Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky
Univerzita Komenského, Bratislava

Kryptoanalýza hašovacích funkcií pomocou SAT solverov

(Diplomová práca)

Tomáš Žubrietovský

Vedúci: doc. RNDr. Martin Stanek PhD.

Bratislava, 2010

UNIVERZITA KOMENSKÉHO, BRATISLAVA
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

Kryptoanalýza hašovacích funkcií pomocou SAT solverov

DILOMOVÁ PRÁCA

Študijný program: Informatika
Študijný odbor: 9.2.1. Informatika
Školiace pracovisko: Katedra informatiky
Školiteľ: doc. RNDr. Martin Stanek PhD.

BRATISLAVA, 2010

Tomáš Žubrietovský

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím citovaných zdrojov.

. . . .

Poďakovanie

Chcem sa poďakovať môjmu školiteľovi doc. RNDr. Martinovi Stanekovi PhD. za výber témy, cenné rady a trpezlivosť, spolubývajúcim za mnohé užitočné rady a Mgr. Michalovi Foríškovi za poskytnutie templatov pre sadzbu v programe TEX.

Abstrakt

Autor:	Tomáš Žubrietovský
Názov práce:	Kryptoanalýza hašovacích funkcií pomocou SAT solverov
Škola:	Univerzita Komenského
Fakulta:	Fakulta matematiky, fyziky a informatiky
Katedra:	Katedra informatiky
Vedúci diplomovej práce:	doc. RNDr. Martin Stanek PhD.

V tejto diplomovej práci sa venujeme kryptoanalýze hašovacích funkcií pomocou SAT solverov. Použitie SAT solverov na kryptoanalýzu hašovacích funkcií nie je nová oblasť, ale stále nie úplne preskúmaná. Preto sme sa rozhodli použiť SAT solvery na jednu z nových hašovacích funkcií a zistiť ako sú nové hašovacie funkcie odolné proti tomuto útoku. Vybrali sme hašovaciu fuknciu BLAKE, ktorá sa dostala do druhého kola o návrh na novú hašovaciu funkciu SHA-3. Na tejto hašovacej funkcii sme testovali jednosmernosť a hľadanie kolízií. Výsledky sme porovnali s výsledkami dosiahnutými pre staršiu hašovaciu funkciu MD5.

Kľúčové slová: SAT solver, BLAKE, MD5, kryptoanalýza

Predhovor

Táto diplomová práca je z oblasti kryptológie. Venujeme sa v nej kryptoanalýze hašovacích funkcií pomocou SAT solverov. Na preklad hašovacej funkcie do výrokovej formuly sme použili dve metódy, jednu od autorov Dejana Jovanovića a Predraga Janičića a druhú od autora Milana Šešuma. Týmito metódami sme vytvorili výrokové formuly pre jednosmernosť a hľadanie kolízií pre novú hašovaciu funkciu BLAKE a staršiu hašovaciu funkciu MD5. Výsledky sme porovnali s v závere zhodnotili úspešnosť tohto útoku.

Obsah

1 SAT solver **11**

1.1 Úvod do výrokovej logiky 11

1.2 SAT problém 13

1.3 SAT algoritmy 13

1.3.1 Davisov-Putnamov algoritmus 14

1.3.2 Davisov-Logemannov-Lovelandov (DLL) algoritmus 15

1.4 DLL algoritmus 16

1.4.1 Heuristika 18

1.4.2 Odvodzovacie mechanizmy, prehľadávanie 20

1.4.3 Analýza konfliktov 21

1.4.4 Predspracovanie, náhodný reštart 27

1.5 Boolean Constraint Propagation 28

1.5.1 BCP s počítadlami 28

1.5.2 BCP so smerníkmi 29

2 Preklad hašovacej funkcie do výrokovej formuly **31**

2.1 Hašovacia funkcia 31

2.2 Preklad do SAT jazyka 33

2.2.1 Popis programu 33

2.2.2 Preklad formuly do CNF 35

3 Testovanie **37**
3.1 Spôsob testovania 37
3.2 Hašovacia funkcia MD5 39
3.2.1 Popis hašovacej funkcie 40
3.2.2 Testovanie hašovacej funkcie MD5 41
3.3 Hašovacia funkcia BLAKE 45
3.3.1 Popis hašovacej funkcie BLAKE-32 45
3.4 Testovanie hašovacej funkcie BLAKE-32 48
3.5 Zhrnutie 53

4 Záver **55**

Úvod

Snaha zistiť, či daná formula je splniteľná viedlo počas viacerých rokov k vzniku stále lepších algoritmov na riešenie tohto problému. Implementovaním týchto algoritmov začali vznikať programy, kroré sa nazývajú SAT solvery. Tým, že veľa problémov riešených v matematike a informatike sa dá zakódovať ako booleovský výraz, použitie SAT solverov získalo v informatike široké uplatnenie.

Jedným z možných použití SAT solverov je hľadanie vzorov a kolízií v hašovacích funkciách.

V roku 2007 Národný inštitút pre štandardy a technológiu (NIST) vyhlásil súťaž o návrh pre novú hašovaciu funkciu pod názvom SHA-3. Do súťaže sa zapojilo 64 návrhov, z ktorých 51 sa dostalo do prvého kola a z nich 14 do druhého. Jednou z hašovacích funkcií, ktoré sa prepracovali do druhého kola je aj hašovacia funkcia BLAKE. Túto hašovaciu funkciu sme sa rozhodli podrobiť útoku pomocou SAT solverov a zistiť tak, ako sú nové hašovacie funkcie odolné voči tomuto útoku. Na porovnanie sme si zvolili aj staršiu hašovaciu funkciu MD5 a tiež odtestovali odolnosť tejto funkcie voči útoku pomocou SAT solverov.

Diplomová práca je rozdelená do troch kapitol.

Prvá kapitola je venovaná princípu fungovania SAT solverov. Na začiatku sme zadefinovali niektoré základné pojmy z výrokovej logiky potrebné pre ďalší text. Ďalšie časti prvej kapitoly sú už venované priamo SAT solverom, ich histórii, rôznym typom algoritmov používaných v SAT solveroch a podrobnejšiemu opisu najviac používaného algoritmu.

V druhej kapitole sú opísané spôsoby prekladu hašovacej funkcie do výrokovej formuly.

Dôraz je kladený na dva spôsoby, jeden od autorov Dejana Jovanovića a Predraga Janičića a druhý od autora Milana Šešuma, ktoré sú však založené na rovnakom princípe.

Predposledná kapitola je venovaná vlastnej práci. V prvej časti sme sa venovali testovaniu odolnosti staršej hašovacej funkcie MD5 voči útoku pomocou SAT solverov a navrhli sme úpravu prekladu hašovacej funkcie do výrokovej formuly a porovnali ju s pôvodným spôsobom prekladu. V druhej časti sme pomocou SAT solverov hľadali vzory a kolízie pre hašovaciu funkciu BLAKE-32.

Poslednou kapitolou je záver diplomovej práce. V ňom sme zhodnotili výsledky testov a načrtli možné smery ďalšej práce v tejto oblasti.

Kapitola 1

SAT solver

1.1 Úvod do výrokovej logiky

Táto časť obsahuje základné pojmy a definície z oblasti výrokovej logiky, ktoré sa budú využívať v ďalšom texte.

Prvky neprázdnej množiny P budeme nazývať prvotné formuly. Môžu to byť vety prirodzeného jazyka, slová nejakého formálneho jazyka alebo písmená $(p, q, r, ..., p_1, p_2, ...)$. Jazyk, ktorý okrem prvkov množiny P obsahuje aj symboly pre logické spojky $(\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow)$, pomocné symboly (rôzne typy zátvoriek), budeme označovať L(P) resp. L_P.

Výrokové formuly jazyka L_P (niekedy nazývané aj booleovské formuly) definujeme pomocou nasledujúcich syntaktických pravidiel:

1. každá prvotná formula $p \in P$ je výroková formula.
2. ak sú výrazy A, B výrokové formuly, potom výrazy $\neg A, (A \wedge B), (A \vee B), (A \Rightarrow B), (A \Leftrightarrow B)$ sú výrokové formuly.
3. každá výroková formula vznikne konečným použitím pravidiel (1) a (2).

Výroková formula sa niekedy nazýva aj booleovská formula.

Literálmi budeme nazývať prvotné formuly a ich negácie. Pre potreby SAT solverov je dobré zaviesť pojem konjunktívnej normálnej formy (CNF). Formula je v CNF, ak je

zapísaná ako konjunkcia klauzúl, pričom pod klauzolou rozumieme disjunkciu literálov. Dá sa dokázať, že každá formula výrokovej logiky je ekvivalentná nejakej formule v CNF.

Klauzulu s jedným literálom budeme nazývať jednoliterálnou a budeme ju označovať $[l]$, kde l je literál.

Negáciu literálu l budem označovať $\bar{l}$ a literál l sa bude nazývať čistý literál v množine klauzúl S, ak $\bar{l}$ nie je v žiadnej klauzule z S.

Pre každú prvotnú formulu zavedieme pojem pravdivostného ohodnotenia (valuácie) prvotnej formuly.

Definícia 1.1.1 (CNF) *Pravdivostné ohodnotenie (valuácia) prvotných formúl jazyka* L_p *je každé zobrazenie* $v : P \to \{0, 1\}$*, ktoré každej prvotnej formule* $p \in P$ *priradí hodnotu* 0 *(nepravda) alebo hodnotu* 1 *(pravda).*

Potom budeme hovoriť, že $\overline{v}(A)$ je pravdivostná hodnota formuly A pri ohodnotení v, pričom formula A je pravdivá pri ohodnotení v, ak $\overline{v}(A) = 1$, inak je formula A nepravdivá.

Ak pre formulu A bude existovať ohodnotenie v literálov formuly také, že platí $\overline{v}(A) = 1$, budeme hovoriť, že formula A je splniteľná.

Pre potreby algoritmov používaných v SAT solveroch definujeme metódu, ktorá sa nazýva metóda rezolventy a používa sa na elimináciu premenných vo výrokovej formule.

Definícia 1.1.2 (Metóda rezolventy) *Nech* C_1 *a* C_2 *sú ľubovoľné dve klauzuly. Ak existuje literál* L_1 *v* C_1*, ktorý je kontrárny literálu* L_2 *v* C_2*, tak vynecháme* L_1 *a* L_2 *z* C_1*, resp.* C_2 *a zostrojíme disjunkciu zostávajúcich klauzúl. Klauzulu, ktorá vznikne takýmto spôsobom nazývame rezolventa* C_1 *a* C_2*.*

Príklad 1.1.1: Uvažujme nasledujúce klauzuly

$$C_1 : p \vee q \vee r \qquad C_2 : \bar{p} \vee s$$

Keďže literál $\bar{p}$ je kontrárny k literálu p, rezolventa klauzúl C_1 a C_2 bude klauzula $q \vee r \vee s$.

1.2 SAT problém

Definícia 1.2.1 (SAT jazyk) *SAT jazyk je množina všetkých splniteľných booleovských formúl.*

Riešiť SAT problém znamená zistiť, či daná booleovská formula patrí do SAT jazyka, čiže, či je splniteľná. SAT je NP-úplný, čiže ak $P \subsetneq NP$, potom neexistuje deterministický polynomiálny algoritmus riešiaci SAT. Napriek tomu existujú množiny splniteľných booleovských formúl, pre ktoré existuje deterministický polynomiálny algoritmus. Napr. 2-SAT, množina booleovských formúl v CNF, kde každá klauzula obsahuje najviac dva literály, patrí do triedy P.

Na riešenie SAT problému sa používajú programy, ktoré sa vo všeobecnosti nazývajú SAT-solvery.

Riešenie booleovskej splniteľnosti a SAT-solvery majú dlhú históriu. Za začiatok sa považuje rok 1960, kedy bol publikovaný Davisov-Putnamov algoritmus (DP) [9]. Podľa [7] bol v princípe identický algoritmus vymyslený L. Lowenheimom už pol storočie predtým [16]. DP algoritmus je pre praktické použitie v SAT solveroch z dôvodov neefektívnosti a veľkých nárokov na pamäť nepoužiteľný, preto bol v roku 1962 vylepšený a publikovaný ako Davisov-Logemannov-Lovelandov algoritmus (DLL) [8]. Väčšina dnešných SAT-solverov je založená práve na tomto algoritme.

1.3 SAT algoritmy

Okrem DP a DLL algoritmov používajú niektoré SAT-solvery aj iné algoritmy. Príkladom môže byť napríklad SAT-solver Heerhugo [15], ktorý využíva Stalmarckov algoritmus [36], ďalšími SAT algoritmami sú stochastické algoritmy (GSAT [24] algoritmus, WalkSAT [28]), alebo algoritmy používajúce binárny rozhodovací diagram [45]. V tejto časti sa budeme venovať už len DP a DLL algoritmom.

1.3.1 Davisov-Putnamov algoritmus

Popis algoritmu [33]:

Vstupom algoritmu je booleovská formula F v CNF tvare.

P: Ak vstupná formula F obsahuje prázdnu klauzulu, vráť „nesplniteľná“.

1. Odstráň z F všetky klauzuly, ktoré obsahujú súčasne oba literály x a $\bar{x}$ premennej x. Ak F je teraz prázdna formula, vráť „splniteľná“.

2. Ak existuje premenná x, taká že F obsahuje $[x]$ a súčastne $[\bar{x}]$, vráť „nesplniteľná“.

3. Ak existuje premenná x, pre ktorú F obsahuje $[x]$, odstráň z F všetky klauzuly obsahujúce x a vymaž $\bar{x}$ zo všetkých klauzúl, ktoré ju obsahujú. Ak je teraz F prázdna formula, vráť „splniteľná“. Ak F obsahuje prázdnu klauzulu, choď na krok 2.

4. Kým existuje čistý literál v F, odstráň všetky klauzuly v ktorých sa vyskytuje. Ak F je teraz prázdna, vráť „splniteľná“.

5. Vyber nejakú premennú x, ktorá má literál v jednej z najkratších klauzúl v F. Nech A je konjunkcia všetkých klauzúl z F, ktorú obsahujú literál x. Nech B je konjunkcia všetkých klauzúl z F obsahujúce literál $\bar{x}$. Nech C je konjunkcia všetkých klauzúl, ktorú neobsahujú ani x ani $\bar{x}$. Nech A' vznikne z A odstránením x z každej klauzuly a B' odstránením $\bar{x}$ z každej klauzuly v B. Nahraď formulu F formulou $(A' \vee B') \wedge C$. Použi distribučný zákon na transformovanie F do CNF. Choď na krok 1.

Podmienka P sa kontroluje vždy po vykonaní jedného z krokov 1 až 5. Krok 3 sa nazýva pravidlo jednoliterálnych klauzúl, krok 4 pravidlo čistých literálov a krok 5 pravidlo rezu alebo pravidlo rezolventy.

