DLL knižnica hašovacej funkcie Blake3 pre Python

[1]*Patrik ZELEŇÁK,* [2]*Miloš DRUTAROVSKÝ*

[1,2]Katedra elektroniky a multimediálnych telekomunikácií, Fakulta elektrotechniky a informatiky, Technická univerzita v Košiciach, Slovenská republika

[1]patrik.zelenak@student.tuke.sk, [2]milos.drutarovsky@tuke.sk

Abstrakt – **Článok opisuje nový stavebný kryptografický blok, ktorým je vysoko-paralizo-vateľná hašovacia funkcia Blake3. Pozornosť je venovaná opisu vnútornej štruktúre funkcie Blake3, ktorá využíva štruktúru Merklového hašovacieho stromu. Jadro funkcie Blake3 podporuje spracovanie dát s využitím vektorizovaných SIMD inštrukcií. Článok sa zaoberá prepojením optimalizovanej implementácie funkcie Blake3, napísanej v jazyku Rust, s jazykom Python vo forme DLL knižnice. Uvedené sú výsledky rýchlostných testov, na základe ktorých sú porovnané vybrané implementácie funkcie Blake3. Blake3 je porovnaný s klasickými hašovacími funkciami ako SHA2 alebo SHA3 na systéme s viacerými vláknami.**

Kľúčové slová – **Kryptografia, Python, Blake3, SIMD, Rust, Dynamická knižnica**

I. Úvod

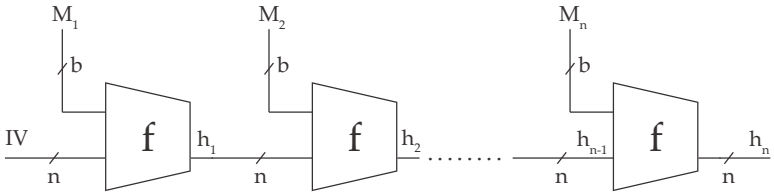
Kryptografia je veda, ktorá využíva matematické metódy utajovania obsahu a preukázateľnosti pôvodu prenášaných správ. Jej cieľom je utajiť obsah prenášaných správ, čo zahŕňa riešenie základných bezpečnostných otázok ako je dôvernosť a integrita dát, autentizácia a autorizácia užívateľov a podobne. Bezpečný prenos a utajenie prenášaného obsahu je možné dosiahnuť s vy-užitím kryptografických algoritmov a protokolov (digitálne podpisy, výmena kľúčov, autentizácia správ, a pod.), ktorých základ tvoria kryptografické primitíva.

V súčasnosti sa pri vývoji softvérových aplikácií využívajú rôzne programovacie jazyky. Za-ujímavou voľbou pre vývoj aplikácií je aj jazyk Python [1]. Python je v súčasnosti jedným z naj-populárnejších programovacích jazykov, ktorého využitie stúplo za posledných 5 rokov až o 12,1% v porovnaní s inými programovacími jazykmi. Vďaka svojím výhodám, ako napríklad intuitívna syntax, jednoduchá interakcia s inými programovacími jazykmi, či široká developerská komunita, získal svoje postavenie v oblastiach dátovej vedy (data science), strojového učenia, vývoja webových aplikácií a v mnohých ďalších odvetviach. Programovací jazyk Python disponuje kryptografickými knižnicami ako napríklad Cryptography, PyCryptodome, Hashlib a mnoho ďalších. Opis týchto základných kryptografických knižníc pre jazyk Python je možné nájsť v bakalárskej práci [2]. Kryptografické knižnice sú v práci opísane nie len z hľadiska rozsahu podpory kryptografických algoritmov a protokolov, ale aj z hľadiska rýchlosti spracovania kryptografických algoritmov. Jazyk Python obsahuje aj niekoľko zásadných nevýhod ako napríklad vysoká spotreba operačnej pamäte alebo jeho nízka rýchlosť [3]. Štandardne je vykonávanie programu v jazyku Python pomalšie ako v jazyku C, čo predstavuje jednu z jeho nevýhod. Tieto nevýhody však vieme čiastočne eliminovať prepojením externých modulov, napísaných v inom rýchlejšom programovacom jazyku.

