr.cz/research/view_pub.php?id=11276.

[9] Radek Hranický, Petr Matoušek, Ondřej Ryšavý a Vladimír Veselý. „Experimental Evaluation of Password Recovery in Encrypted Documents“. In: *Proceedings of ICISSP 2016*. Roma, IT: SciTePress - Science a Technology Publications, 2016, s. 299–306. ISBN: 978-989-758-167-0. URL: http://www.fit.vutbr.cz/research/view_pub.php.cs?id=11052.

[10] Radek Hranický, Petr Matoušek, Ondřej Ryšavý a Vladimír Veselý. „Experimental Evaluation of Password Recovery in Encrypted Documents“. Angl. In: *Proceedings of ICISSP 2016*. Roma, IT: SciTePress - Science a Technology Publications, 2016, s. 299–306. ISBN: 978-989-758-167-0. URL: http://www.fit.vutbr.cz/research/view_pub.php?id=11052.

[11] Sol Ji Kang, Sang Yeon Lee a Keon Myung Lee. „Performance Comparison of OpenMP, MPI, and MapReduce in Practical Problems“. In: *Advances in Multimedia* (srp. 2015), s. 9.

[12] Alexander Kasabov a Jochem van Kerkwijk. *Distributed GPU password cracking*. Tech. zpr. "Research Project 1, Final version rev. 2". Květ. 2010.

[13] Keonwoo Kim. „Distributed password cracking on GPU nodes“. In: *2012 7th International Conference on Computing and Convergence Technology (ICCCT)*. Pros. 2012, s. 647–650.

[14] Ondřej Kos. „Obnova hesel v distribuovaném prostředí“. czech. Dipl. Brno, CZ: Vysoké učení technické v Brně, Fakulta informačních technologií, 2015. URL: http://www.fit.vutbr.cz/study/DP/DP.php?id=18213.

[15] Jorge L. Reyes-Ortiz, Luca Oneto a Davide Anguita. „Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf“. In: *Procedia Computer Science* 53 (2015), s. 121–130. ISSN: 1877-0509. DOI: http://dx.doi.org/10.1016/j.procs.2015.07.286. URL: http://www.sciencedirect.com/science/article/pii/S1877050915017895.

[16] Vassil Roussev, Liqiang Wang, Golden Richard a Lodovico Marziale. „A Cloud Computing Platform for Large-Scale Forensic Computing“. In: *Advances in Digital Forensics V: Fifth IFIP WG 11.9 International Conference on Digital Forensics, Orlando, Florida, USA, January 26-28, 2009, Revised Selected Papers*. Ed. Gilbert Peterson a Sujeet Shenoi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, s. 201–214. ISBN: 978-3-642-04155-6. DOI: 10.1007/978-3-642-04155-6_15. URL: http://dx.doi.org/10.1007/978-3-642-04155-6_15.

[17] V. L. Thing a H.-M. Ying. „Making a faster cryptanalytic time-memory trade-off“. In: *Advances in Cryptology* (2003), s. 617–630.

[18] J. Zou, D. D. Lin a G. C. Mi. „A universal distributed model for password cracking“. In: *2011 International Conference on Machine Learning and Cybernetics*. Sv. 3. Čvc 2011, s. 955–960. DOI: 10.1109/ICMLC.2011.6016851.