Veľakrát je potrebné v prípade splniteľnosti formuly vedieť nájsť aj jedno zo splniteľných ohodnotení. Z tohoto dôvodu sa v 3. a 4. kroku algoritmu neodstraňujú klauzuly ani literály z formuly, ale sa im priraďujú pravdivostné hodnoty a v metóde rezolventy sa vo formule F nechajú aj klauzuly, z ktorých rezolventa vznikla.

Keďže v DP algoritme je pravidlo rezolventy najčastejšie používané pravidlo na elimináciu premenných, počet klauzúl vo formule môže rásť kvadraticky, v najhoršom prípade môžu byť požiadavky DP algoritmu na pamäť až exponenciálne. Preto je použitie DP algoritmu pri väčšom počte premenných prakticky nemožné.

1.3.2 Davisov-Logemannov-Lovelandov (DLL) algoritmus

Na rozdiel od DP algoritmu sú pamäťové požiadavky pre DLL algoritmus väčšinou predvídateľné. DLL je vyhľadávací algoritmus. Vyhľadávací priestor je často reprezentovaný ako binárny strom, v ktorom každý uzol reprezentuje booleovskú formulu vzhľadom na priradenie premenných. Listy reprezentujú kompletné priradenie (všetky premenné majú priradenú pravdivostnú hodnotu), kým vnútorné vrcholy reprezentujú čiastočné priradenie (niektoré premenné majú priradenú pravdivostnú hodnotu, zvyšné sú voľné). Ak ktorýkoľvek list má priradenú pravdivostnú hodnotu 1, potom formula je splniteľná.

Vstupom algoritmu je booleovská formula v CNF tvare.

Pre DDL algoritmus sú dôležité tri pravidlá.

1. *Pravidlo jednotkových literálov.* Ak nesplniteľná klauzula má všetky okrem jedného literálu označené ako nepravdivé, potom ostávajúci voľný literál musí mať pravdivostnú hodnotu 1, aby klauzula bola splniteľná. Takéto klauzuly sa nazývajú jednotkové klauzuly a voľné literály sa nazývajú jednotkové literály.

2. *Konfliktné pravidlo.* Ak priradenie pravdivostných hodnôt premeným sposobí, že všetky literály v klauzule majú pravdivostnú hodnotu 0, potom neexistuje žiadny splniteľný list pre podstrom pod týmto čiatočným priradením a SAT solver potrebuje vykonať návrat a nájsť list v binárnom strome reprezentujúci splniteľnú formulu. Klauzula, ktorej všetky literály majú priradenú hodnotu 0 sa nazýva konfliktná klauzula.

3. *Pravidlo čistých literálov.* Ak je nejaký literál čistý vo všetkých klauzulách vstupnej formuly, potom mu môže byť priradená pravdivostná hodnota 1.

Proces postupného priraďovania pravdivostnej hodnoty 1 všetkým jednotkovým literálom, pokým existujú jednotkové klauzuly sa nazýva Boolean Constraint Propagation a tento proces podrobnejšie opíšeme v podkapitole 1.5.

DLL algoritmus je popísaný v nasledujúcej kapitole.

1.4 DLL algoritmus

Tradičný DLL algoritmus je často prezentovaný ako rekurzívny algoritmus. Toto však nie je zvyčajný spôsob implementácie DLL algoritmu v SAT solveroch. Marques-Silva [27] zobral implementácie DLL algoritmu z rôznych SAT solverov, zovšeobecnil ich a prepísal do iteratívnej podoby ako je v algoritme 1.1.

```
DLL_iterative()
{
  status = preprocess();
  if (status!=UNKNOWN)
    return status;
  while(1) {
    decide_next_branch()
    while(true)
    {
      status = deduce();
      if (status == CONFLICT)
      {
        blevel = analyze_conflict();
        if (blevel < 0)
          return UNSATISFIABLE;
        else
          backtrack(blevel);
      }
      else if (status == SATISFIABLE)
        return SATISFIABLE;
      else break;
  }
}
```

Algoritmus 1.1: Iteratívny DLL algoritmus

Algoritmus 1.1 je rozvetvovací a prehľadávací algoritmus. Algoritmy použité v jednotlivých SAT solveroch sa od tohoto algoritmu líšia vačšinou v detailnej implementácii konkretných metód.

Na začiatku algoritmu sú všetky premenné voľné. V premennej `status` sa zaznamenáva aktuálny stav formuly. Na začiatku je stav neznámy (UNKNOWN), v prípade nájdenia konfliktnej klauzuly sa premenná `status` nastaví na CONFLICT, ak je formula splniteľná potom je stav SATISFIABLE.

Ako prvá sa vykoná metóda `preprocess()`, ktorá zistí, či sa dá triviálne vyhodnotiť splniteľnosť formuly a či je možné priradiť pravdivostné hodnoty niektorým premenným bez vetvenia. V prípade, ak nie je možné triviálne vyhodnotiť splniteľnosť formuly, spustí sa hlavný while cyklus. K vetveniu je spomedzi voľných premenných vybraná jedna premenná, nazývaná aj vybraná premenná a je s ňou spojená úroveň rozhodnutia, ktorá je číslovaná od jednotky a inkrementovaná s každým výberom novej premennej pre vetvenie. Priradenia vykonané v metóde `preprocess()` majú úroveň rozhodnutia 0. O výber premennej pre vetvenie pomocou rôznych heuristík sa stará metóda `decide_next_branch()`. V metóde `deduce()` sa priraďujú pravdivostné hodnoty premenným, ktoré je možné priradiť na základe priradenia vybrannej premennej. Všetky tieto premenné majú potom rovnakú úroveň rozhodnutia ako vybraná premenná. Tento proces sa nazýva rozhodovanie a vykonáva sa pokým existuje premenná, ktorej vieme priradiť pravdivostnú hodnotu.

Ak po skončení rozhodovania sú všetky klauzuly splniteľné, celá klauzula je splniteľná. V príprade, že existuje konfliktá klauzula, aktuálna vetva nevedie k splniteľnému priradeniu, a preto je porebné vykonať návrat. Ak nenastala ani jedna z týchto možností, je potrebné zvoliť novú vybranú premennú v metóde `decide_next_branch()` a znovu pokračovať metódov `deduce()`.

Úlohou metódy `analyze_conflict` je v prípade existencie konfliktnej premennej zistiť, na ktorú hĺbku rozodnutia sa vrátiť a ako zmeniť priradenie pravdivostnej hodnoty. V prípade, že návrat je potrebný na úroveň menšiu ako 0, výsledná formula je nesplniteľná. Aby pri ďalšom prehľadávaní nedošlo k rovnakému priradeniu, ktoré spôsobilo daný konflikt, je možné do pôvodnej formuly pridať klauzulu, ktorú budem nazývať naučená

klauzula alebo konflitkná klauzula. Ako sa konštruuje a ako vyzerá naučená klauzula je popísané v časti venujúcej sa analýze konfliktov.

Uvedené metódy podrobnejšie opíšem v nasledujúcich podkapitolách.

1.4.1 Heuristika

V prípade, ak nie je možná ďalšie volanie funkcie `deduce()`, je potrebné vybrať novú premennú z voľných premenných a priradiť jej pravdivostnú hodnotu. O toto sa stará metóda `decide_next_branch()`. Výber a priradenie pravdivostnej hodnoty sa deje na základe rôznych heuristických metód. Ich použitie môže vytvárať veľkosťou úplne iné prehľadávacie stromy pre rovnaký DLL algoritmus. Preto efektívnosť SAT solvera je veľmi závislá na výbere heuristickej metódy pre výber ďalšej premennej.

V tejto časti uvediem najznámejšie z nich.

Najjednoduchšou heuristikou je náhodný výber jednej premennej z voľných premenných a priradenie náhodnej pravdivostnej hodnoty. Takáto heuristika sa nazýva RAND.

Bohmova heuristika

Podľa Bohmovej heuristiky sa vybere premenná s maximálnym vektorom $(H_1(x), H_2(x), \ldots, H_n(x))$ v lexikografickom poradí, kde

$$H_i(x) = \alpha \max(h_i(x), h_i(\neg x)) + \beta \min(h_i(x), h_i(\neg x))$$

a $h_i(x)$ je počet ešte nevyriešených klauzúl, ktoré obsahujú i literálov a literál x. Hodnoty α a β sú volené heuristicky.

MOM heuristika

MOM je skratka od Maximum Occurrences on clauses of Minimum size. Tomuto názvu zodpovedá aj spôsob výberu premennej touto heuristikou.

Nech $f(l)$ je počet výskytov literálu l v najmenšej nesplniteľnej klauzule. Premenná vybraná touto heuristikou je tá, ktorá maximalizuje hodnotu

$$[f(x) + f(\neg x)] * 2^k + f(x) * f(\neg x)$$

pre dostatočne veľkú konštantu k.

Jeroslow-Wang heuristiky

Pre každý literál l sa vypočíta hodnota

$$J(l) = \sum_{l \in \omega, \omega \in \varphi} 2^{-|\omega|}$$

$l \in \omega$ znamená, že literál l je jeden z literálov klauzuly ω a $\omega \in \varphi$, že klauzula ω je jedna z klauzúl vstupnej formuly φ v CNF tvare.

One-side Jeroslow-Wang heuristika vyberá premennú, ktorá prislúcha literálu s najväčšou hodnotou $J(l)$. Two-side Jeroslow-Wang heuristika vyberá premennú x, pre ktorú je hodnota $J(x) + J(\neg x)$ najväčšia a priraďuje jej pravdivostnú hodnotu 1, ak $J(x) \geq J(\neg x)$ a inak hodnotu 0.

VSIDS heuristika

Významnou heuristikou je heuristika použitá v SAT solveri chaff [30], ktorá sa nazýva Variable State Independent Decaying Sum (VSIDS). Túto heuristiku popíšeme v nasledujúcich krokoch.

1. Každý literál má počítadlo na začiatku inicializované na 0 (v niektorých implementáciách tejto heuristiky na počet výskytov daných literálov vo vstupnej formule).

2. Keď je naučená klauzula pridaná do pôvodnej formuly, inkrementujú sa počítadlá literálov vyskytujúcich v pridanej klauzule. Počítadlo vlastne drží hodnotu, ako často bol daný literál použitý.

3. Vybranou premennou je premenná prislúchajúca k literálu s najväčšou hodnotou počítadla.

4. Všetky počítadlá sú periodicky delené určenou konštantou.

VSIDS heuristika je nezávislá od stavu premenných, pretože hodnota počítadla je nezávislá od hodnôt počítadiel iných literálov. VSIDS heuristika je úspešná, ak je do pôvodnej formuly pridaných veľa naučených klauzúl, v opačnom prípade nie je až taká efektívna.

DLCS a DLIS heuristiky

Obidve heuristiky sú založené na počte výskytov literál v ešte neohodnotených klauzulách. Nech x je premenná, potom pre DLIS (Dynamic Largest Individual Sum) heuristiku platí:

$$Skore(x) = \max(vyskyt(x), vyskyt(\neg x))$$

a pre DLCS (Dynamic Largest Combined Sum) platí:

$$Skore(x) = vyskyt(x) + vyskyt(\neg x).$$

Funkcia $vyskyt(x)$ označuje počet neohodnotených klauzúl, v ktorých sa x nachádza ako literál. Vybranou premennou je premenná s najvyšším skóre.

VSADS heuristika

VSADS (Variable State Aware Decaying Sum) heuristika vzniká kombináciou VSIDS a DLCS heuristík. Nech x je premenná potom platí:

$$Skore(x) = 0.5 * Skore_{DLCS}(x) + \max(Skore_{VSIDS}(x), Skore_{VSIDS}(\neg x)),$$

kde $Skore_{DLCS}(x)$ je skóre premennej x podľa DLCS heuristiky a $Skore_{VSIDS}(x)$ je počítadlo literálu x z VSIDS heuristiky. Vybranou premennou je premenná s najvyšším skóre.

1.4.2 Odvodzovacie mechanizmy, prehľadávanie

Spomedzi viacerých známych odvodzovacích mechanizmov sa ako najlepší ukazuje pravidlo jednotkových literálov. Ako bolo uvedené v predchádzajúcej podkapitole, postupné používanie pravidla jednotkových literálov, kým existujú jednotkové klauzuly, sa nazýva

Klauzuly	Poradie
....	....
....	....
$(\neg a + \neg b + c)$	1.
$(\neg a + \neg b + \neg c)$	2.
$(\neg a + b + c)$	3.
$(\neg a + b + \neg c)$	4.
....	....
....	....

Poradie	Priradenie
1.	....
2.	....
3.	$a = 1$
4.	....
5.	....
6.	....

Tabuľka 1.1: Príklad nechronologického backtrackingu [45]

Boolean Constrain Propagation (BCP) alebo unit propagation, ktorého implementácia sa nachádza práve vo funkcii `deduce()`.

BCP je súčasťou odvodzovacieho mechanizmu väčšiny dnešných moderných SAT solverov. Efektivita SAT solverov je priamo závislá na implementácii ich BCP časti.

V metóde `deduce()` je možné implementovať aj iné pravidlá okrem pravidla jednotkových literálov. Jedným z najznámejších je už hore spomenuté pravidlo čistých literálov. V praxi je však veľmi náročné zistiť, či v danej formule existuje čistý literál, preto sa toto pravidlo v SAT solveroch používa zriedka. Ďalším často skúmaným odvodzovacím pravidlom je zdôvodnenie rovnosti. Zdôvodnenie rovnosti používa prídavnú datovú štruktúru k udržiavaniu informáci, že dve premenné sú ekvivalentné.

V prípade, ak sa počas používania pravidla jednotkových literálov objaví konfliktná klauzula, je potrebné vykonať návrat a zrušiť priradenie pravdivostných hodnôt premenným, ktoré spôsobili konflikt. O toto sa stará metóda `backtrack(level)`. Ide vlastne o inverznú operáciu k operácii BCP. Funkcia `backtrack(level)` závisí od algoritmu použitého vo funkcii `deduce()`.

1.4.3 Analýza konfliktov

Keď metóda `deduce()` vráti, že sa vyskytla konfliktná klauzula, solver potrebuje vykonať návrat a vrátiť predchádzajúce rozhodnutia. O to, na ktorú úroveň rozhodnutia sa vrátiť

a ktoré priradenia je potrebné zrušiť sa stará metóda `analyze_conflict()`.

Najjednoduchšou metódou pre analýzu konfliktov je tzv. chronologický backtracking. Pre každú vybranú premennú sa používa príznak, v ktorom sa zaznamenáva, či sa danej premennej už skúšali priradiť obe pravdivostné hodnoty alebo ešte nie.