Tento článok obsahuje stručný opis novej paralizovateľnej hašovacej funkcie Blake3, ktorá využíva štruktúru binárneho Merklového stromu [4]. Blake3 je špeciálne navrhnutá pre rýchle hašovanie dát a overenie integrity dát. Rýchlosť funkcie Blake3 značne ovplyvňuje jej štruktúra. Vďaka Merklovému stromu je Blake3 vysoko paralelizovateľná a každé vlákno CPU jednotky môže spracovávať svoj blok dát nezávisle. Jadro funkcie Blake3 je špeciálne navrhnuté pre podporu vektorizovaných SIMD inštrukcií, čo má za následok ďalšie zvýšenie rýchlosti pri spracovaní dát.

II. Paralizovateľná hašovacia funkcia Blake3

Konvenčné hašovacie funkcie (napríklad SHA2 alebo SHA3) spracúvajú dáta sekvenčným spôsobom, to znamená, že sa vstupné dáta typicky rozdelia na bloky dát o určitej veľkosti a tie sa spracúvajú s istou vzájomnou závislosťou (viď. Obr. 1). Každý z blokov je spracovaný kompresnou funkciou *f*. Výstup kompresnej funkcie sa stáva vstupom pre následujúcu kompresnú funkciu. To znamená, že pre spracovanie i-teho bloku M_i, musíme zahašovať všetky predchadzajúce bloky dát. Hašovací kód h_n, teda výstup z hašovacej funkcie, dostávame po aplikácií kompresnej funkcie na posledný blok dát.



Obr. 1 Štruktúra konvenčných hašovacích funkcií

Hašovacia funkcia Blake3 však pracuje úplne inak. Blake3 je moderná hašovacia funkcia vyvinutá v roku 2020 skupinou vývojárov za účelom čo najefektívnejšie a najrýchlejšie zahašovať dáta a overiť ich integritu. Blake3 nevyužíva sekvenčné spracovanie dát ako to je u konvenčných hašovacích funkcií ale svoju štruktúru buduje na základoch Merklového stromu [5]. Merklova stromová konštrukcia prináša niekoľko výhod ako napríklad neobmedzenú možnosť paralelizmu pri spracovaní veľkého bloku dát, overovanie integrity streamovaných dát, či overenie integrity dát pri ich sťahovaní z viacerých zdrojov simultánne. K rýchlosti funkcie Blake3 prispieva aj podpora vektorizovaných SIMD inštrukcií, ktorými je možné spracovanie dát výrazne urýchliť na súčasných procesorových platformách.

III. Vnútorná štruktúra funkcie Blake3

Merklov hašovaci strom je hašovacia konštrukcia vynájdená Ralphom Merklom, ktorá vykazuje výhodne vlastnosti pre efektívny paralelizovateľný výpočet integrity dát. Ide o binárnu stromovú štruktúru, ktorej listy reprezentujú bloky vstupných dát C_i s veľkosťou s a každý rodičovský vrchol je odvodený od jeho dvoch potomkov. Realizáciu Merklového stromu pre hašovaciu funkciu Blake3 možno opísať v štyroch krokoch.