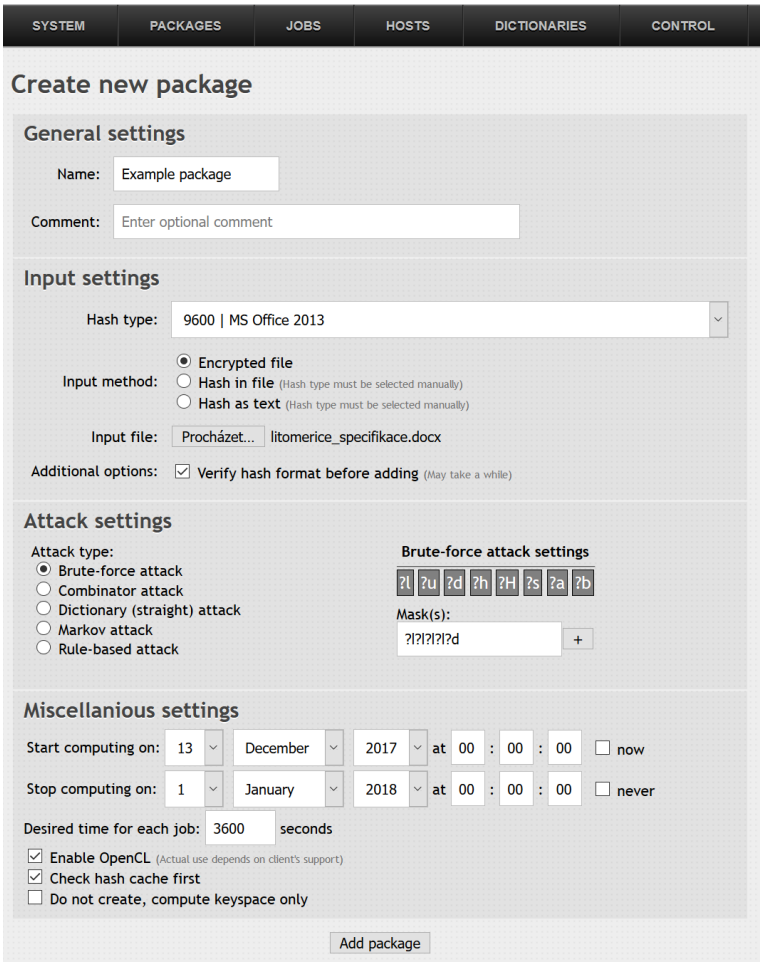
A Popis jednotlivých subsystémů

Tato sekce obsahuje popis jednotlivých funkčních bloků dle schématu na obrázku 2. V sekci se vyskytují identifikátory jako názvy tabulek a sloupců v databázi serveru, tyto jsou popsány v příloze B.

A.1 WebAdmin

Fitcrack WebAdmin je webová aplikace v jazyce PHP, která slouží pro správu distribuované sítě a v ní běžících výpočtů. Nástroj obsahuje prostředky pro uživatelsky přívětivou tvorbu pracovních balíčků (viz obrázek 3), sledování připojených klientů a jejich mapování na balíčky, spouštění, zastavování a restart pracovních balíčků. Pro každý pracovní balíček umožňuje sledovat průběh výpočtu a zobrazit seznam provedených, naplánovaných i běžících úloh. Dále umožňuje spravovat slovníky hesel, ovládat jednotlivé funkční bloky serveru, procházet logy a souhrnné statistiky.

Rozhraní WebAdmin běží nad serverem Apache a jeho dílčí skriptu jsou prováděny na základě HTTP požadavků administrátora systému. Pomocí databáze MySQL komunikuje s funkčními bloky Generator, Assimilator a vestavěnými bloky serverové části systému BOINC. Kromě této komunikace také přes Unixový shell v případě potřeby spouští procesy aplikace XtoHashcat a aplikací ze sady hashcat-utils.



Obrázek 3. Tvorba pracovních balíčku v nástroji WebAdmin

A.2 XtoHashcat

Popis U množství formátů šifrovaných médií, jako jsou např. dokumenty Office, archivy ZIP, RAR a další, je možné automaticky detekovat konkrétní formát, verzi zabezpečení a na základě těchto informací provést extrakci metadat potřebných k útoku na zabezpečení tohoto média. Tato metadata obsahují typicky informace o formátu a způsobu zabezpečení, kryptografické heše, kryptografickou sůl a případně i část zašifrovaných dat, je-li k úspěšnému útoku nutno část dat také dešifrovat. Metadata jsou extrahována do formátu používaného nástrojem hashcat. Příklady hešů pro konkrétní formáty jsou k dispozici[21] na webu hashcatu. K extrakci těchto metadat slouží nástroj XtoHashcat. Jedná se

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

o skript v jazyce Python3, který dokáže vzít na vstupu libovolný soubor spadající do množiny podporovaných formátů a na výstupu dodá metadata sloužící jako vstup do nástroje hashcat. Kromě toho na další řádek vytiskne číslo detekovaného formátu, které odpovídá hodnotě spouštěcího argumentu `-hash-type` pro nástroj hashcat. XtoHashcat využívá volně dostupných skriptů a binárních souborů, které dokáží extrahovat potřebná metadata ze souborů podporovaných formátů. Tento nástroj se nachází v repozitáři "xtohashcat"v domovském adresáři uživatele "boincadm". Zde je možné nalézt jak samotný skript, tak krátké README a jednotlivé extrahující skripty.