Akonáhle sa objaví konflikt, nájde sa premenná s najvyššou úrovňou rozhodnutia, ktorej sa ešte neskúšali priradiť obe pravdivostné hodnoty. Tejto premennej sa priradí opačná pravdivostná hodnota, akú mala doteraz a zrušia sa všetky priradenia, medzi týmto a aktuálnou úrovňou rozhodnutia. Tento backtracking sa používa v niektorých SAT solveroch, napr. satz [25] a tie solvery, ktoré ho používajú sú ekvivalentné rekurzívnemu DLL algoritmu. Avšak tento backtracking je často nie veľmi efektívny.

Lepším riešením je nehľadať premennú s najvyššou úrovňou rozhodnutia, ale nájsť priradenie, ktoré spôsobilo konflikt. Často to je premenná s najnižšou úrovňou rozhodnutia, ktorej sa ešte neskúšali priradiť obe pravdivostné hodnoty. Takáto metóda sa nazýva nechronologický backtracking. V tabuľke 1.1 je príklad nechronologického backtrackingu. Vo vstupnej formule sa nachádzajú zobrazené klauzuly. Výber a priradenie premenných je znázornené v pravej časti tabuľky. Prvé priradenie zaujímavé pre naše klauzuly je na úrovni 3, kde premennej a je priradená pravdivostná hodnota 1. Na úrovniach 4 a 5 sú priradené pravdivostné hodnoty ďalším premenným. Na základe klauzuly 4 sa solver na úrovni 6 rozhodne priradiť premennej b hodnotu 1. Toto priradenie však spôsobí konflikt na klauzulách 1 alebo 2. Podobne preklopenie pravdivostnej hodnoty pre premennú b na 0 spôsobí konflikt v klauzule 3 alebo 4. Vybraná premenná na úrovni 6 bola skúšaná pre obe pravdivostné hodnoty, preto solver potrebuje vykonať návrat. Chronologický backtracking skúsi preklopiť pravdivostnú hodnotu vybranej premennej na úrovni 5, pretože je to premenná s najvyššou úrovňou rozhodnutia, ktorá ešte nebola skúšaná pre obe pravdivostné hodnoty. Avšak preklopenie pravdivostnej hodnoty premennej na úrovni 5 a ani na úrovni 4 konflikt nevyrieši. K vyriešeniu konfliktu je potrebné preklopiť pravdivostnú hodnotu premennej a na úrovni 3. Nechronologický backtracking to dokáže správne identifikovať, a preto vykoná návrat priamo na úroveň 3 a preskočí úrovne 4 a 5.

Pre budúcnosť je dobré zaznamenať tento konflikt v podobe naučenej klauzuly, ktorá

sa pridá do pôvodnej formuly a tým sa oreže priestor prehľadávania do budúcnosti. O toto sa stará mechanizmus nazývaný conflict-driven learning.

Nechronologický backtracking, niekedy označovaný aj ako conflict-directed backjumping, bol spolu s conflict-driven learning prvýkrát implementovaný v SAT solveroch GRASP [27] a relsat [3]. Ďalšie moderné SAT solvery, ako napr. SATO [44], chaff [30] a BerkMin [14] používajú vo svojich implementáciách podobné techniky. Bežne sa používajú dva prístupy na ilustrovanie myšlienky analýzy konfliktov. Jeden používa implikačný graf, druhý pravidlo rezolventu.

Implikačný graf

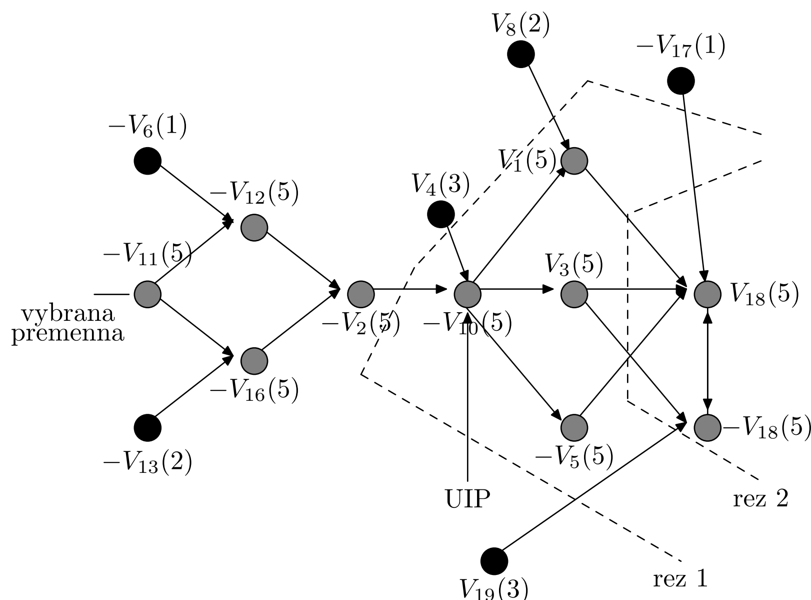
Implikačný graf je orientovaný acyklický graf a je ním možné vyjadriť vzťahy medzi priradeniami premenných počas hľadania ohodnotenia výrokovej formuly. Príklad implikačného grafu je znázornený na obrázku 1.1.

Každý vrchol reprezentuje priradenie. Kladná premenná (V_{cislo}) reprezentuje priradenie 1 a záporná (-V_{cislo}) priradenie 0. Hrana incidentná s vrcholom je dôvod, prečo došlo k danému priradeniu. Napríklad na obrázku 1.1 incidentné hrany do vrcholu $-V_{10}$ sú z vrcholov V_4 a $-V_2$, čo znamená, že ak V_4 je pravda a V_2 je nepravda, potom V_{10} musí byť pravda. Vrcholy, z ktorých vedie hrana do vrcholu V sa nazývajú predchodcovia V. Vrcholy reprezentujúce vybrané premenné nemajú predchodcov. Úroveň rozhodnutia je v grafe označená číslom pri názve vrcholu.

V implikačnom grafe, v ktorom nie sú konflikty je najviac jeden vrchol pre každú premennú. Konflikt nastane ak implikačný graf obsahuje vrcholy pre obe pravdivostné hodnoty jednej premennej. Takáto premenná sa nazýva konfliktná premenná. Na obrázku 1.1 je konfliktnou premennou premenná V_{18}.

Vrchol a v implikačnom grafe dominuje vrcholu b vtedy a len vtedy ak každá cesta z vybranej premennej rovnakej úrovne ako a vedie do vrcholu b cez vrchol a. UIP (Unique Implication Point) je vrchol na aktuálnej úrovni, ktorý dominuje obom vrcholom prislúchajúcim ku konfliktnej premennej. Na obrázku 1.1 vrchol V_{10} dominuje vrcholom V_{18} a $-V_{18}$, a preto V_{10} je UIP. Vybraná premenná na danej úrovni je vždy UIP. Pre daný kon-

Obr. 1.1: Implikačný graf spolu s rezmi prislúchajúcimi naučeným klauzulám [45]



flikt môže existovať viac UIP. Pre náš príklad existujú 3 UIP, V_{11}, V_2 a V_{10}. UIP vrcholy sa zoraďujú v smere od konfliktu, čiže prvým UIP z príkladu je vrchol V_{10}.

Nechronologický backtracking a conflict-driven learning môže byť v SAT solveroch dosiahnutý skúmaním implikačného grafu. Napríklad na obrázku 1.1 sa dá ľahko vidieť, že priradenia $V_1, V_3, -V_5, -V_{17}, V_{19}$ vedú k tomu, že do V_{18} sú priradené obe pravdivostné hodnoty, čiže nastane konflikt. Keďže každé pravdivostné ohodnotenie konfliktnej premennej je vždy nepravdivé, môžeme písať:

$$V_1 \wedge V_3 \wedge \neg V_5 \wedge \neg V_{17} \wedge V_{19} \Rightarrow nepravda$$

Podľa tautológie

$$(A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A)$$

platí

$$pravda \Rightarrow \neg(V_1 \wedge V_3 \wedge \neg V_5 \wedge \neg V_{17} \wedge V_{19})$$

a použitím DeMorganovho pravidla dostávame:

$$pravda \Rightarrow (\neg V_1 \vee \neg V_3 \vee V_5 \vee V_{17} \vee \neg V_{19})$$

Klauzula $(\neg V_1 \vee \neg V_3 \vee V_5 \vee V_{17} \vee \neg V_{19})$ môže byť ako naučená klauzula pridaná do pôvodnej formuly, čím je daný konflikt určený.

Túto formulu sme dostali tak, že sme graf na obrázku 1.1 rozdelili na dve časti. Časť, kde sa nachádzajú všetky vybrané premenné označíme ako určujúca strana a časť, kde sú konfliktné premenné ako konfliktná strana. Ak je rez vybraný tak, že naučená klauzula obsahuje len jednu premennú, ktorej bola priradená pravdivostná hodnota na aktuálnej úrovni a všetkým ostatným premenným v klauzule na nižšej úrovni, potom túto klauzulu nazveme uplatňujúcou (asserting). Po vykonaní backtrackingu sa táto klauzula stáva jednotkovou klauzulou a jednotkovému literálu sa musí priradiť opačná pravdivostná hodnota ako pri konflikte. Toto môže viesť k rýchlemu riešeniu konfliktu, preto je vždy žiadúce, aby naučená klauzula bola uplatňujúca. K tomu je potrebné, aby len jedna hrana z premennej, ktorej bola priradená pravdivostná hodnota na aktuálnej úrovni rozhodnutia, pretínala daný rez. To je ekvivalentné tomu, že UIP sa nachádza na určujúcej strane a všetky premenné priradené po UIP sa nachádzajú na konfliktnej strane. Na obrázku 1.1 sú zobrazené dva rezy, kde rezu 1 zodpovedá naučená klauzula $(\neg V_{19} \vee V_2 \vee \neg V_4 \vee \neg V_8 \vee V_{17})$ a rezu 2 klauzula $(\neg V_1 \vee \neg V_3 \vee V_5 \vee V_{17} \vee \neg V_{19})$.

Akonáhle sa počas behu DLL algoritmu objaví konflikt, SAT solver zostrojí implikačný graf a pokúsi sa nájsť rez, ktorého naučená klauzula je uplatňujúca. Takýto rez sa dá vždy nájsť, lebo vybraná premenná je vždy UIP. Ku každému rezu existuje iná schéma, pomocou ktorej sa vyberá naučená klauzula. Známe sú napr. GRASP schéma, 1-UIP schéma, 2-UIP schéma, All-UIP schéma [45].

Metóda rezolventy

Pri tejto metóde sa využíva pravidlo rezolventy. V algoritme 1.2 je zobrazený algoritmus analýzy konfliktov pomocou metódy rezolventy.

Daný cyklus v algoritme sa vykoná pokým nie je klauzula `c1` uplatňujúca. Tento algoritmus vysvetlíme na príklade, ktorý je zobrazený na obrázku 1.2.

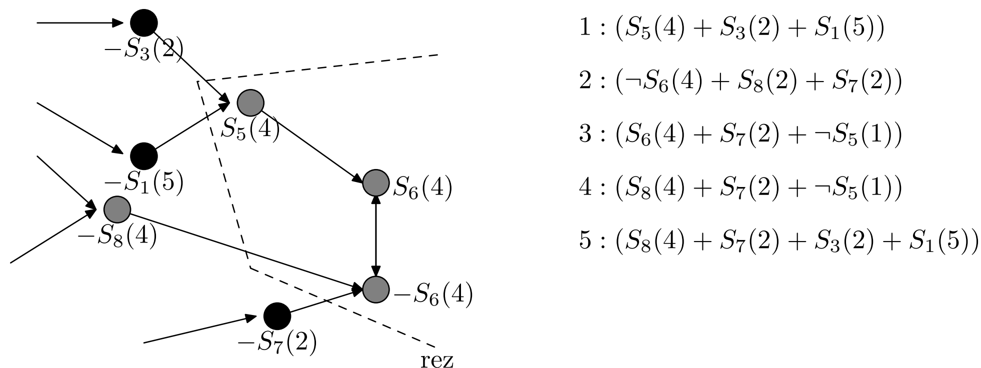
Klauzuly 1 a 2 sú predchádzajúcimi klauzulami pre S_5 a S_6. Predchádzajúca klauzula pre nejakú premennú je jednotková klauzula, ktorej všetky literály okrem literálu prislúchajúcemu danej premennej majú pravdivostnú hodnotu *nepravda*. Klauzula 3 je konfliktnou klauzulou. Algoritmus z 1.2 najpr z konfliktnej klauzuly 3 vyberie literál `lit`, ktorému bola pravdivostná hodnota priradená ako posledná. V našom prípade to je literál $\neg S_6$. Získa sa premenná `prem`, ku ktorej prislúcha literál `lit` (premená S_6). Ďalej sa nájde klauzula `ant`, ktorá je predchádzajúcou klauzulou pre premennú `prem`. V našom príklade je ňou klauzula 2. Funkcia `resolve(c1, ant, prem)` vráti klauzulu, ktorá je rezolventou klauzúl `c1` a `ant` podľa premennej `prem` (klauzula 4).

```
analyzuj_konflikt_rezolvent(){
  if (current_decision_level()==0)
    return -1;
  c1 = find_conflicting_clause();
  do {
    lit = choose_literal(c1);
    prem = variable_of_literal(lit);
    ant = antecedent(var);
    c1 = resolve(c1, ant, prem);
  } while(!asserting(c1));
  add_clause_to_database(c1);
  back_dl = klauzula_asserting_level(c1);
  return back_dl;
}
```

Algoritmus 1.2: Analýza konfliktu pomocou metódy rezolvetu [45]

Klauzulu 5 sme dostali rovnakým spôsobom z klauzúl 2 a 4. Keďže klauzula 5 je už uplatňujúca, cyklus skončí. Tento príklad ukazuje, že naučená formula nájdená pomocou implikačného grafu je ekvivalentná s tou, ktorá vznikne pomocou metódy rezolventu.

Obr. 1.2: Učenie pomocou metódy rezolventu



1.4.4 Predspracovanie, náhodný reštart

Predspracovanie sa vykonáva v metóde `preprocess()`. Pretože táto metóda sa v DLL algoritme vykoná len raz, sú v nej implementované zložitejšie vyvodzovacie mechanizmy, ktoré by nebolo možné použiť pre každý uzol prehľadávacieho stromu. Sú to napr. pravidlo čistých literálov, rekurzívne učenie [23], a algebraické zjednodušovanie [26].

Ďalšou technikou často používanou v moderných SAT solveroch je náhodný reštart. Táto technika sa používa v prípade, keď sa SAT solver pri riešení dostane do konfliktu, ktorého riešenie je ťažké nájsť, lebo zlé rozhodnutie sa vykonalo dávno. Akonáhle je v SAT solveri vyvolaný náhodný reštart, solver opustí aktuálny vyhľadávací strom, ale zapamätá si niektoré naučené klauzuly z minulosti. Z tohto dôvodu sa pôvodné vyhľadávanie nestratí úplne.