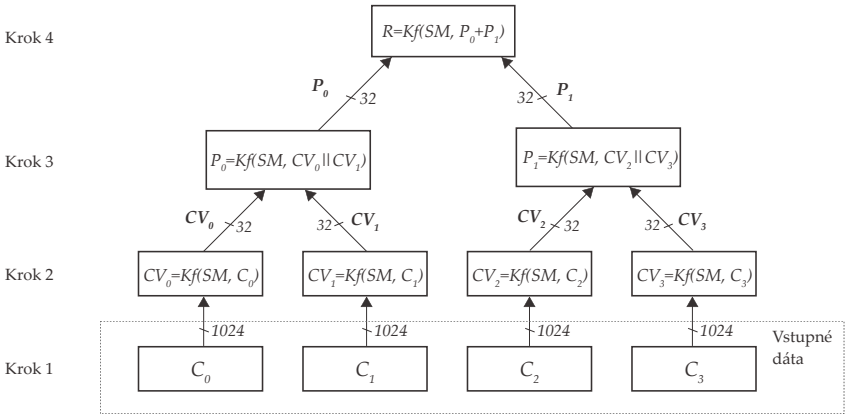
1) **Rozdelenie vstupných dát do blokov C_i.** Vstupné dáta rozdelíme do blokov dát C_i o veľkosti $s = 1024$ bajtov. Tieto bloky dát budeme označovať ako segmenty. Segmenty tvoria listy binárného hašovacieho stromu.

2) **Nezávislé hašovanie blokov dát C_i.** Na jednotlivých segmentoch C_i sa realizuje nezávisle spracovanie dát. V prípade hašovacej funkcie Blake3 sa segmenty C_i spracúvajú kompresnou funkciou, ktorej výstup je označova-ný ako zreťazovacia hodnota (chaining value) v binárnom strome. Veľkosť zreťazovacej hodnoty (CV_i) v Merklovom strome pre funkciu Blake3 je 32 bajtov. Zreťazovacie hodnoty slúžia na tvorbu nadradených rodičovských uzlov Merkloveho stromu. Práve kvôli nezávislému spracovaniu segmentov poskytuje Merklová stromová štruktúra neobmedzenú možnosť paralelizmu pri spracovaní dát.

3) **Tvorba a hašovanie rodičovských uzlov P.** Každý rodičovský uzol pozostáva práve z dvoch hodnôt a to zo zraťazovacej hodnoty ľavého potomka a zraťazovacej hodnoty pravého potomka. Na Obr. 2 je vidieť, že rodičovský uzol P_0 pozostáva z jeho ľavého potomka C_0 a pravého potomka C_1, konkrétne z ich zreťazovacích hodnôt CV_0 a CV_1. Hašovaním rodičovského uzla získavame opäť zreťazo-vaciu hodnotu, ktorá sa stáva jedným zo vstupov svojho nadriadeného rodičovského uzla. Takéto hierarchické spracovávanie uzlov, smerom od listov ku koreňu, vykonávame až kým sa nedostaneme ku koreňu binárneho stromu.

4) **Hašovanie koreňa stromu R.** Vo všeobecnosti sa hašovanie koreňa stromu realizuje rovnakým spôsobom ako hašovanie rodičovských uzlov. Na Obr. 2 môžeme vidieť, že koreň stromu R má dvoch potomkov P_0 a P_1. Zreťazovacie hodnoty potomkov koreňa tvoria vstup pre kompresnú funkciu *Kf()* a výstupom *Kf()* získavame hašovací kód Merklovej hašovacej štruktúry v hašovacej funkcii Blake3. Teda po aplikácií $H(P_0 \| P_1)$

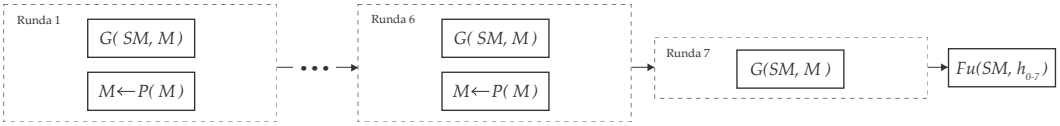
získavame výstupný hašovací kód funkcie Blake3.