Rozhraní XtoHashcat je spouštěn jako skript přes unixový shell nástrojem WebAdmin. XtoHashcat je spouštěn typicky při vytváření nového pracovního balíčku, pokud administrátor jako formát heše zvolí volbu „autodetect“.

```
XtoHashcat.py <PathToFile> [-f <Mode>]
```

Činnnost Činost nástroje XtoHashcat ilustruje následující algoritmus:

Algoritmus 2: Princip činnosti bloku XtoHashcat

```
1 if Zadán argument -f then
2 |  Předpokládej formát zadaný parametrem -f.
3 else
4 |  Zjisti formát na základě signatury a přípony.
5 if Nebyl nastaven validní formát then
6 |  Ukonči program.
7 Zavolej odpovídající skript pro extrakci metadat.
8 Vytiskni získaný heš.
9 Vytiskni číslo detekovaného hashcat formátu.
```

A.3 hashcat-utils

Popis Balíček nástrojů hashcat-utils obsahuje sadu nástrojů dodávaných spolu s nástrojem hashcat. Jedná se o nástroje pro zpracování masek hesel, výpočet složitosti úloh, generování pravděpodobnostních matic pro útok pomocí Markovských řetězců, apod. Tyto nástroje spouští v případě potřeby přes Unixový shell nástroj WebAdmin, typicky při vytváření nového pracovního balíčku.

Pozn.: Kromě hashcat-utils je v současném prototypu na straně serveru spouštěn i samotný hashcat z důvodu ověřování formátů vstupních hešů. Toto ovšem vyžaduje zbytečné zavádění kódu OpenCL kernelů a nutnost podpory OpenCL na CPU severu. Alternativní řešení je popsáno v sekci "hashcat".

Rozhraní Nástroje ze sady hashcat-utils jsou spouštěny jako samostatné aplikace přes unixový shell pomocí nástroje WebAdmin.

A.4 Generator

Popis Blok Generator běží na serveru jaké démon, jehož účelem je generování dílčích úloh k pracovním balíčkům. Toto zahrnuje i úlohy typu benchmark, které slouží ke změření výkonu konkrétních uzlů při daném formátu a typu útoku. Pro každý uzel, u kterého je výledek benchmarku pro dané zadání již znám, jsou generovány dvě úlohy: jedna, která je uzlu následně ihned přiřazena a druhá „do zásoby“ pro příští iteraci výpočtu daného uzlu. Generátor také implementuje adaptivní plánovací algoritmus popsaný v sekci 3, který rozhoduje, která úloha bude přidělena kterému uzlu a vytváří úlohy s ohledem na aktuální výkon uzlu, náročnost úlohy, konfiguraci serveru a nastavení pracovního balíčku. Tento blok také řeší chyby při výpočtu a výpadky v síti, např. v případě neúspěšně zpracované úlohy přiřadí tuto jinému uzlu. Velikost úlohy je definována pomocí rozsahu indexů hesel, které má uzel při práci zkoušet. Zatímco ovšem benchmark nástroje hashcat vrací reálný počet zpracovaných hesel za sekundu, u reálných úloh je při některých typech útoku (mask attack, rule-based attack, ...) nutno počítat s normalizovanými "hashcat-indexy", kde pro jeden index může být uvažováno i více hesel. Důvodem je jednak snaha předejít přetékání 64-bitových čísel, jednak skutečnost, že implementace pravidel pro modifikaci hesel (u rule-based attack) je řešena na klientské části v samotném nástroji hashcat. Blok Generator tedy také implementuje převod mezi reálnými indexy a "hashcat-indexy", aby mohl být administrátor systému o stavu výpočtu korektně informován.

Každá vygenerovaná úloha je vždy zaregistrována v BOINC tabulce workunit (viz sekci B), kde se jí přiřadí ID, kterým je po úspěšném výpočtu svázána s výsledkem, tedy položkou z BOINC tabulky result. Současně jsou také vytvořena odpovídající vstupní data pro klienta, či klienty (v případě replikace úlohy).

Rozhraní Generator běží na serverové části jako démon v nekonečné smyčce a jeho běh je nezbytný pro provádění výpočtu. Se zbytkem systému komunikuje skrze databázi MySQL a souborový systém na serveru.