1.5 Boolean Constraint Propagation

BCP niekedy nazývané aj unit clause propagation je hlavná operácia pre rozhodovanie v SAT solveroch. Jeho úlohou je nájsť všetky jednotkové klauzuly a priradiť jednotkovým literálom správne pravdivostné hodnoty. BCP toto vykonáva iteratívne až pokým neexistuje žiadna jednotková klauzula alebo sa objavila konfliktná klauzula. V prvom prípade SAT solver vykoná ďalší výber premennej a pokračuje v rozhodovaní, v druhom prípade sa vyvolá funkcia `analyze_conflict()` a návrat.

Efektívnosť celého SAT solvera je z veľkej vačšiny závislá na efektívnosti konkrétnej implementácie BCP. V ďalšom texte opíšeme najznámejšie implementácie BCP algoritmu.

1.5.1 BCP s počítadlami

Pre BCP s počítadlami existuje viacero variantov (schém). Najjednoduchšia schéma pod názvom 2-Couter schéma bola použitá napr. v SAT solveri GRASP [27]. Táto schéma používa dve počítadlá pre každú klauzulu, jedno pre počet literálov, ktorým bola priradená pravdivostná hodnota 1, druhé pre počet literálov s pravdivostnou hodnotou 0. Samozrejme pre každú klauzulu je držaný aj celkový počet literálov v nej. Okrem týchto počítadiel obsahuje každá premenná dva zoznamy, ktoré obsahujú všetky klauzuly, kde sa daná premenná vyskytuje v normálnom alebo negovanom tvare. Keď je nejakej premennej priradená pravdivostná hodnota, upravia sa počítadlá všetkých klauzúl, ktoré obsahujú literály prislúchajúce k danej premennej. Dôvodom použitia dvoch počítadiel je, že pre každú klauzulu vieme hneď zistiť, či je jednotková.

V inej schéme, nazývanej aj 1-Counter schéma, klauzula obsahuje len jedno počítadlo, ktoré udržiava počet nenulových literálov v danej klauzule. Výhodou oproti 2-Counter schéme je, že pri priradení hodnoty nejakej premennej stačí upraviť len jedno počítadlo pre danú klauzulu. Avšak nevýhodou voči 2-Counter schéme je, že pri zisťovaní, či daná klauzula je jednotková, je potrebné spraviť prechod klazulou, či zvyšný literál je voľný alebo pravdivý.

1.5.2 BCP so smerníkmi

Existuje viacero typov BCP so smerníkmi, ale na rozdiel od BCP s počítadlami sa ani v jednom nepoužívajú nijaké počítadlá. SAT solver SATO [44] používa práve BCP so smerníkmi. Každá klauzula si drží dva smerníky pre dva literály v klauzule, jeden sa nazýva head a druhý tail. Na začiatku head ukazuje na prvý literál v klauzule a tail na posledný. Všetkým literálom pred head smerníkom a všetkým za tail smerníkom je priradená hodnota 0.

Každá premenná si drží 4 spájané zoznamy, ktoré obsahujú smerníky na klauzuly. Ak p je premenná, tak spájané zoznamy pre túto premennú sú: `kl_poz_head(p)`, `kl_neg_head(p)`, `kl_poz_tail(p)`, `kl_neg_tail(p)`, pričom `kl_poz_head(p)` je zoznam klauzúl, v ktorých je ako head smerník kladný literál prislúchajúci k premennej p, `kl_neg_head(p)` sú klauzuly, v ktorých je head smerník negovaný literál premennej p a `kl_poz_tail(p)`, `kl_neg_tail(p)` rovnako, ale pre tail smerník. Akonáhle je premennej p pre každú klauzulu C zo zoznamu `kl_neg_head(p)` priradená pravdivostná hodnota 1, prehľadajú sa všetky literály od head literálu až po tail literál. Počas tohto prehľadávania môže dôjsť k 4 prípadom.

1. Ak sa nájde literál, ktorý je pravdivý, tak daná klauzula je splniteľná a prehľadávanie pre túto klauzulu môže skončiť.

2. Ak sa nájde voľný literál m, nastaví sa ako head literál pre túto klauzulu, vyhodí sa táto klauzula zo zoznamu `kl_neg_head(p)` a pridá sa do prislúchajúceho zoznamu pre head pre premennú prislúchajúcu k m.

3. Ak všetky literály medzi head a tail literálom sú nepravdivé tail literál je voľný, potom tail literál je jednotkový literál.

4. Ak nastane rovnaký prípad ako 3, ale tail literál je nepravdivý, potom daná klauzula je konfliktná.

Rovnaký postup sa vykoná aj pre klauzuly v zozname `kl_neg_tail(p)`. Tento algoritmus sa nazýva head/tail algoritmus.

Ďalším zo série BCP algoritmov so smerníkmi je 2-literal watching algoritmus. Podobne ako head/tail algoritmus si aj v tomto každá klauzula drží dva literály nazývané sledované literály. Ku každej premennej p prislúchajú dva zoznamy `poz_watched(p)` a `neg_watched(p)`, ktoré obsahujú klauzuly, kde literál prislúchajúci k p je sledovaný literál a je buď negovaný (`neg_watched(p)`) alebo nie (`poz_watched(p)`). Keď je premennej p priradená pravdivostná hodnota 1, pre každú klauzulu v zozname `neg_watched(p)` sa hľadá nenulový literál. Počas prehľadávania môžu nastať tieto prípady:

1. Ak sa nájde nenulový literál m , ktorý nie je sledovaný literál, odstráni sa smerník na túto klauzulu zo zoznamu `neg_watched(p)` a pridá sa smerník na m do listu premennej prislúchajúcej k m.

2. Ak sa nájde len jeden voľný literál, ktorý je zároveň sledovaný literál, potom táto klauzula je jednotková klauzula a daný literál je jednotkový literál.

3. Ak jediný nenulový literál je ďalší sledovaný literál a je pravdivý, potom daná klauzula je splniteľná.

4. Ak v klauzule neexistuje žiaden nenulový literál, potom daná klauzula je konfliktná.

2-literal watching algoritmus má všetky výhody oproti algoritmom s počítadlami ako mal algoritmus založený na head/tail mechanizme. Naviac v 2-literal watching schéme na rozdiel od ostatných algoritmom trvá vrátenie sa do stavu pred priradením premennej konštantný čas. Je to preto, lebo sledované literály sú posledné literály v klauzule, ktorým je priradená pravdivostná hodnota 0, čiže ktorýkoľvek návrat spôsobí, že sledovaný literál už nebude mať hodnotu 0. Z tohto dôvodu je tento algoritmus rýchlejší ako algoritmy s počítadlami aj ako algoritmus s head/tail mechanizmom.

Kapitola 2

Preklad hašovacej funkcie do výrokovej formuly

2.1 Hašovacia funkcia

Hašovacia funkcia h je zobrazenie $h : X \rightarrow Y$, kde Y je konečná množina, pričom X môže, ale nemusí byť konečná. Hodnota $x \in X$ sa nazýva dokument, správa alebo vzor, hodnota $h(x)$ najčastejšie odtlačok. Bity vstupnej správy dĺžky M označíme $p_1 p_2 ... p_M$. Hašovacia funkcia transformuje vstupnú správu na postupnosť bitov $h_1 h_2 ... h_N$. Formulu, ktorá zodpovedá výpočtu jedného bitu h_i z odtlačku označíme $H_i(p_1, p_2, ..., p_M)$.

Definícia 2.1.1 (jednosmernosť) *Hašovacia funkcia $h : X \rightarrow Y$ je jednosmerná, ak pre dané $y \in Y$ nie je možné efektívne nájsť $x \in X$, pre ktoré platí $h(x) = y$.*

Jednosmernosť, v anglickej terminológii nazývané pre-image resistance znamená, že zo samotného odtlačku nevieme nájsť pôvodný text. Hľadanie pôvodného textu k odtlačku inými slovami znamená hľadať ohodnotenie v premenných $p_1, ..., p_M$ také, že $I_v(H_i(p_1, p_2, ..., p_M)) = h_i$, $\forall x \in \{1, 2, ..., N\}$, kde I_v je interpretácia ohodnotenia v.
Platí:

$$I_v(H_i(p_1, p_2, ..., p_M)) = \begin{cases} 1 & \text{ak } h_i = 1 \\ 0 & \text{ak } h_i = 0 \end{cases}$$

Ďalej definujeme

$$\overline{H_i}(p_1, p_2, ..., p_M) = \begin{cases} H_i(p_1, p_2, ..., p_M) & \text{ak } h_i = 1 \\ \neg(H_i(p_1, p_2, ..., p_M)) & \text{ak } h_i = 0 \end{cases}$$

Formula $\overline{H_i}$ je pravdivá vtedy a len vtedy, ak hašovacia funkcia transformuje vstupnú správu zodpovedajúcu ohodnoteniu v na odtlačok, kde i-ty bit je rovný h_i. Nakoniec definujeme formulu $\mathcal{H}$ a to týmto spôsobom:

$$\mathcal{H}(p_1, p_2, ..., p_M) = \bigwedge_{i=1,...,N} \overline{H_i}(p_1, p_2, ..., p_M)$$

Invertovať postupnosť bitov $h_1 h_2 ... h_N$ znamená nájsť ohodnotenie premenných, pre ktoré je formula $\mathcal{H}(p_1, p_2, ..., p_M)$ splniteľná.

Definícia 2.1.2 (odolnosť 2. vzoru) *Hašovacia funkcia $h : X \to Y$ má odolnosť 2. vzoru, ak pre dané $x \in X$, nie je možné efektívne nájsť $y \in X \setminus \{x\}$, pre ktoré platí $h(x) = h(y)$.*

V angličtine sa odolnosť 2. vzoru označuje ako second pre-image resistance a znamená, že k danému dokumentu nie je možné efektívne nájsť iný dokument s rovnakým odtlačkom.

Podobne ako jednosmernosť, slabá odolnosť voči kolíziám sa dá tiež redukovať na riešenie splniteľnosti nasledujúcej booleovskej formuly.

$$\mathcal{H}'(q_1, q_2, ..., q_M) = \mathcal{H}(q_1, q_2, ..., q_M) \wedge (q_1^{p_1} \vee q_2^{p_2} \vee ... \vee q_M^{p_M})$$

kde

$$q_i^{p_i} = \begin{cases} \neg q_i & \text{ak } p_i = 1 \\ q_i & \text{ak } p_i = 0 \end{cases}$$

Formula $\mathcal{H}'(q_1, q_2, ..., q_M)$ je splniteľná, ak vstupná správa $q_1 q_2 ... q_M$ má rovnaký odtlačok ako správa $p_1 p_2 ... p_M$ a líši sa od nej aspoň v jednom bite.

Definícia 2.1.3 (odolnosť voči kolíziám) *Hašovacia funkcia $h : X \to Y$ je odolná voči kolíziám, ak nie je možné efektívne nájsť dvojicu $x, y \in X$, takú, že $x \neq y$ a $h(x) = h(y)$.*

Odolnosť voči kolíziám inak aj collision resistance znamená, že nevieme nájsť dva dokumenty s rovnakým odtlačkom. Hľadať kolíziu pomocou výrokovej formuly znamená nájsť ohodnotenie, pre ktoré je nasledujúca formula splniteľná. M je dĺžka kolidujúcich správ, ktoré hľadáme.

$$\bigwedge_{i=1,...,N} (H_i(p_1, p_2, ..., p_M) \Leftrightarrow H_i(p'_1, p'_2, ..., p'_M)) \wedge (\neg \bigwedge_{i=1,...,M} (p_i \Leftrightarrow p'_i))$$

2.2 Preklad do SAT jazyka

Na preklad hašovacej funkcie do booleovskej formuly sme použili dve metódy. Jednou bola metóda od autorov Dejana Jovanovića a Predraga Janičića [17], ktorú implementovali v programe hashSAT. Druhá metóda je od autora Milana Šešuma [46], ktorá je vlastne len vylepšením prvej metódy. V nasledujúcom texte budeme metódu Dejana Jovanovića a Predraga Janičića označovať JJ metóda a metódu od Milana Šešuma MS metóda. V ďalšej časti opíšeme hlavnú myšlienku, ktorá je pre obe metódy rovnaká.

2.2.1 Popis programu

Na preklad hašovacej funkcie do inštancie jazyka SAT autori použili vlastnosť jazyka C++, preťaženie operátorov. Logické operátory $\wedge$, $\vee$, $\neg$ a $\oplus$ boli implementované priamo. V algoritme 2.1 je príklad implementácie operátora $\vee$. Vstupom tejto metódy sú dva 32-bitové vektory, kde každému bitu zodpovedá jedna výroková formula a výstupom 32-bitový vektor, kde každému bitu i zodpovedá výroková formula, ktorá vznikla ako *or* formúl zodpovedajúcich i-temu bitu vstupných vektorov.

```
Word Word::operator | (const Word &w) const
{
    Word orWord;   // The or of the words

    /* Resize the bit array */
  orWord.bitArray.resize(this->bitArray.size(), NULL);
```

```
  /* Compute the or */
    for(int i = 0; i < bitArray.size(); i ++) {

        /* Set the operation formula */
        Formula *f = new FormulaOr(bitArray[i],
                                 w.bitArray[i]);
        orWord.setFormulaAt(i, f);
    }

    /* Return the calculated or */
    return orWord;
}
```

Algoritmus 2.1: Implementácia operátora $\vee$

Okrem logických operátorov bolo potrebné implementovať aj ďalšie operátory. Implementácia operátora += je v algoritme 2.2. Podobne ako pri predchádzajúcej metóde sú vstupom tiež dva 32-bitové vektory a výstupom 32-bitový vektor, ktorý vznikol pripočítaním jedného vstupného vektora k druhému modulo 2^{32}.