Obr. 2 Merklova stromová štruktúra hašovacej funkcie Blake3

Obr. 2 demonštruje rozdelenie vstupných dát na segmenty (C_0, C_1, C_2, C_3) o veľkosti 1024 bajtov. Ak je posledný segment menší ako 1024 bajtov, bude mu pridaná vhodná výplňová schéma (padding) v podobe nulových bajtov. Následne sa segmenty spracúvajú funkciou *Kf()*, ktorá na výstup vracia zreťazovaciu hodnotu. Zreťazovacie hodnoty dvoch segmentov tvoria vstupy pre výpočet zretazovacej hodnoty ich nadradeného rodičovského uzla. Kompresná funkcia *Kf()* požaduje dva vstupné parametre, a to stavovú maticu *SM* s veľkosťou 64 bajtov a blok vstupných dát *M* taktiež o veľkosti 64 bajtov. Listy a rodičovské uzly sa spracúvajú odlišným spôsobom. V prípade rodičovských uzlov je hodnota *M* rovná $CV_l||CV_p$. Pre listy binárneho stromu platí, že sa list (1024 bajtov) rozdelí na 64-bajtové subbloky $M_0, ..., M_{15}$, ktoré sa následne spracúvajú sekvenčne [2].

Kompresná funkcia *Kf()* spracúva dáta v 7 rundách. Každá runda, okrem poslednej, je tvorená tvz. *G* funkciou a následnou permutáciou vstupných dát, čo možno vidieť na Obr. 3. V poslednej runde sa taktiež aplikuje *G* funkcia, no permutácia vstupných dát sa už nevykonáva. Konečným krokom kompresnej funkcie *Kf()* je transformácia stavovej matice označená ako *Fu()*.



Obr. 3 Bloková schéma kompresnej funkcie *Kf()*

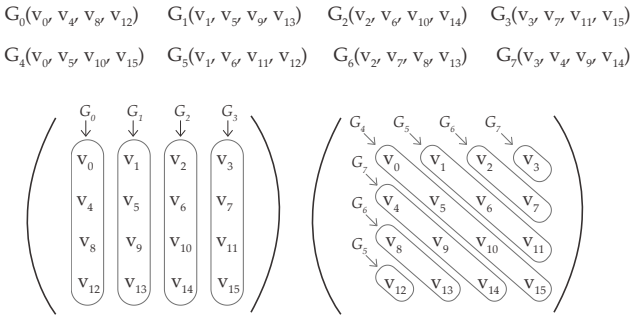
Funkcia Blake3 podporuje spracovanie dát pomocou vektorizovaných SIMD inštrukcií, ktoré využíva práve *G* funkcia. *G* funkcia má za úlohu zmeniť stav matice *SM* v závislosti od jej aktuálneho stavu a bloku vstupných dát. Stavovú maticu *SM* (64 bajtov) možno zapísať v podobe šestnástich 32-bitových slov. Stavová matica je inicializovaná pri každom spracovaní bloku dát a obsahuje parametre ako Inicializacný vektor IV, zreťazovacie hodnoty h_{0-7}, počítadlo segemntov t_{0-1} a podobne [6]. Tieto slová možno vyjadriť v matici 4×4 následovne:

$$\begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ IV_0 & IV_1 & IV_2 & IV_3 \\ t_0 & t_1 & b & d \end{pmatrix} \rightarrow \begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix}$$

Obr. 4 Stavová matica *SM* 4×4, ktorá obsahuje šestnástich 32-bitových slov vyjadrených aliasmi v_{0-15}

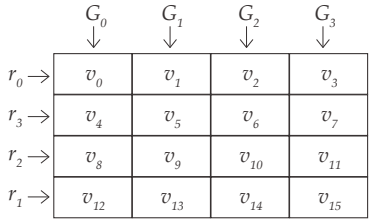
Matica vľavo (viď. Obr. 4) predstavuje parametre stavovej matice *SM* a matica vpravo predstavuje aliasy hodnôt stavovej matice (matice vľavo), ktoré slúžia na jednotný opis parametrov stavovej matice [6].