Činnost V systému Fitcrack blok Generator vytváří 2 typy úloh: úlohy typu *benchmark* a *standardní* úlohy. Úlohy typu benchmark jsou generovány pro nově připojené uzly, které se dosud nepodílely na zpracování žádné úlohy, případně pro uzly, na kterých nastala chyba. Jejich cílem je změření výkonu dané stanice, aby bylo možno uzlu přidělit odpovídající množství práce. Standardní úlohy obsahují informace o zvoleném typu útoku: slovník u slovníkového útoku, masky pro útok hrubou silou, apod.

V systému Fitcrack blok Generator udržuje v databázi vždy dvě naplánované úlohy (proběhl-li benchmark): jednu, na které klient aktuálně pracuje a druhou, která je připravena k odeslání klientovi ihned po obdržení výsledku první úlohy.

Tímto způsobem eliminujeme prodlevy mezi generováním úloh. Detailněji činnost tohoto modulu přibližuje následující pseudokód:

Algoritmus 3: Princip činnosti bloku Generator

```
1  while (1) do
2      // Inicializace
3      Smaž uzly, které se podílely na řešení úlohy, jejíž pracovní balíček
         byl již dokončen, buď úspěšně (finished) nebo neúspěšně
         (exhausted). Pokud některý z balíčků přesáhl stanovenou dobu
         ukončení, nastav jeho stav na Finishing (12).
4      foreach Běžící pracovní balíček (stav ≥ 10) do
5          if Není nastaven čas zahájení then
6              Nastav čas zahájení na aktuální čas.
7          if K balíčku se vážou masky hesel then
8              Ulož je do příslušného pole, které odpovídá danému balíčku.
9          Najdi uzly, které se mají podílet na výpočtu (a dosud se
             nepodílely) a umísti je do databáze.
10         // Benchmark
11         foreach Přidělený aktivní uzel, který má stav Benchmark (0) do
12             if Uzel ještě nemá naplánován benchmark then
13                 Naplánuj pro tento uzel benchmark.
14         // Výpočet
15         foreach Aktivní uzel ve stavu Normal (1) do
16             if Počet naplánovaných úloh pro uzel ≥ 2 then
17                 Pokračuj na další uzel.
18             if Stav uzlu je Running (10) then
19                 Vygeneruj novou úlohu podle typu balíčku, případně
                     znovu přiděl nedokončené úlohy.
20             if Stav uzlu je Finishing (12) then
21                 Znovu přiděl nedokončené úlohy Pokud taková neexistuje,
                     nastav stav uzlu na Done (3).
22         // Kontrola stavu
23         if Stav balíčku je Finishing (12) a neobsahuje žádné úlohy then
24             if Aktuální čas > plánovaný čas ukončení then
25                 Nastav stav balíčku na Timeout (4).
26             else
27                 if Aktuální index ≥ maximální index then
28                     Nastav stav balíčku na Exhausted (2) - stavový
                         prostor vyčerpán.
29                 else
30                     Nastav stav balíčku na Ready (0) - výpočet byl
                         pozastaven
31     Čekej stanovený časový interval.
```

A.5 Validator

Popis Příchozí výsledky úloh od klientů jsou nejprve zpracovány blokem Validator. Jeho cílem je ověřit korektnost syntaxe příchozí zprávy a také, v případě přiřazení jedné úlohy více klientům (tzv. replikace), ověřit shodu obou získaných výsledků. Pro činnost výpočtu nejsou veškeré operace tohoto bloku bezpodmínečně nutné, záleží na konkrétním nasazení a úvaze administrátora, zda uvažuje možnost kompromitace uzlů a pozměnění zpráv od klienta, potažmo replikaci úloh.

Rozhraní Podobně jako Generator, také Validator běží jako démon v nekonečné smyčce a se zbytkem systému komunikuje skrze databázi MySQL a souborový systém.

A.6 Assimilator

Popis Výsledek každé úlohy je po zpracování funkčním blokem Validator předán bloku Assimilator. Jeho cílem je samotné zpracování výsledku a na základě toho rozhodnutí o dalším postupu. V případě nalezení hesla Assimilator aktualizuje databázi, zruší všechny aktuálně prováděné úlohy v pracovním balíčku a vymaže nepotřebné záznamy v databázi. V případě úspěšně dokončené úlohy typu benchmark v databázi aktualizuje informaci o výkonu daného uzlu na naměřenou hodnotu, apod.