Program dostane na vstup reťazec a výstupom je výroková formula zodpovedajúca danému vstupnému reťazcu a jeho odtlačku. Každý bit výstupného 128 bitového reťazca je reprezentovaný ako výroková formula $H_i(p_1, p_2, ..., p_M)$, pričom výsledná formula je daná tým, či chceme počítať kolíziu alebo vzor danej hašovacej funkcie.

```
Word& Word::operator += (const Word &w)
{
    Formula* c = new FormulaNT;     // The carry bit

    /* Compute the sum */
    for(int i = bitArray.size() - 1; i >= 0; i --)
    {
        Formula *andF =
          new FormulaAnd(bitArray[i], w.bitArray[i]);
        Formula *orF  =
          new FormulaOr(bitArray[i], w.bitArray[i]);
        Formula *xorF =
          new FormulaXor(bitArray[i], w.bitArray[i]);

        /* Sum of the bits with carry */
```

```
        Formula* sumF = new FormulaXor(xorF, c);

        /* Carry of the sum */
        c = new FormulaOr(andF, new FormulaAnd(c, orF));

        /* Set the sum formula at i-th bit */
        setFormulaAt(i, sumF);
    }

    /* Delete the last carry */
    delete c;

    /* Return the calculated sum */
    return *this;
}
```

Algoritmus 2.2: Implementácia operátora + =

2.2.2 Preklad formuly do CNF

Keďže SAT solvery pracujú s výrokovými formulami v CNF a doterajší preklad hašovacej funkcie do výrokovej formuly nevytvára priamo booleovskú formulu v konjuktívnej normálnej forme, nevyhnutným krokom je preklad výrokovej formuly do CNF.

Problémom je, že priamočiare prepisovanie formuly do CNF má exponenciálnu časovú zložitosť a taktiež aj formula narastá exponenciálne. Z tohto dôvodu použili autori Dejan Jovanović a Predrag Janičić na preklad formuly do CNF tzv. Tseitinova Normálnu Formu.

Tseitinova Normálna Forma

Nech Φ je ľubovoľná formula a nech $Sub(\Phi)$ je množina podformúl formuly Φ. Pre každú neatomickú podformulu $\psi \in Sub(\Phi)$ pridáme novú výrokovú premennú q_ψ. V prípade ak ψ je atomická formula, dostávame $q_\psi = \psi$. Znakom $\otimes$ označíme ľubovoľnú logickú spojku z množiny $\{\wedge, \vee, \oplus\}$. Potom formula

$$q_\Phi \wedge \bigwedge_{\phi \in Sub(\Phi), \phi = \phi_1 \otimes \phi_2} (q_\phi \Leftrightarrow (q_{\phi_1} \otimes q_{\phi_2})) \wedge \bigwedge_{\phi \in Sub(\Phi), \phi = \neg \phi_1} (q_\phi \Leftrightarrow \neg q_{\phi_1})$$

Typ formuly	Tseitinova Normálna Forma
$\phi = \neg\phi_1$	$(q_\phi \vee q_{\phi_1}) \wedge (\neg q_\phi \vee \neg q_{\phi_1})$
$\phi = \phi_1 \wedge \phi_2$	$(q_\phi \vee \neg q_{\phi_1} \vee \neg q_{\phi_2}) \wedge (\neg q_\phi \vee q_{\phi_1}) \wedge (\neg q_\phi \vee q_{\phi_2})$
$\phi = \phi_1 \vee \phi_2$	$(\neg q_\phi \vee q_{\phi_1} \vee q_{\phi_2}) \wedge (q_\phi \vee \neg q_{\phi_1}) \wedge (q_\phi \vee \neg q_{\phi_2})$

Tabuľka 2.1: Transformácia formúl do Tseitinovej Normálnej Formy

je splniteľná vtedy a len vtedy ak bola splniteľná formula Φ. Príklady formúl a ekvivalentných formúl v Tseitinovej Normálnej Forme je v tabuľke 2.1. Prepis tejto formuly do CNF tvaru je už priamočiary.

Kapitola 3

Testovanie

3.1 Spôsob testovania

Testované boli dve hašovacie funkcie, staršia hašovacia funkcia MD5 [32] a jedna z nových hašovacích funkcií BLAKE-32 [2]. Na rozdiel od hašovacej funkcie BLAKE-32, hašovacia funkcia MD5 už bola testovaná na odolnosť voči útoku pomocou SAT solverov. Ako uvádzajú autori Abishek Kumarasubramanian a Ramarathnam Venkatesan vo svojej práci [22], podarilo sa im zaútočiť na jednosmernosť pre prvých 26 kôl hašovacej funkcie MD5. Hľadaniu kolízií pomocou útoku s využitím SAT solverov sa venovala práca [29]. V nej autori Ilya Mironov a Lintao Zhang uvádzajú, že tento útok generuje kolíziu pre MD5 približne každých 100 hodín.

My sme pri hašovacej funkcii MD5 testovali jednosmernosť a pri hašovacej funkcii BLAKE-32 jednosmernosť, odolnosť 2. vzoru a odolnosť voči kolíziám.

Za veľkosť vstupnej správy sme zvolili najväčšiu možnú dĺžku, ktorá je deliteľná 8 a zároveň sa zmestí do jedného 512-bitového vstupného bloku. Keďže pri oboch hašovacích funkciách výroková formula závisí aj od paddingu, táto dĺžka je 440 bitov.

Na zistenie, ako úspešný bude SAT solver, ak sa na vstup pošle kratšia vstupná správa a do potrebnej dĺžky bude v súlade s predpísaným paddingom pre obe hašovacie funkcie doplnená nulovými bitmi sme zvolili aj správu dĺžky 400 bitov.

Pre potreby testovania sme si definovali funkciu $\pi(n)$, ktorá je až na počet kôl rovnaká ako kompresná funkcia konkrétnej hašovacej funkcie. Ak kompresná funkcia obsahuje k kôl, funkcia $\pi(n)$ pozostáva z prvých n kôl kompresnej funkcie, kde $n <= k$. Budeme hovoriť, že funkcia $\pi(n)$ prislúcha k hašovacej funkcii H, ak vznikla z kompresnej funkcie hašovacej funkcie H.

Pre dĺžky 400 a 440 bitov bolo vyrobených 100 vstupných správ. Pre každú z týchto vstupných správ sa vyrobila výroková formula prislúchajúca ku konštrukcii odtlačku tejto správy pre funkiu $\pi(n)$ prislúchajúcu ku konktrétnej hašovacej funkcii. Takto vyrobenené výrokové formuly boli dané na vstup SAT solveru, ktorý na každej z nich bežal vopred stanovený čas. Takýmto spôsobom sme testovali jednosmernosť, odolnosť 2. vzoru a aký vplyv na silu hašovacej funkcie BLAKE-32 majú jednotlivé kroky kola tejto funkcie.

Jednou z vlastností SAT solverov je, že pre jednoduchšie vstupné formuly dokážu zistiť, či je vstupná formula splniteľná alebo nie do pár sekúnd, ale ak vstupná formula dosiahne určitú zložitosť, SAT solver splniteľnosť v reálnom čase nedokáže zistiť. Preto sme na danej výrokovej formule zodpovedajúcej $\pi(n)$ nechali SAT solver bežať relatívne krátky čas. V testoch pre jednosmernosť, odolnosť 2. vzoru a vplyv krokov jedného kola hašovacej funkcie BLAKE-32 sme čas stanovili na jednu minútu.

Na testovanie odolnosti voči kolíziám nebolo vyrobených 100 výrokových formúl zodpovedajúcich 100 vstupným správam ako pri testovaní jednosmernosti a odolnosti 2. vzoru, ale len jedna výroková formula pre vstupnú správu dĺžky 400 alebo 440 bitov, ktorá sa dala na vstup SAT solveru.

Testy boli vykonávané na notebookovom dvojjadrovom procesore Intel Core 2 Duo P8400 2,26GHz s 2GB RAM a 32 bitovom operačnom systéme. Na testovanie boli použité SAT solvery PrecoSAT [5] a RSat [31].

PrecoSAT

SAT solver PrecoSAT získal zlatú medailu zo SAT 2009 competition v kategórii Application. Ako u väčšiny SAT solverov je jeho základom DLL algoritmus. Používa tri predspracovacie techniky, algoritmus silne súvislých komponentov [38], pravidlo poškodeného

literálu [19] a SatElite predprocesor [12]. Na redukciu počtu navštívených klauzúl počas BCP sa v PrecoSAT používajú blokované literály [37].

RSat

SAT solver RSat je držiteľ zlatej medajly zo SAT 2007 competition v kategórii Industrial. RSat je podobne ako PrecoSAT založený na DLL algoritme. Pri BCP používa implementáciu so smerníkmi nazývanú 2-watched literals schéma, ktorá bola opísaná v predchádzajúcej kapitole. Na zaznamenanie konfliktov využíva conflict-driven learning spolu s technikou minimalizácie konfliktných klauzúl [4], ktorá umožňuje redukovať veľkosť naučenej klauzuly a na predspracovanie používa techniky predprocesora SatElite.

3.2 Hašovacia funkcia MD5

Hašovacia funkcia MD5 [32] patrí medzi staršie hašovacie funkcie. Navrhol ju Ronald Rivest v roku 1992 a išlo o vylepšenú hašovaciu funkciu MD4.

Už v roku 1993 sa podarilo nájsť kolíziu pri rôznych inicializačných vektoroch [10]. O tri roky neskôr sa Dobbertinovi podarilo nájsť kolíziu kompresnej funkcie [11]. Avšak prvú kolíziu hašovacej funkcie MD5 predstavila až Xiaoyun Wang na konferencii Crypto 04 v auguste 2004 [41]. Podrobnejší popis jej algoritmu na hľadanie kolízii v MD5 predstavila v [42]. V roku 2006 popísal V. Klíma svoj spôsob hľadania kolízií pomocou tunelovania, pomocou ktorého sa mu podarilo nájsť kolíziu na 3,2 GHz Pentiu 4 v priemere do 17s [21].

Keďže na hľadanie kolízií existuje v súčasnosti viacero úspešných metód, pričom na hľadanie vzorov odtlačkov pre MD5 existujú len metódy pracujúce pre obmedzený počet kôl [34] alebo len so zložitosťou o niečo menšou ako pri prehľadávaní všetkých možností [1], v našej práci sme sa rozhodli pomocou SAT solverov odtestovať len jednosmernosť hašovacej funkcie MD5.

3.2.1 Popis hašovacej funkcie

Z dôvodu používania hašovacej funkcie MD5 v tejto práci, v tejto časti v skratke opíšeme túto hašovaciu funkciu.

Správa, ktorej chceme vytvoriť odtlačok je rozdelená na bloky dĺžky 512 bitov, pričom posledný blok je doplnený na 448 bitov a zvyšných 64 bitov je dĺžka pôvodnej správy. Každý blok správy dĺžky 512 bitov je rozdelený na 16 reťazcov $M_0, .., M_{15}$ dĺžky 32 bitov a je daný na vstup kompresnej funkcii. Samotná kompresná funkcia hašovacej funkcie MD5 sa skladá zo 64 kôl, ktoré su rozdelené do 4 častí po 16 kôl. Na začiatku sú 4 registre a, b, c, d inicializované na známe hodnoty IV_1, IV_2, IV_3, IV_4. Každý register je veľkosti 32 bitov.

Kolo MD5

Jedno kolo hašovacej funkcie MD5 pozostáva zo 6 krokov .

$$\begin{aligned} a &\mathrel{+}= f_i(b, c, d) \\ a &\mathrel{+}= M_i \\ a &\mathrel{+}= r_i \\ a &<<= s_i \\ a &\mathrel{+}= b \end{aligned}$$

$<<=$ označuje bitový cyklický posun doľava o s krokov. r_i je pseudonáhodná konštanta dĺžky 32 bitov. Operácia $+=$ je pripočítanie k pôvodnej hodnote modulo 2^{32}. Za tým nasleduje cyklický posun registrov doprava, čiže platí

$$\begin{aligned} a_{new} &= d \\ b_{new} &= a \\ c_{new} &= b \\ d_{new} &= c \end{aligned}$$

kde a_{new}, b_{new}, c_{new} a d_{new} sú nové registre a, b, c, d.

Pre funkciu f_i platí

$$f_i = \begin{cases} F(x,y,z) = (x \wedge y) \vee (\neg x \wedge z) & \text{pre } 0 \leq i \leq 15 \\ G(x,y,z) = (x \wedge z) \vee (y \wedge \neg z) & \text{pre } 16 \leq i \leq 31 \\ H(x,y,z) = x \oplus y \oplus z & \text{pre } 32 \leq i \leq 47 \\ I(x,y,z) = y \oplus (x \oplus \neg z) & \text{pre } 48 \leq i \leq 63 \end{cases}$$

Nové inicializačné vektory IV_1, IV_2, IV_3, IV_4 vzniknú z pôvodných inicializačných vektorov a registrov a, b, c, d.

$$\begin{aligned} IV_1 &\mathrel{+}= a \\ IV_2 &\mathrel{+}= b \\ IV_3 &\mathrel{+}= c \\ IV_4 &\mathrel{+}= d. \end{aligned}$$

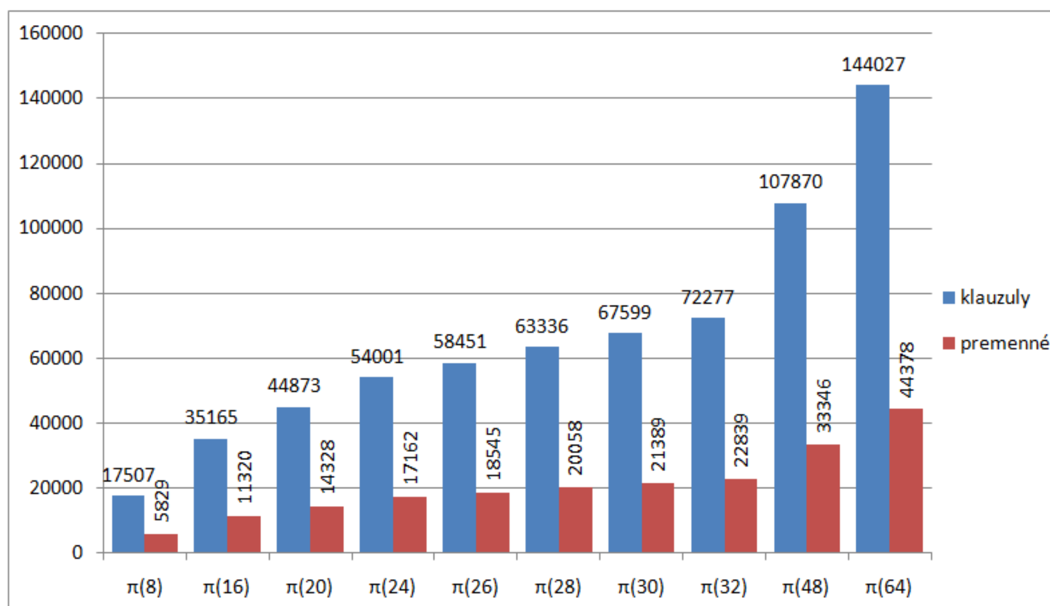
IV_1, IV_2, IV_3, IV_4 sú výstupom kompresnej funkcie a použijú sa ako vstup do ďalšej kompresnej funkcie pre ďalší 512 bitový blok vstupnej správy. Tento proces sa opakuje pre všetky bloky vstupnej správy a jej odtlačkom je výstup poslednej kompresnej funkcie.