G funkcia pozostáva z 8 operácií $G_{0–7}$ a je aplikovaná na každý stĺpec matice 4×4, a potom aj na každú diagonálu. Spracovacie stĺpcov a diagonál prebieha v 8 operáciách $G_{0–7}$ paralelne. Na Obr. 5 môžeme vidieť grafické znázornenie ale aj matematické vyjadrenie týchto G operácií, pričom slova $v_{0–15}$ predstavuju aliasy stavovej matice SM.



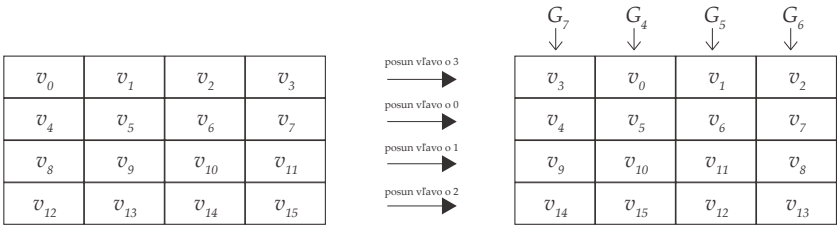
Obr. 5 Grafické znázornenie a matematické vyjadrenie realizácie G operácií paralelným spôsobom

Funkcia Blake3 podporuje 3 rôzne spôsoby využitia SIMD inštrukcii. Článok opisuje prvý z troch spôsobov využitia SIMD inštrukcií, ktorý je podobný ako u predchodcov funkcie Blake3 (Blake2b, resp. Blake2s). Slová matice sa rozdelia po riadkoch do štyroch 128-bitových vektorov $r_{0–3}$ (Obr. 6). Následne sa vykoná vektorizovaná G funkcia (ktorá pracuje s vektorovými inštrukciami) na všetkých 4 stĺpoch paralelne. To znamená, že operácie $G_{0–3}$ (spracovanie stĺpcov) sa vykonajú naraz po riadkoch pomocou SIMD inštrukcií. Inak povedané, každý riadok obsahuje prvé slovo z jednotlivých $G_{0–3}$ operácii a spracovaním prvého riadku vykonáme spracovanie prvého slova vo všetkých $G_{0–3}$ operáciách. Takýmto spôsobom pokračujeme v spracovaní všetkých vektorov $r_{0–3}$.



Obr. 6 Paralelné spracovanie stĺpcov operáciami $G_{0–3}$

Po spracovaní stĺpcov (vykonanie operácií $G_{0–3}$) nasleduje diagonalizácia riadkov (rotácia riadkov), čo zabezpečí vhodné preusporiadanie riadkov tak, aby bolo možné opäť vykonať vektorizované spracovanie stĺpcov, tentokrát pre operácie $G_{4–7}$ (viď Obr. 7). Po vykonaní operácii $G_{4–7}$, nasleduje inverzná diagonalizácia riadkov, teda preusporiadenie riadkov do pôvodného stavu. Takéto využitie SIMD inštrukcií je vhodné aplikovať na malé vstupy. V prípade veľkého množstva vstupných dát je vhodné využiť niektorý z ďalších dvoch spôsobov využitia SIMD inštrukcií [6].



Obr. 7 Rotácia vektorov a následné paralelné spracovanie stĺpcov operáciami $G_{4–7}$ operácií

IV. Paralizovateľná dynamická knižnica Blake3 pre jazyk Python

Autori hašovacej funkcie Blake3 napísali optimalizovanú, plne paralizovateľnú implementáciu funkcie Blake3 v jazyku Rust [7]. Implementácia funkcie Blake3 je dostupná aj v jazyku

C, avšak nie je paralizovateľná. Programovací jazyk Rust je relatívne nový programovací jazyk, ktorý bol prvýkrát predstavený v roku 2014 spoločnosťou Microsoft. Ide o programovací jazyk, ktorý by vďaka svojej rýchlosti a bezpečnosti mohol predstavovať alternatívu programovacieho jazyka C resp. C++. Vývojári jazyka Rust propagujú jeho schopnosť pomáhať vývojárom vytvárať rýchle a bezpečné aplikácie. Programovací jazyk Rust neobsahuje garbage collector, runtime alebo manuálnu správu pamäte, čo súvisí s jeho rýchlosťou a bezpečnosťou a taktiež zabraňuje chybám segmentácie. Rust je špeciálne navrhnutý pre bezpečnú správu pamäte a vývoj súbežných (concurrency) a paralizovateľných aplikácií.