Rozhraní Podobně jako Generator a Validator, také Assimilator běží jako démon v nekonečné smyčce a se zbytkem systému komunikuje skrze databázi MySQL a souborový systém serveru.

Činnost Assimilator periodicky kontroluje BOINC tabulku workunit. Pokud nalezne nový dosud nezpracovaný výsledek, dojde k jeho ověření. Podle obsahu výstupu jsou odpovídajícím způsobem aktualizovány údaje v databázi. V případě úspěšného dokončení pracovního balíčku (nalezení hesla) server přikáže zastavit výpočet všem klientům, kteří pracují na dosud běžících úlohách z tohoto balíčku. To je opět provedno modifikací údajů v databázi. Detailněji to ukazuje následující algoritmus:

Algoritmus 4: Princip činnosti bloku Assimilator

```
1  while (1) do
2      Zjisti od kterého uzlu přišel výsledek a ke kterému pracovnímu
         balíčku patří.
3      Přečti typ výsledku
4      switch typ do
5          case benchmark do
6              if Výsledek je v pořádku (kód 0) then
7                  Přečti změřený výkon a ulož jej do databáze.
8              else
9                  Naplánuj nový benchmark za delší čas.
10         case normal do
11             if Bylo nalezeno heslo (kód 0) then
12                 Přečti heslo a ulož jej do databáze (a případně cache).
13                 Uprav stav pracovního balíčku na Finished (1).
14                 Zašli všem zapojeným uzlům pokyn k ukončení výpočtu.
15                 Odstraň/archivuj dosud nedokončené úlohy.
16                 Přečti čas výpočtu úlohy a ulož jej.
17             else
18                 if Nebylo nalezeno heslo (kód 1) then
19                     Aktualizuj velikost úloh dle algoritmu 1.
20                     Aktualizuj počet již ověřených hesel.
21                 else
22                     // Chyba výpočtu
23                     Zruš úlohy pro daný uzel a nastav jim příznak retry.
24                     Vynuluj výkon uzlu v databázi a naplánuj nový
                         benchmark.
```

A.7 Transitioner

Popis Funkční blok Transitioner zpracovává a aktualizuje změny ve stavu jednotlivých úloh a jejich výsledků. Pro každou dokončenou úlohu ukládá do databáze přijaté výsledky, v případě chyby generuje příslušná hlášení do logu, apod.

Rozhraní Stejně jako předchozí tři bloky, běží také Transitioner jako démon v nekonečné smyčce a se zbytkem systému komunikuje skrze databázi MySQL a souborový systém. Narozdíl od bloků Generator a Scheduler však Transitioner běží asynchronně.

A.8 Scheduler

Popis Jak bylo uvedeno výše, komunikace v distribuované síti funguje na principu klient-server. Právě blok Scheduler implementuje vyřizování požadavků od klientských uzlů. Blok BOINC Client zašle serveru zprávu obsahující informace o klientském uzlu, seznam dokončených úloh a požadavek na další úlohu. Na straně serveru přijme tuto zprávu Scheduler, který v databázi aktualizuje dokončené úlohy a je-li pro klienta připravena další úloha, zašle mu ji formou odpovědi. Zaslání nové úlohy předpokládá vygenerovanou úlohu blokem Generator a alokovaný paměťový prostoru příslušné velikosti blokem Feeder.

Rozhraní Narozdíl od předchozích, není blok Scheduler implementován jako démon, ale jako FastCGI nad serverem Apache, protože komunikace klienta se serverem pomocí pomocí výše uvedného protokolu běží právě nad HTTP/HTTPS. Scheduler proto může běžet ve více instancích současně, v závislosti na intenzitě požadavků klientských uzlů.

A.9 Feeder

Popis Kromě zadání jednotlivých úloh jsou klientům zasílány také další soubory nutné pro provoz klientské části: binární soubory nástrojů Runner, hashcat, soubory s kódem v OpenCL C pro GPU, slovníky a další. Systém Fitcrack díky frameworku BOINC zajišťuje, aby na klientech byla vždy aktuální verze všech potřebných souborů. K tomuto účelu nástroj Feeder vytváří bloky sdílené paměti pro uložení všech výše uvedených dat, která budou zasílana klientům pomocí bloku Scheduler.

Rozhraní Blok Feeder běží jako démon v nekonečné smyčce. Se zbytkem systému komunikuje pomocí databáze MySQL a souborového systému.