3.2.2 Testovanie hašovacej funkcie MD5

Táto podkapitola popisuje testovanie hašovacej funkcie MD5. Ako bolo naznačené na začiatku kapitoly, pre MD5 sa testovala jednosmernosť pre normálnu a upravenú konštrukciu výrokovej formuly.

Jednosmernosť

Porovnať, na ktorej z dvoch výrokových formúl v CNF tvare bude SAT solver bežať dlhšie, kým zistí, či je splniteľná alebo nie, nemusí hneď znamenať porovanie počtu klauzúl a premenných týchto dvoch formúl. Ale keďže pri hašovacej funkcii MD5 sú výrokové formuly pre každé kolo vyrábané rovnakým spôsobom a zložitosti jednotlivých kôl sú približne rovnaké, väčší počet kôl a s tým spojený aj väčší počet premenných a klauzúl



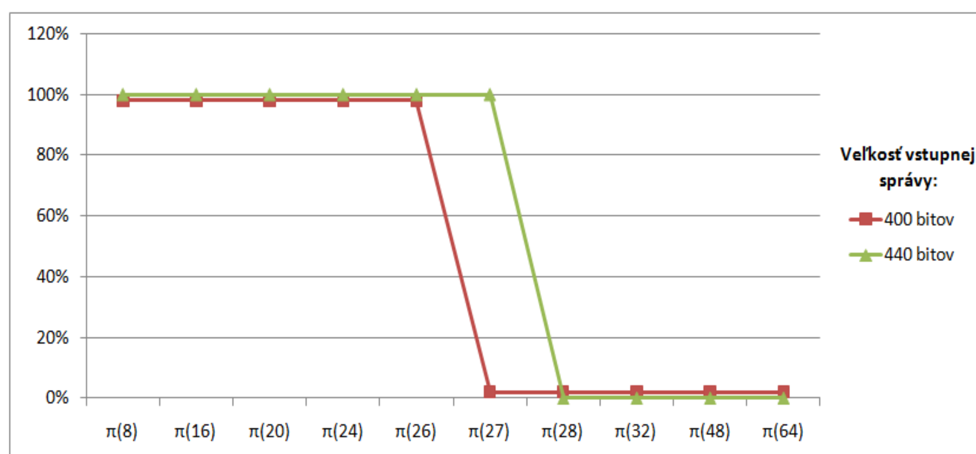
Obr. 3.1: Nárast počtu premenných a klauzúl pre funkciu $\pi(n)$ prislúchajúcu k hašovacej funkcii MD5

automaticky znamená aj zložitejšiu výrokovú formulu pre SAT solver. Z tohto dôvodu je možné zložitosti výrokových formúl pre jednotlivé kolá hašovacej funkcie MD5 ilustrovať pomocou počtu premenných a klauzúl.

Graf 3.1 znázorňuje počty premenných a klauzúl vo výrokových formulách konštruovaných pre jednosmernosť pre funkciu $\pi(n)$ prislúchajúcu k hašovacej funkcii MD5, pričom vstupné správy boli dĺžky 440 bitov. Z výsledkov vyplýva, že dvakrát viac kôl automaticky spôsobí aj dvojnásobný nárast počtu premenných a klauzúl.

Jednosmernosť sme testovali spôsobom popísaným za začiatku kapitoly, výsledky sú uvádzané v grafe 3.2. Pre 400 bitové vstupné správy sme vzor dokázali nájsť pre prvých 26 kôl hašovacej funkcie MD5, pre 440 bitové vstupné správy pre prvých 27 kôl. Pripomíname, že hašovacia funkcia MD5 sa skladá zo 64 kôl.

Zlepšiť tieto výsledky by bolo možné zjednodušením vstupnej výrokovej formuly. O toto sme sa pokúsili v nasledujúcej podkapitole.



Obr. 3.2: Úspešnosť hľadania vzorov pre MD5

Jednosmernosť - upravený preklad do VF

Cieľom tejto úpravy bolo zmenšiť výslednú výrokovú formulu, a tak dosiahnuť lepšie výsledky ako v predchádzajúcej časti.

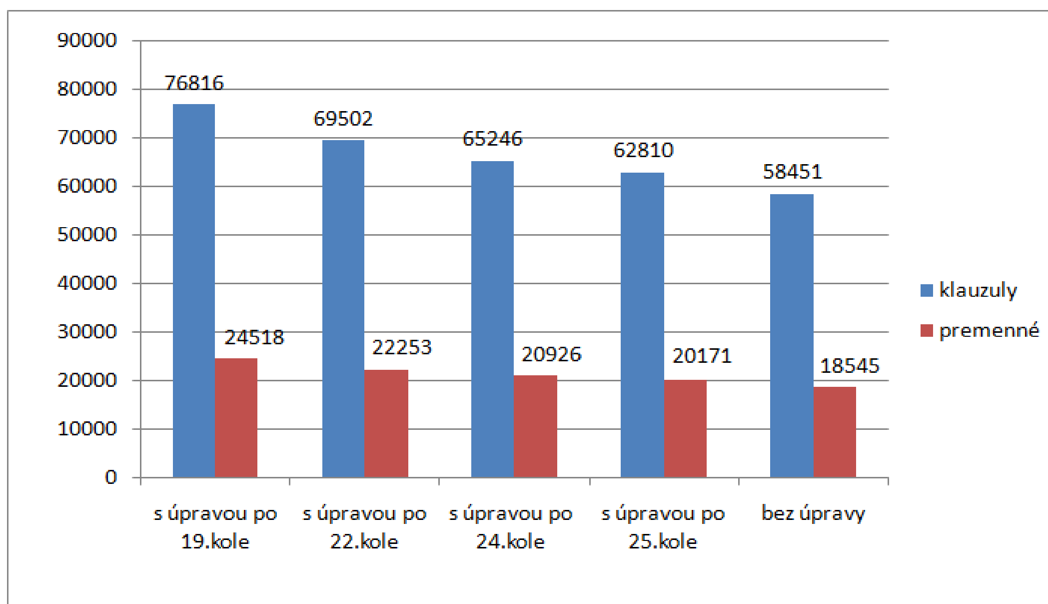
Pri nezmenenom preklade hašovacej funkcie do výrokovej formuly vchádzajú pri každom kole do fukcií F,G,H,I celé formuly pre každý zo 128 výstupných bitov. Pri našej úprave sme medzi dvomi kolami hašovacej funkcie nahradili aktuálne výrokové formuly pre dané bity novými premennými a v ďalšom kole hašovacej funkcie použili už len nové premenné.

Nech kolo, po ktorom bola vykonaná náhrada označíme q, formulu pre i-ty výstupný bit po q kolách hašovacej funkcie MD5 označíme V_i^q a novú premennú, ktorou nahradíme formulu V_i^q označíme ako c_i^q. Potom výslednou formulou pre i-ty bit po r kolách hašovacej funkcie MD5 bude formula

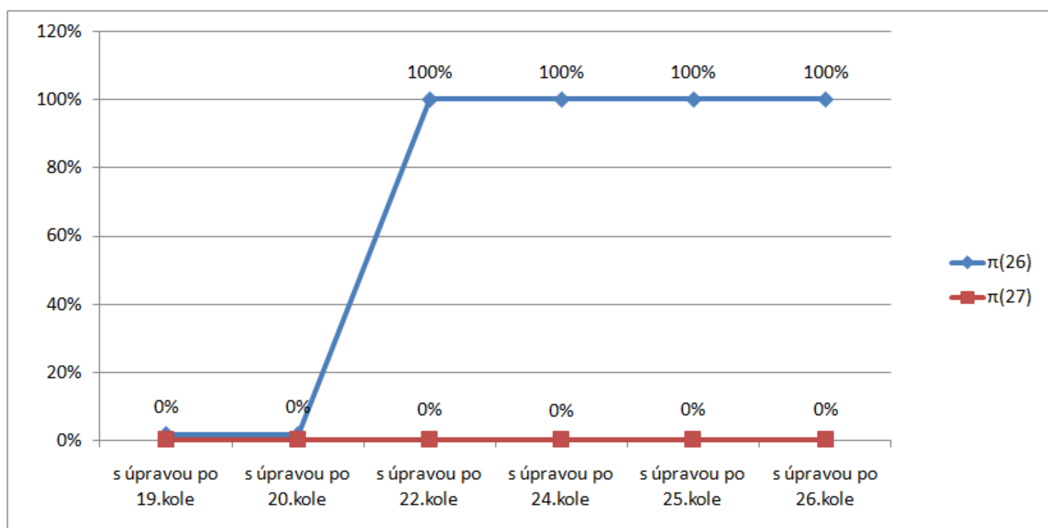
$$(V_i^q \Leftrightarrow c_i^q) \wedge V_i^r,$$

kde V_i^r je formula, ktorá vznikla od náhrady až po r-té kolo hašovacej funkcie MD5. Výslednú formulu sme použili ako vstup do SAT solvera.

V grafe 3.3 je znázornené porovnanie počtu klauzúl a premenných pre upravené a neupravené vytváranie výrokovej formuly pre funkciu $\pi(26)$ prislúchajúcu k hašovacej



Obr. 3.3: Veľkosti klauzúl a premenných pre upravené a neupravené vytváranie výrokovej formuly pre funkciu $\pi(26)$ prislúchajúcu k hašovacej funkcii MD5



Obr. 3.4: Úspešnosť hľadania vzorov pre upravený preklad MD5 do výrokovej formuly pre $\pi(26)$ a $\pi(27)$

funkcie MD5 a 400 bitové vstupné správy. V dolnej časti grafu uvádzame, po ktorom kole sme pri konštrukcii výrokovej formuly vykonali nahradenie aktuálnej výrokovej formuly novou premennou. Ako je vidieť z grafu, touto úpravou sme nedosiahli zmenšenie vstupnej výrokovej formuly. Dôvodom je, že pri preklade upravenej výrokovej formuly do CNF sa musia použiť aj nové premenné, a tým výroková formula v CNF tvare obsahuje viac premenných a tým pádom aj viac klauzúl. Avšak ako je možné vidieť v grafe 3.4 úspešnosť hľadania vzorov pre upravenú výrokovú formulu sa nezhoršila, ale nedosiahli sme tým ani zlepšenie, ktoré bolo cieľom tejto úpravy.

3.3 Hašovacia funkcia BLAKE

Hašovacia funkcia BLAKE [2] je návrh na novú hašovaciu funkciu SHA-3. Spomedzi 51 návrhov z prvého kola sa BLAKE dostal medzi 14 návrhov v druhom kole. Túto hašovaciu funkciu sme zvolili preto, že jej použité operácie sú dosť podobné ako MD5 a neobsahuje operácie, ktoré by veľmi zväčšili výslednú výrokovú formulu (napr. násobenie, operácie s maticami a pod.)

3.3.1 Popis hašovacej funkcie BLAKE-32

Hašovacia funkcia BLAKE používa iteračný mód hašovacej funkcie HAIFA [6] a jej kompresný algoritmus je modifikovaná verzia Bernsteinovej prúdovej šifry ChaCha [20]. BLAKE je rodina štyroch hašovacích funkcií: BLAKE-28, BLAKE-32, BLAKE-48 a BLAKE-64. Jednotlivé verzie sa líšia vstupnými hodnotami, rozdielnym paddingom a rozdielnymi veľkosťami odtlačkov. Keďže sme v našej práci použil 32 bitovú verziu tejto hašovacej funkcie (BLAKE-32), ďalej sa už budeme zaoberať len popisom tejto verzie.

Podobne ako pri hašovacej funkcii MD5 je vstupná správa rozdelená na bloky dĺžky 512 bitov. Posledný blok je doplnený na 447 bitov. Doplnenie pozostáva z jednotkového bitu a potrebného počtu nulových bitov. Do dĺžky 448 bitov je doplnený opäť jednotkovým bitom a zvyšných 64 bitov je rezervovaných pre dĺžku originálnej vstupnej správy.

BLAKE-32 používa 8 iniciálnych hodnôt $IV_0, IV_1, IV_2, IV_3, IV_4, IV_5, IV_6, IV_7$ veľkosti 32-bitov nastavené na rovnaké hodnoty ako SHA-256 a 16 konštánt $c_0, ..., c_{15}$.

Výstupom hašovacej funkcie BLAKE-32 je 256 bitový odtlačok.

Kompresná funkcia

Kompresná funkcia pozostáva zo 4 vstupných hodnôt

- chain hodnoty $h = h_0, ..., h_7$
- bloky správ $m = m_0, ..., m_{15}$
- soľ $s = s_0, ..., s_3$
- počítadlo $t = t_0, t_1$

Všetky slová h_i, m_j, s_k a t_l majú dĺžku 32 bitov. Počítadlo je iné pre každý blok správy a skladá sa z tvoch hodnôt, t_0 a t_1. Ak vstupná správa má dĺžku l, a je rozdelená do p blokov, potom hodnoty $t_0^0, ..., t_0^{p-2}$ sú nastavené na hodnotu 512 a posledná na dĺžku vstupnej správy, čiže l. Hodnoty $t_1^0, ..., t_1^{p-2}$ sú nastavené na 0, hodnota t_1^{p-1} je nastavená podľa toho či vstupná správa presahuje dĺžku 2^{512} bitov.

Soľ je volená používateľom a je nastavená na 0, ak nie je potrebná. V našej práci sme soľ nastavili na 0.

Pre takéto vstupné hodnoty potom môžeme kompresnú funkciu napísať v tvare

$$h' = kompres(h, m, s, t)$$

Inicializácia

Hodnoty $v_0, ..., v_{15}$ sú inicializované ako matica 4x4 nasledujúcim spôsobom.

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} = \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7 \end{pmatrix}$$

Kolo kompresnej funkcie

Kompresná funkcia sa skladá z 10 sérii po osem volaní funkcie G_i. V jednej sérii ide o transformáciu hodnôt $v_0, ..., v_{15}$ pomocou nasledujúcich funkcií:

$$
\begin{array}{cccc}
G_0(v_0, v_4, v_8, v_{12}) & G_1(v_1, v_5, v_9, v_{13}) & G_2(v_2, v_6, v_{10}, v_{14}) & G_3(v_3, v_7, v_{11}, v_{15}) \\
G_4(v_0, v_5, v_{10}, v_{15}) & G_5(v_1, v_6, v_{11}, v_{12}) & G_6(v_2, v_7, v_8, v_{13}) & G_7(v_3, v_4, v_9, v_{14})
\end{array}
$$

Keďže každá séria obsahuje 8 volaní funkcie $G_i, \forall i \in \{0, ..., 7\}$, celá kompresná funkcia hašovacej funkcie BLAKE-32 pozostáva z 80 volaní tejto funkcie. Jedno volanie funkcie G_i budeme nazývať jedným kolom hašovacej funkcie BLAKE-32. Funkcia $G_i(a, b, c, d)$ pozostáva z krokov:

$$
\begin{aligned}
a &= a + b + (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)}) \\
d &= (d \oplus a) >>> 16 \\
c &= c + d \\
b &= (b \oplus c) >>> 12 \\
a &= a + b + (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)}) \\
d &= (d \oplus a) >>> 8 \\
c &= c + d \\
b &= (b \oplus c) >>> 7
\end{aligned}
$$

Funkcia $\sigma_r(i)$ je permutácia prvkov z množiny $\{1, ..., 15\}$ a $r \in \{0, ..., 9\}$ je číslo série.