A. Prepojenie optimalizovanej implementácie funkcie Blake3 s jazykom Pythom

Prepojili sme optimalizovanú implementáciu Blake3, ktorej zdrojový kód je vytvorený v jazyku Rust, s jazykom Python. Prepojenie externého Rust modulu (Blake3) sme realizovali vytvorením dynamickej Python knižnice, ktorú je možné použiť v Python aplikácií na hašovanie súboru s využitím funkcie Blake3. Výhodou tohto prepojenia je zachovanie rýchlosti, ktorú nám ponúka jazyk Rust a zároveň využitie komfortu jazyka Python.

B. Nástroje pre vytvorenie dynamickej Python knižnice

Pre vytvorenie dynamického Python modulu sme využil nástroje zobrazene v Tabuľke 1.

Tabuľka 1
Nástroje použité na vytvorenie dynamickej kniznice

Programovací jazyk/modul	Verzia
Rust (programovací jazyk)	rustc 1.56.1
Cargo (balíčkový manažér pre Rust)	cargo 1.56.0
Python (programovací jazyk)	Python 3.8.10
pip (balíčkový manažér pre Python)	pip 22.0.3
PyO3 (prepájací modul)	0.15.1
Blake3 (hašovacia funkcia)	1.3.1

Pred samotným vytvorením dynamického Python modulu je nutné namapovať kód externého Rust modulu s jazykom Python. Takéto mapovanie je možné realizovať pomocou Rust balíčka PyO3 [8]. Pri vytváraní dynamického Python modulu sme postupovali manuálnym spôsobom, ktorý sa javí ako najjednoduchší. Manuálne vytvorenie dynamického Python modulu spočíva vo vytvorení dynamickej knižnice v jazyku Rust. Následou modifikáciou dynamickej knižnice získame dynamický Python modul.

V. Experimentálne testovanie rýchlosti funkcie Blake3

Vytvorili a experimentálne sme otestovali Python implementáciu hašovacej funkcie Blake3, ktorá je založená na prepojení optimalizovanej implementácie napísanej v jazyku Rust.

A. Porovnanie rôzných implenetácii hašovacej funkcie Blake3

Pre paralizovateľné implementácie funkcie Blake3 sme vykonali rýchlostné testy, ktorých výsledok je zobrazený v Tabuľke 2.

Tabuľka 2
Rýchlostný test hašovacej funkcie Blake3, vykonaných na procesore I5-8250U a OS Windows 10, pre rôzne implementácie funkcie Blake3

Rýchlostný test – Hašovacia funkcia Blake3 otestovaná na rôznych implementáciách				
Implementácia	Vlákna (Threads)/ Jadrá	100 MB	200 MB	1 GB
Opt. Rust impl.	1 vlákno	0,064 s	0,128 s	0,655 s
	2 vlákna	0,033 s	0,069 s	0,341 s
	3 vlákna	0,023 s	0,043 s	0,242 s
	4 vlákna	0,021 s	0,036 s	0,201 s
	5 vlákien	0,019 s	0,031 s	0,165 s
	6 vlákien	0,015 s	0,029 s	0,153 s
	7 vlákien	0,014 s	0,028 s	0,142 s
	8 vlákien	0,013 s	0,027 s	0,138 s
rust_py_blake3	1 vlákno	0,065 s	0,129 s	0,665 s
	2 vlákna	0,035 s	0,089 s	0,346 s
	3 vlákna	0,024 s	0,046 s	0,301 s
	4 vlákna	0,022 s	0,043 s	0,203 s
	5 vlákien	0,021 s	0,034 s	0,171 s
	6 vlákien	0,018 s	0,032 s	0,167 s
	7 vlákien	0,017 s	0,031 s	0,156 s
	8 vlákien	0,016 s	0,029 s	0,146 s