A.10 File deleter

Popis Při komunikaci mezi klientem a server dochází k předávání řady souborů, zejména pak vstupních a výstupních souborů každé úlohy. Tyto soubory jsou ukládány ve specializovaných adresářích pro download a upload. Po dokončení každé úlohy a ověření jejích výsledků bloky Validator a Assimilator již tyto soubory nejsou potřeba. Pro jejich odstranění slouží blok File deleter.

Rozhraní Modul File deleter běží na serveru jako démon v nekonečné smyčce. Se zbytkem systému komunikuje pomocí databáze MySQL a souborového systému serveru.

A.11 BOINC Client

Popis Aplikace BOINC Client je jedinou součástí klientské části systému Fitcrack, kterou je nutné na cílovou stanici instalovat ručně. BOINC Client zajišťuje komunikaci se serverovou částí (konkrétně blokem Scheduler) formou zasílání požadavků na zadání nových úloh a hlášení výsledků úloh již provedených. BOINC Client po autentizaci vůči serveru zajistí uložení binárních a datových souborů zaslaných serverem. Pro každou úlohu pak vytvoří lokální kopii vstupních souborů. Dle specifikace úlohy BOINC Client provede spuštění příslušných binárních souborů, sleduje výpočet a po jeho dokončení vytvoří výstupní soubory a odešle je zpět na server. V konfiguračním souboru aplikace BOINC Client lze také nastavit, kdy se má výpočet na daném uzlu provádět, jaké výpočetní prostředky a v jakém rozsahu je možno využít, apod.

Rozhraní Na klientském uzlu může BOINC Client běžet jako démon, potažmo u Unixových systémů jako Cron job, případně může být spouštěn manuálně. Se serverovou částí systému komunikuje pomocí protokolu BOINC scheduling server protocol, který běží nad HTTP/HTTPS. Binární soubory dané úlohy spouští pomocí systémových volání. Je-li na klientské části aplikace BOINC Manager, komunikuje s ní BOINC Client pomocí technologie Graphical user interface Remote procedure calls (GUI RPC).

A.12 BOINC Manager

Popis Aplikace BOINC Manager je volitelnou součástí klientské strany systému Fitcrack. Jedná se o grafický nástroj pro konfiguraci aplikace BOINC Client a sledování výpočtu na daném uzlu. S blokem BOINC Client komunikuje pomocí GUI RPC.

Rozhraní BOINC Manager je spouštěn manuálně či automaticky jako samostatná aplikace, která nemusí běžet po celou dobu provádění výpočtu.

A.13 Runner

Popis Runner slouží jako middleware na klientské části systému. Po zahájení každé úlohy zpracuje vstupní parametry zaslané serverem a následně je předá bloku BOINC Client. Kromě těchto vstupů má také vlastní konfigurační soubor, který specifikuje, které GPU a s jakými parametry je možné k výpočtu použít, poněvadž BOINC Client tuto funkci v současné implementaci neposkytuje. Na základě těchto informací následně pomocí systémových volání spustí instanci nástroje hashcat s parametry specifickými pro řešenou úlohu. V průběhu výpočtu periodicky sleduje činnost aplikace hashcat a sleduje, zda nedošlo k chybě. Jakmile hashcat skončí daný běh, zpracuje zprávu pro server ve formě výstupních souborů, které následně zpracuje BOINC Client a zašle je severu.

Rozhraní Aplikace runner je spouštěna pomocí systémových volání blokem BOINC Client, se kterým komunikuje pomocí souborů, potažmo stdout/stderr. S nástrojem hashcat komunikuje Runner pomocí vstupních parametrů, zpracováváním výstupů stdout/stderr a návratovou hodnotou aplikace. Aplikaci Runner není nutno na klientskou stanici nijak instalovat, stažení aktuální verze a její konfiguraci zajišťují automaticky ostatní bloky systému Fitcrack.

A.14 hashcat

Popis Nástroj hashcat slouží v současné verzi systému Fitcrack jako výpočetní řešení pro lámání kryptografických hešů. Pro běh aplikace hashcat je na koncovém stroji nutná podpora technologie OpenCL pro příslušné procesory, s čímž souvisí nutnost instalace příslušných ovladačů a knihoven. Aplikaci hashcat není nutno na klientskou stanici nijak instalovat, stažení aktuální verze a její konfiguraci zajišťuje systém Fitcrack.