Po postupnosti kôl nové hodnoty $h'_0, ...h'_7$ vzniknú zo stavov $v_0, ..., v_{15}$, vstupných hod-

nôt $h_0, ...h_7$ a soli $s_0, ..., s_3$:

$$\begin{aligned}
h'_0 &= h_0 \oplus s_0 \oplus v_0 \oplus v_8 \\
h'_1 &= h_1 \oplus s_1 \oplus v_1 \oplus v_9 \\
h'_2 &= h_2 \oplus s_2 \oplus v_2 \oplus v_{10} \\
h'_3 &= h_3 \oplus s_3 \oplus v_3 \oplus v_{11} \\
h'_4 &= h_4 \oplus s_0 \oplus v_4 \oplus v_{12} \\
h'_5 &= h_5 \oplus s_1 \oplus v_5 \oplus v_{13} \\
h'_6 &= h_6 \oplus s_2 \oplus v_6 \oplus v_{14} \\
h'_7 &= h_7 \oplus s_3 \oplus v_7 \oplus v_{15}
\end{aligned}$$

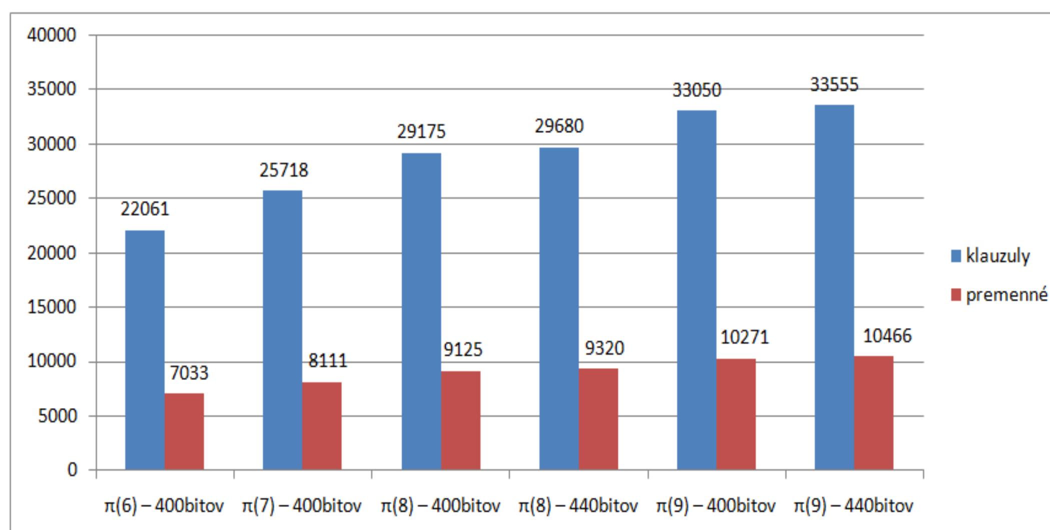
3.4 Testovanie hašovacej funkcie BLAKE-32

Pri hašovacej funkcii BLAKE-32 sme testovali jednosmernosť, odolnosť 2. vzoru, odolnosť voči kolíziám a aký vplyv na silu hašovacej funkcie majú jednotlivé kroky funkcie G_i. Spôsob testovania je popísaný na začiatku kapitoly.

Jednosmernosť, odolnosť 2. vzoru

Keďže sme ako vstupné správy použili texty dĺžky 400 a 440 bitov, tak na jeden odtlačok sa v priemere zobrazí $2^{400}/2^{256}$ resp. $2^{440}/2^{256}$ vstupných správ. Preto nie je možné zistiť, ktorá správa bola skutočným vzorom pre daný odtlačok. Z tohto dôvodu vlastne pri útoku na jednosmernoť hašovacej funkcie, útočíme na odolnosť 2. vzoru, preto v grafoch sú uvedené len výsledky získané z výrokových formúl konštruovaných pre jednosmernosť, výsledky testov pre odolnosť 2. vzoru nebudeme uvádzať. Keďže obidva používané SAT solvery dosahovali skoro identické výsledky, budeme uvádzať len výsledky dosiahnuté SAT solverom PrecoSAT.

Graf 3.5 znázorňuje počty klauzúl a premenných pre funkciu $\pi(n)$ prislúchajúcu k hašovacej funkcii BLAKE-32.

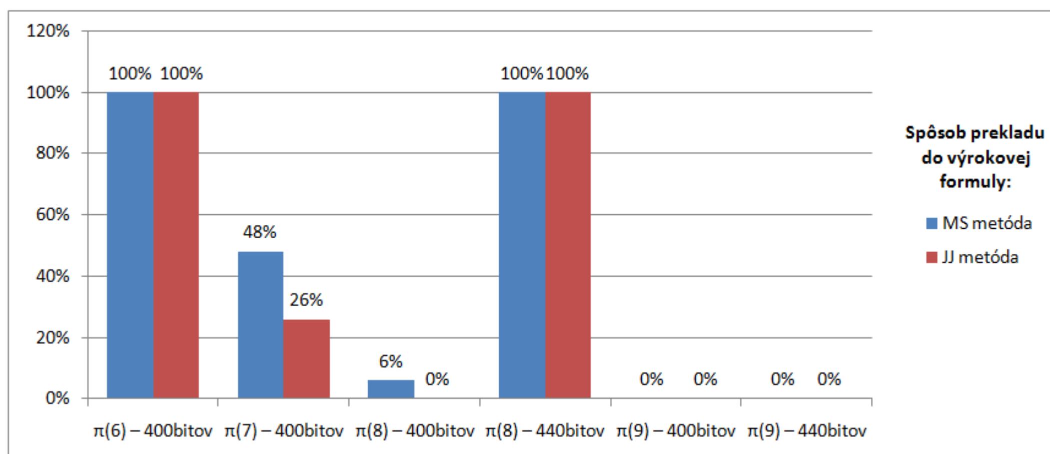


Obr. 3.5: Počty klauzúl a premenných pre funkciu $\pi(n)$ prislúchajúcu k hašovacej funkcii BLAKE-32

Výsledky testovania pre jednosmernosť sú znázornené v grafe 3.6. Testovali sme $\pi(6)$ až $\pi(9)$ pre 400 a 440 bitové vstupné texty a zároveň sme porovnali metódy na preklad hašovacej funkcie do výrokovej formuly.

Z grafu 3.6 vyplýva, že metóda prekladu do výrokovej formuly od Milana Šešuma bola pre 400 bitový vstup o čosi úspešnejšia, ale vzhľadom na celkový počet kôl hašovacej funkcie BLAKE-32, ktorých je 80, to nebolo výrazné zlepšenie. Pre 440 bitový vstup a $\pi(8)$ sme aj pomocou JJ metódy našli všetkých 100 vzorov, preto MS metóda nemohla dopadnúť lepšie. Pre $\pi(9)$ a 400 resp. 440 bitový vstup dosiahli prislúchajúce výrokové formuly takú zložitosť, že JJ a ani MS metóda neboli úspešné.

Je zaujímavé si všimnúť, že aj keď počty klauzúl a premenných v CNF pre funkciu $\pi(9)$ prislúchajúcu k hašovacej funkcii BLAKE-32 boli menšie ako počty klauzúl a premenných v CNF pre funkciu $\pi(26)$ prisluchajúcu hašovacej funkcii MD5, pri funkcii $\pi(9)$ nebol SAT solver úspešný ani v jednom prípade, pričom pri funkcii $\pi(26)$ bol úspešný vo všetkých 100 prípadoch. Z toho vyplýva, že aj keď väčší počet premenných a klauzúl pri väčšom počte kôl v konkrétnej hašovacej funkcii znamená zložitejšiu formulu, pri porovnaní dvoch



Obr. 3.6: Úspešnosť hľadania vzorov oboma metódami pre BLAKE-32

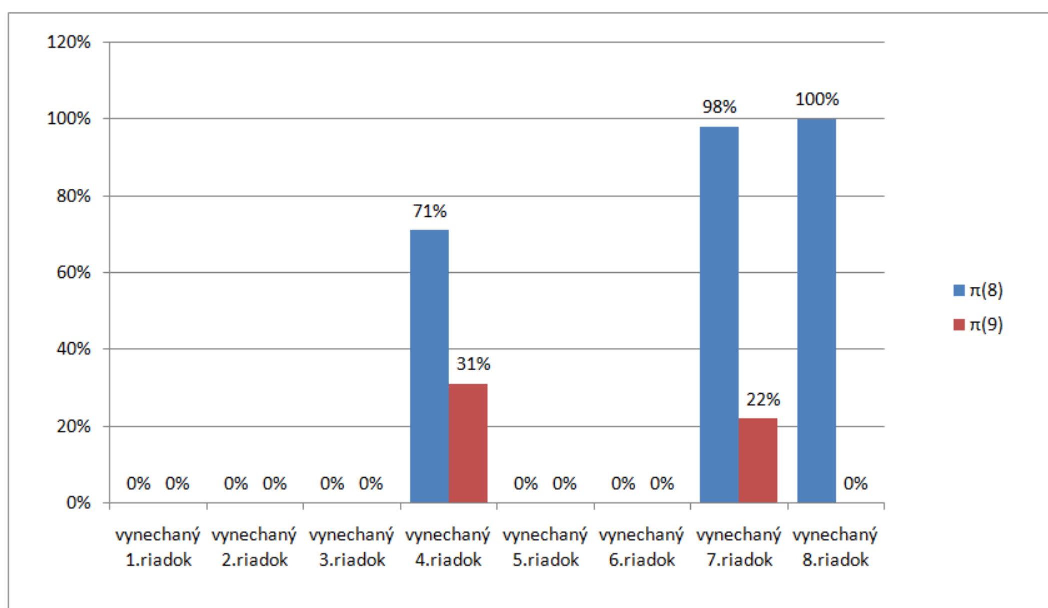
formúl z rôznych hašovacích funkcií nemusí väčší počet premenných a klauzúl znamenať aj zložitejšiu formulu pre SAT solver. Konkrétne pri porovnaní týchto dvoch hašovacích funkcií to znamená starostlivejší návrh hašovacej funkcie BLAKE-32.

Úprava kompresnej funkcie

V tejto časti sme zisťovali, aký vplyv na silu hašovacej funkcie BLAKE-32 majú jednotlivé kroky funkcie G_i.

Postupne sme pre funkciu $\pi(8)$ resp. funkciu $\pi(9)$ vždy odstránili jeden z 8. krokov funkcie G_i pre každé $i \in \{0, ..., 7\}$. Výsledky pre 400 bitové vstupné správy sme zaznamenali do grafu 3.7.

Pre funkciu $\pi(8)$ dokázalo odstránenie 4.,7. alebo 8. riadku funkcie G_i výrazne oslabiť hašovaciu funkciu, avšak už pre funkciu $\pi(9)$ nebolo toto oslabenie také výrazné. Preto z pohľadu analýzy pomocou SAT solverov ani vynechanie jedného kroku funkcie G_i výrazne neoslabí hašovaciu funkciu BLAKE-32.



Obr. 3.7: Vplyv jednotlivých krokov funkie G_i na silu hašovacej funkcie BLAKE-32

				Kolízia	
Dĺžka vstupnej správy	**Počet kôl**	**Veľkosť formuly**	**Čas RSat/ PrecoSAT**	**Vzor 1**	**Vzor 2**
400 bitov	$\pi(8)$	67312/21677	360s/96s	nájdená kolízia	nájdená kolízia
	$\pi(9)$	75062/23969	61351s/-	x	x
	$\pi(10)$	83554/26477	29612s/-	x	x
440 bitov	$\pi(8)$	68962/22307	4s/13s	nájdená kolízia	nájdená kolízia
	$\pi(9)$	76712/24599	39594s/-	x	x
	$\pi(10)$	85204/27107	32145s/-	x	x

Tabuľka 3.1: Hľadanie kolízií pre BLAKE-32

	Kolízia - $\pi(8)$	
Dĺžka vstupnej správy	**Vzor 1**	**Vzor 2**
400 bitov	$m_0 = d8adc13a$ $m_8 = 746f6ff3$ $m_1 = 1a14df84$ $m_9 = 1cd41cce$ $m_2 = e6bfa06f$ $m_{10} = 64b9cc32$ $m_3 = bb66a8a3$ $m_{11} = 14a05063$ $m_4 = 21985968$ $m_{12} = 5c7b8000$ $m_5 = 17cc10b8$ $m_{13} = 00000001$ $m_6 = 7d30fe25$ $m_{14} = 00000000$ $m_7 = 72c93c9a$ $m_{15} = 00000190$	$m_0 = 5a65efc6$ $m_8 = 3b6876d4$ $m_1 = a2be4949$ $m_9 = 6186ffab$ $m_2 = 22c6fa01$ $m_{10} = 12c0ead4$ $m_3 = 416b15b0$ $m_{11} = d258f4fe$ $m_4 = c3cf02c9$ $m_{12} = be508000$ $m_5 = b52688d2$ $m_{13} = 00000001$ $m_6 = a3b9e743$ $m_{14} = 00000000$ $m_7 = cc3a5bdc$ $m_{15} = 00000190$
440 bitov	$m_0 = cff822e5$ $m_8 = 4ee1dc3c$ $m_1 = 010b2ebf$ $m_9 = dc652307$ $m_2 = ded4fd0e$ $m_{10} = 00430872$ $m_3 = 987dc38f$ $m_{11} = 74d5e870$ $m_4 = 6088da85$ $m_{12} = 4a99d8ea$ $m_5 = e61117c6$ $m_{13} = 6fbd1281$ $m_6 = 1141c0cf$ $m_{14} = 00000000$ $m_7 = 9354b079$ $m_{15} = 000001B8$	$m_0 = c59c8dd2$ $m_8 = 5946af3b$ $m_1 = 987e7fb6$ $m_9 = 2b4d59c7$ $m_2 = e25ec092$ $m_{10} = c496ccf9$ $m_3 = d5002c87$ $m_{11} = 8c70cee0$ $m_4 = ddde2384$ $m_{12} = 6496db44$ $m_5 = 7b0f4ec4$ $m_{13} = 567bc181$ $m_6 = 067a6137$ $m_{14} = 00000000$ $m_7 = 599ceda2$ $m_{15} = 000001B8$

Tabuľka 3.2: Nájdené kolízie pre funkciu $\pi(8)$ prislúchajúcu k hašovacej funkcii BLAKE-32

Hľadanie kolízií

V poslednej časti sme sa venovali hľadaniu kolízií pre hašovaciu funkciu BLAKE-32 pomocou SAT solvera. V tomto prípade sme výrokovú formulu konštruovali len metódou Dejana Jovanovića a Predraga Janičića.