Rozhraní Aplikace hahcat je spouštěna pomocí systémových volání modulem Runner, se kterým komunikuje pomocí vstupních parametrů, standardního výstupu a návratové hodnoty.

B Struktura databáze MySQL

V databázi MySQL jsou soustředěna všechna potřebná data. Nalezneme zde množství BOINC tabulek, které jsou vygenerovány automaticky při tvorbě projektu. S těmito tabulkami pak pracuje vnitřně systém BOINC a administrátoři projektu je nemodifikují.

Dále jsou zde přítomny námi vytvořené tabulky. Ty obsahují užitečná data, která slouží pro generování úloh, ukládání výsledků výpočtu, nalezených hesel apod. V následujících podsekcích se podíváme na jejich strukturu. Jedná se o následující tabulky.

- fc_hashcache
- fc_host
- fc_host_activity
- fc_job
- fc_mask
- fc_package
- fc_settings

B.1 fc_hashcache

Do této tabulky jsou ukládána nalezená hesla v kombinaci s původním vstupem. To pak umožňuje uživateli prohledat všechna dříve nalezené hesla či heše před tím, než začne proces samotné obnovy. Schéma této tabulky je následující.

- **id** – Primární klíč tabulky
- **hash_type** – Typ heše v hashcat formátu `-attack-mode`
- **hash** – Původní vstup do nástroje hashcat
- **result** – Získané heslo či vstup heše
- **added** – Datum a čas přidání položky

B.2 fc_host

Tato tabulka obsahuje seznam aktivních hostů, kteří se aktuálně podílejí na výpočtu. Pomocí této tabulky jsou jednotlivý klienti svázáni s balíčky a je zde uveden také jejich současný výkon. Schéma této tabulky je následující.

- **id** – Primární klíč tabulky
- **boinc_host_id** – Reálné ID hosta uvedené v BOINC tabulce *host*
- **power** – Poslední naměřený výkon hosta vzhledem k danému formátu, v počtu reálných hesel za sekundu
- **package_id** – Identifikátor balíčku, ke kterému se tento záznam váže
- **status** – Stav tohoto hosta, může nabývat těchto hodnot:
  - **0** – Host provádí benchmarku nebo čeká na jeho přidělení
  - **1** – Host provádí klasický výpočet
  - **3** – Host dokončil všechnu přidělenou práci a již mu není generována další
  - **4** – Na hostu došlo k chybě
- **time** – Datum a čas přidání položky

B.3 fc_host_activity

Tato tabulka slouží k provázání hostů a balíčků. Určuje tedy, kteří hosté se mají účastnit kterého výpočtu. Schéma této tabulky je následující.

- **id** – Primární klíč tabulky
- **boinc_host_id** – Reálné ID hosta uvedené v BOINC tabulce *host*
- **package_id** – Identifikátor balíčku

B.4 fc_job

Tato tabulka obsahuje záznamy o všech provedených úlohách. Nalezneme zde všechny potřebné informace o tom, kdo úlohu prováděl, k jakému balíčku se váže, jak je veliká, zda a kdy byla dokončena apod. Schéma této tabulky je následující.

- **id** – Primární klíč tabulky
- **package_id** – ID balíčku, do kterého úloha patří
- **workunit_id** – Reálné ID úlohy uvedené v BOINC tabulce *workunit*
- **host_id** – ID hosta z tabulky *fc_host*
- **boinc_host_id** – Reálné ID hosta uvedené v BOINC tabulce *host*

- **start_index** – Index stavového prostoru, kde úloha začíná
- **start_index_2** – Index pro první slovník v kombinačním útoku. Použitý jen pokud jsou klienti příliš slabí nebo slovníky příliš velké.
- **hc_keyspace** – Stavový prostor hesel úlohy v hashcat počítání. Pro slovníkový a kombinační útok se jedná o počet reálných hesel.
- **passwords_verified** – Průběh úlohy – pro maskový útok udává počet již ověřených hashcat indexů, pro slovníkový a kombinační útok se jedná o počet reálných hesel
- **mask_id** – Odkaz na masku do tabulky *fc_mask*, se kterou daná úloha počítá.
- **duplicated** – Příznak, zda byla úloha kvůli jejímu nedokončení rozdělena na více úloh
- **duplicate** – Pokud je duplicated 1, obsahuje ID původní úlohy
- **time** – Datum a čas přidání položky
- **cracking_time** – Čas běhu dané úlohy
- **retry** – Příznak, zda se jedná o znovu přiřazenou úlohu
- **finished** – Příznak, zda byla úloha dokončena

B.5 fc_mask

Tato tabulka obsahuje seznam mask, které byly kdy použity ve slovníkových útocích. Také zde nalezneme vazbu mezi každou maskou a balíčkem. Schéma této tabulky je následující.