Kolízie sme hľadali pre funkcie $\pi(8)$, $\pi(9)$ a $\pi(10)$ prislúchajúce k hašovacej funkcii BLAKE-32. Vzhľadom na to, že pre funkcie $\pi(9)$ a $\pi(10)$ sa nám už kolízie nepodarilo nájsť, hľadanie pre väčší počet kôl by bolo zbytočné.

Výsledky testov uvádzame v tabuľke 3.1. V stĺpci čas sú uvádzané časy potrebné pre SAT solver PrecoSAT a SAT solver RSat na nájdenie kolízie. Znak - znamená, že sme kolízie pre daný SAT solver a danú funkciu $\pi(n)$ nehľadali a znak `x`, že sa nám do daného času kolíziu nepodarilo nájsť. Konkrétne príklady nájdených kolízií pre 400 a 440 bitové vstupné správý pre funkciu $\pi(8)$ pomocou SAT solvera RSat uvádzame v tabuľke 3.2. Hodnoty pre vstupné správy sú zadané v šestnástkovej sústave.

V tabuľke 3.2 uvedená kolízia pre funkciu $\pi(8)$ a 400 resp. 440 bitový vstup je len jedna z viacerých, ktoré sa nám podarilo nájsť, lebo každé spustenie SAT solvera na danej výrokovej formule môže vygenerovať stále iné ohodnotenie premenných a tým aj inú kolíziu. Je to spôsobené náhodnosťou v algoritmoch SAT solverov.

Dôvodom nenájdenia kolízií pre väčší počet volaní funkcie G_i bola veľká zložitosť prislúchajúcich výrokových formúl.

3.5 Zhrnutie

Pri hašovacej funkcii MD5 sa nám do jednej minúty pomoocu SAT solvera podarilo nájsť vzory pre 400 bitové vstupné správy pre prvých 26 kôl a pre 440 bitové vstupné správy pre prvých 27 kôl. Týmito výsledkami sme dosiahli rovnaký, pre 440 bitové vstupné správ dokonca o jedno kolo lepší výsledok ako autori Abishek Kumarasubramanian a Ramarathnam Venkatesan vo svojej práci [22], v ktorej hľadali vzory tiež pomocou SAT solverov. Pre väčší počet kôl sa nám vzor nepodarilo nájsť za viac ako 8 hodín. Domnievame sa, že v súčasnosti je použitie SAT solverov na hľadanie vzorov pre plnú hašovaciu funkciu

MD5 neúčinné.

Pri hašovacej funkcii BLAKE-32 zatiaľ nebola pomocu SAT solverov testovaná ani jedna z vlastností hašovacej funkcie (jednosmernosť, odolnosť 2. vzoru, odolnosť voči kolíziám). Pre dôvody popísané na začiatku kapitoly sme sa nakoniec obmedzili na testovanie jednosmernosti a odolnosti voči kolíziám. Pri jednosmernosti sa nám podarilo zaútočiť len na prvých 8 kôl hašovacej funkcie. Pre väčší počet kôl sa nám vzor nepodarilo nájsť za viac ako 8 hodín. Pripomíname, že celkový počet kôl hašovacej funkcie BLAKE-32 je 80.

Pri testovaní, aký vplyv na silu hašovacej funkcie BLAKE-32 majú jednotlivé kroky funkcie G_i sa nám podarilo zistiť, že nepoužitie ktoréhokoľvek z ôsmych krokov G_i nemá výrazný vplyv na silu hašovacej funkcie.

Pri hľadaní kolízie sme sa sústredili na vstupné texty dĺžky 400 aj 440 bitov, pre ktoré existuje dostatok kolidujúcich správ. Podobne ako pri jednosmernosti sa nám podarilo nájsť kolíziu len pre prvých 8 kôl hašovacej funkcie BLAKE-32. Pre viac kôl nebol SAT solver úspešný do času presahujúceho 8 hodín.

Na základe výsledkov dosiahnutých pri testovaní hašovacej funkcii BLAKE-32 je zrejmé, že použitie súčasných SAT solverov na kryptoanalýzu tejto hašovacej funkcie je neúčinné.

Kapitola 4

Záver

V 1.kapitole sme popísali čo sú to SAT solvery, aké algoritmy používajú a podrobnejšie opísali DP a DLL algoritmus, ktoré sú najčastejšie používanými algoritmami v SAT solveroch. My sme sa rozhodli SAT solvery použiť na kryptoanalýzu hašovacích funkcií. V 2. kapitole sme opísali metódy prekladu hašovacej funkcie do výrokovej formuly, jednu od autorov Dejana Jovanovića a Predraga Janičića a druhú od autora Milana Šešuma. Pomocou týchto metód sme potom vyrobili výrokové formuly zodpovedajúce hašovacím funkciám MD5 a BLAKE-32 pre testovanie jednosmernosti a hľadanie kolízií. Samotné testovanie sme opísali v 3. kapitole.

Pre hašovaciu funkciu MD5 sa nám podarilo nájsť vzory k odtlačkom zodpovedajúcim prvým 26 kolám kompresnej funkcie pre 400 bitový vstup a pre prvých 27 kôl pre 440 bitový vstup. Pre hašovaciu funkciu BLAKE-32 sme vzory našli pre prvých 8 kôl kompresnej funkcie pre 400 resp. 440 bitový vstup a kolíziu sa nám podarilo nájsť tiež pre prvých 8 kôl kompresnej funkcie pre obe dĺžky vstupov.

Ďalšie možnosti, ako zlepšiť výsledky SAT solverov pri kryptoanalýze hašovacích funkcií vidíme v dvoch oblastiach. Jednou z nich je snaha upraviť SAT solvery tak, aby boli schopné lepšie pracovať s formulami, ktoré zodpovedajú hašovacím funkciám resp. konkrétnej hašovacej funkcii. Druhou a podľa nášho názoru účinnejšou metódou je snaha vylepšiť preklad hašovacej funkcie do výrokovej formuly tak, aby výsledná výroková for-

mula bola jednoduchšia. Myslíme si však, že pokroky ani v jednej z týchto dvoch oblastí nedokážu vylepšiť útoky proti hašovacím funkciám pomocou SAT solverov natoľko, aby boli reálne použiteľné na nové hašovacie funkcie.

Literatúra

[1] Aoki, K. a Sasaki, Y. Preimage Attacks on One-Block MD4, 63-Step MD5 and More. *Selected Areas in Cryptography: 15th International Workshop, SAC 2008, Sackville, New Brunswick, Canada, August 14-15, Revised Selected Papers* (2009), 103–119. http://dx.doi.org/10.1007/978-3-642-04159-4_7.

[2] Aumasson, J. P., Henzen, L., Meier, W. a Phan, R. C. W. *SHA-3 proposal BLAKE*. Submission to NIST, 2008. http://131002.net/blake/.

[3] Bayardo, R. J. a Schrag, R. C. Using CSP look-back techniques to solve real-world SAT instances. *Proceedings of the National Conference on Artificial Intelligence (AAAI)* (1997).

[4] Beame, P., Kautz, H. a Sabharwal, A. *Understanding the power of clause learning*. Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-2003), 2003.

[5] Biere, A. *P{re,i}coSAT@SC'09*. Johannes Kepler University, Linz, Austria, 2009. http://fmv.jku.at/precosat/preicosat-sc09.pdf.

[6] Biham, E. a Dunkelman, O. *A framework for iterative hash functions - HAIFA*. ePrint report 2007/278, 2007.

[7] Chvtal, V. a Szemerdi, E. Many hard examles for resolution. *Journal of the ACM 35*, 4 (1988), 759–768.

[8] Davis, M., Logemann, G. a D.Loveland. A machine program for theorem-proving. *Communications of the ACM 5* (1962), 394–397.

[9] Davis, M. a Putnam, H. A computing procedure for quantification theory. *Journal of the ACM 7* (1960), 201–215.

[10] den Boer, B. a Bosselaers, A. Collisions for the compression function of md5. In *In Advances in Cryptology, Proceedings of EUROCRYPT '93* (1993), pp. 293–304.

[11] Dobbertin, H. *Cryptoanalysis of MD5 Compress.* German Information Security Agency, 1996.

[12] Eén, N. a Biere, A. *Effective preprocessing in SAT through variable and clause elimination.* SAT'05, 2005.

[13] Eén, N. a Sörensson, N. *An Extensible SAT-solver.* Chalmers University of Technology, Sweden, 2003. http://minisat.se/Main.html.

[14] Goldberg, E. a Novikov, Y. BerkMin: A fast and robust SAT-solver. *Proceedings of the IEEE/ACM Design, Automation and Test in Europe (DATE)* (2002).

[15] Groote, J. F. a Warners, J. P. The Propositional Formula Checker HeerHugo. *J. Autom. Reason. 24*, 1-2 (2000), 101–125. http://dx.doi.org/10.1023/A:1006366304347.

[16] Hammer, P. L. a Rudeanu, S. *Boolean Methods in Operations Research and Related Areas.* Springer-Verlag, Berlin, Heidelberg, New York, 1968.

[17] Jovanović, D. a Janičić, P. *Logical Analysis of Hash Functions.* Mathematical Institute, Faculty of Mathematics, Serbia and Montenegro, 2006.

[18] Joščák, D. *Hledání kolizí v hašovacích funkcích.* Univerzita Karlova v Praze, Matematicko-fyzikální fakulta, Diplomová práca, 2006.

[19] Kautz, H. a Selman, B. *Unifying SAT-based and Graph-based Planning.* Department of Computer Science Cornell University, Shannon Laboratory AT&T Labs – Researc, 1999.

[20] Khazaei, S., Meier, W., Rechberger, C., Aumasson, J. P. a Fischer, S. *New features of Latin dances: analysis of Salsa, ChaCha, and Rumba.* In FSE, 2008.

[21] Klima, V. Tunnels in hash functions: Md5 collisions within a minute. *Cryptology ePrint Archive: Report 105/2006*. http://eprint.iacr.org/2006/105.

[22] Kumarasubramanian, A. a Venkatesan, R. *Inversion Attacks on Secure Hash Functions using Sat Solvers.* Microsoft Research, Redmond & Bangalore, 2007. http://www.csi.ucd.ie/staff/jpms/Events/SAT07/slides/kumarasubramanian-sat07- talk.pdf.

[23] Kunz, W. a Pradhan, D. Recursive Learning: A new implication technique for efficient solutions to CAD problems: Test, veri—cation and optimization. *IEEE Transactions on Computer-Aided Design* (1994), 1143–1158.

[24] Levesque, H., Mitchell, D. a Selman, B. A new method for solving hard satisfiability problems. *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)* (1992), 459–465.

[25] Li, C. M. a Anbulagan. Heuristics based on unit propagation for satisfiability problems. *Proceedings of the fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)* (1997), 366–371.

[26] Marques-Silva, J. P. *Algebraic simplification techniques for propositional satis—ability.* Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'2000), September 2000.

[27] Marques-Silva, J. P. a Sakallah, K. A. GRASP - a search algorithm for propositional satisfiability. *IEEE Transactions in Computers 48*, 5 (1999), 506–521.

[28] McAllester, D., Selman, B. a Kautz, H. Evidence for invariants in local search. *Proceedings of the Fourtteenth National Conference on Artificial Intelligence (AAAI'97)* (1997), 321–326.

[29] Mironov, I. a Zhang, L. Applications of SAT solvers to cryptanalysis of hash functions. In *In Theory and Applications of Satisfiability Testing 2006* (2006), pp. 102–115.

[30] Moskewicz, M. W., Madigan, C. F., Zhao, Y. a Malik, S. Chaff: Engineering an efficient SAT solver. *Proceedings of the Design Automation Conference(DAC)* (2001).

[31] Pipatsrisawat, K. a Darwiche, A. *RSat 2.0: SAT Solver Description.* University of California, Los Angeles, 2007. http://reasoning.cs.ucla.edu/rsat/index.html.

[32] Rivest, R. RFC1321 - The MD5 Message-Digest Algorithm. http://www.faqs.org/rfcs/rfc1321.html.

[33] Ryan, L. *Efficient algorithms for clause-learning SAT solvers.* Simon Fraser Univerisity, Diplomová práca, 2004. www.cs.sfu.ca/ mitchell/papers/ryan-thesis.ps.

[34] Sasaki, Y. a Aoki, K. Preimage Attacks on Step-Reduced MD5. In *ACISP '08: Proceedings of the 13th Australasian conference on Information Security and Privacy* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 282–296.

[35] Silva, J. M. *The Impact of Branching Heuristics in Propositional Satisfiability Algorithms.* Technical University of Lisbon, IST/INESC, Lisbon, Portugal. http://algos.inesc.pt/ jpms.

[36] Stalmarck, G. *A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula.* Technical report, European Patent N 0403 454 (1995), US Patent N 5 276 897, Swedish Patent N 467 076 (1989), 1989.

[37] Sörensson, N. MS 2.1 and MS++ 1.0. *SAT Race 2008 editions* (2008).

[38] Tarjan, R. *Depth first search and linear graph algorithms.* SIAM Journal on Computing 1, 1972.

[39] Toman, E. *Vybrané partie z logiky.* Univerzita Komenského, Bratislava, 1998.

[40] Toman, E. *Vybrané partie z logiky.* Univerzita Komenského, Bratislava, 2005. http://www.dcs.fmph.uniba.sk/texty/logika.pdf.

[41] Wang, X., Lai, X., Feng, D. a Yu, H. *Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD.* CRYPTO 2004, August 2004. http://eprint.iacr.org/2004/199.

[42] Wang, X. a Yu, H. *How to break MD5 and other hash functions, presented at EUROCRYPT 2005.* 2004. http://www.infosec.sdu.edu.cn/paper/md5-attack.pdf.

[43] Yi, J. *The Effect of VSIDS on SAT Solver Performance.* University of California, Santa Cruz, 2007. http://www.soe.ucsc.edu/classes/cmps217/Fall07/Project/jaeheon/ final_paper/final_paper/input-dist-subm.pdf.

[44] Zhang, H. SATO: an efficient propositional prover. *Proceedings of the International Conference on Automated Deducation (CADE'97)* (1997), 272–275.

[45] Zhang, L. *Searching for truth: Techniques for satisfiability of boolean formulas.* Princeton University, Princeton, NJ, USA, Dizertačná práca, 2003.

[46] Šešum, M. Osobná korešpodencia.