- **id** – Primární klíč tabulky
- **package_id** – Identifikátor balíčku
- **mask** – Řetězec s hashcat maskou
- **current_index** – Aktuální index průchodu stavovým prostorem masky
- **keyspace** – Reálný počet hesel v masce
- **hc_keyspace** – Počet hashcat indexů v masce

B.6 fc_package

Tato tabulka obsahuje seznam balíčků, tedy souborů určených k obnově. Nalezneme zde všechny potřebné informace nutné pro provedení distribuovaného útoku. Schéma této tabulky je následující.

- **id** – Primární klíč tabulky
- **token** – Unikátní identifikátor sezení
- **attack_mode** – Typ útoku hashcatu je dán levou cifrou, typ podútoku druhou cifrou
  - **00** – Slovníkový útok
  - **01** – Slovníkový útok s pravidly
  - **10** – Kombinační útok
  - **11** – Hybridní útok typu 6
  - **12** – Hybridní útok typu 7

    - **30** – Maskový útok
    - **31** – Maskový útok s .hcstat souborem
- **hash_type** – Typ heše v hashcat formátu
- **hash** – Vstup do nástroje hashcat
- **status** – Stav balíčku
    - **0** – Neprobíhá výpočet
    - **1** – Obnova byla dokončeno a heslo nalezeno
    - **2** – Byl prohledán celý zadaný stavový prostor hesel, ale správné heslo zde nebylo
    - **3** – Package obsahuje chybný vstup
    - **4** – Package byl ukončen z d§vodu vypršení nastaveného času
    - **10** – Probíhá výpočet
    - **12** – Výpočet dobíhá, někteří hosti počítají ale již se negenerují nové úlohy.
- **result** – Správné heslo, pokud bylo nalezeno
- **keyspace** – Počet všech možných hesel (pro kombinační útok údává počet hesel ve druhém slovníku)
- **hc_keyspace** – Stavový prostor v hashcat indexech
- **passwords_verified** – Počet již ověřených hashcat indexů, pro slovníkový a kombinační útok počet ověřených reálných hesel
- **current_index** – Aktuální index stavového prostoru hashcatu, pro kombinační útok se jedná o index v 2. slovníku.
- **current_index_2** – Index pro pohyb v prvním slovníku v kombinačním útoku
- **time** – Datum a čas přidání položky
- **name** – Název balíčku
- **comment** – Komentář k balíčku
- **time_start** – Čas zahájení obnovy
- **time_end** – Čas ukončení obnovy
- **cracking_time** – Celkový čas, který v součtu strávili hosté výpočtem
- **seconds_per_job** – Doba v sekundách, po kterou má běžet jedna úloha
- **config** – TLV formát pro předávání potřebných hodnot hostům
- **dict1** – Název souboru se slovníkem pro slovníkový a kombinační útok
- **dict2** – Název souboru s druhým slovníkem pro kombinační útok
- **rules** – Název souboru s pravidly
- **markov** – Název souboru s .hcstat souborem
- **replicate_factor** – Replikační faktor pro daný balíček

B.7 fc_settings

Tato tabulka slouží pro nastavení hodnot ovlivňujících výpočet napříč různými balíčky, případně některých standardních hodnot. Schéma této tabulky je následující.

- **id** – Primární klíč tabulky

- **delete_finished_jobs** – Příznak, zda se mají dokončené úlohy mazat z tabulky *fc_job*
- **default_seconds_per_job** – Výchozí doba trvání jedné úlohy
- **default_replicate_factor** – Výchozí replikační faktor úloh
- **default_verify_hash_format** – Příznak, zda se má při vytváření balíčku ověřit vstupní heš
- **default_check_hashcache** – Příznak, zda se má při vytváření balíčku ověřit hashcache, viz výše
- **default_job_timeout_factor** – Výchozí doba pro prohlášení úlohy za nedokončenou