Tabuľka 2 predstavuje výsledky rýchlostného testu funkcie Blake3 pre optimalizovanú implementáciu funkcie Blake3 a prepojenú implementáciu rust_py_blake3, ktorá využíva dynamickú Python knižnicu. V prípade optimalizovanej paralizovateľnej implementácie napisanej v programovacou jazyku Rust a nami vytvorenej demonštračnej implementácie rust_py_blake3 bolo možné hašovanie otestovať z hľadiska ryhlosti na rôznom počte aktívných vlakien procesora. Z meraní vyplýva, že implementácia napísaná v jazyku Rust je v porovnaní s implementáciou rust_py_blake3 rýchlejšia v priemere o približne 6.7%.

B. Porovnanie funkcie Blake3 s inými modernými hašovacími funkciami

Hašovacia funkcia Blake3, spolu s ďalšími modernými hašovacími funkciami, bola rýchlostne testovaná. Na základe výsledkov meraní sme vybrané hašovacie funkcie porovnali.

Tabuľka 3
Rýchlostný test hašovacej funkcie Blake3, vykonaných na procesore I5-8250U a OS Windows 10, pre rôzne implementácie funkcie Blake3

Porovnanie vybraných hašovacich funkcií			
Súbor	SHA-256	SHA3-256	rust_py_blake3
20 MB	0,0958 s	0,1177 s	0,0051 s
100 MB	0,4807 s	0,5901 s	0,016 s
500 MB	2,3168 s	2,8908 s	0,0696 s
2 GB	10,0687 s	12,0961 s	0,2934 s

Tabuľka 3 zobrazuje porovnanie konvenčných hašovacích funkcií SHA256 a SHA3-256 s prepojenou implementáciou hašovacej funkcie Blake3. Vybrané konvenčné hašovacie funkcie (z rodiny SHA2 a SHA3) sú implementované v kryptografickej knižnici Cryptography. Z Tabuľky 3 je vidieť, že prepojená implementácia hašovacej funkcie Blake3, využívajúca všetkých 8 dostupných vlákien, je podľa očakávaní výrazne rýchlejšia ako konvenčné hašovacie funkcie a to niekoľko násobne.

VI. Záver

Článok sa venuje opisu kryptografickej hašovacej funkcii Blake3, ktorá svojou paralizovateľnou štruktúrou založenou na Merklovom hašovacom strome vykazuje vhodné vlastnosti pre rýchle a paralelné hašovanie dát. Článok tiež opisuje princíp využitia SIMD inštrukcií vo funkcii Blake3, čo má vplyv na ďalšie zvýšenie rýchlosti hašovania dát. Využitá optimalizovaná implementácia hašovacej funkcie Blake3 je napísaná v jazyku Rust. Článok opisuje prepojenie externého Rust modulu s jazykom Python prostredníctvom dynamickej knižnice. Prepojenie modulu Blake3 s jazykom Python spája výhodne vlastnosti jazyka Rust, ako napríklad rýchlosť vykonávania programu, s komfortom jazyka Python. Blake3 je vhodná na rýchle hašovanie veľkého množstva dát s využitím moderných viacjadrových precesorov.

Literatúra

[1] "Python language," [online] [cit. 2022-05-17], Dostupné z: https://www.python.org/.
[2] P. Zeleňák, "Kryptografia v pythone," [online] [cit. 2022-05-17], košice: Technická univerzita v Košiciach, Fakulta elektrotechniky a informatiky, 2022. 87s.
[3] W. J. Chun, *Core Python Programming*. USA: Prentice Hall PTR, 2006.
[4] J. Chapweske, "Tree hash exchange format (thex)," [online] [cit. 2022-05-17], Dostupné z: https://adc.sourceforge.io/draft-jchapweske-thex-02.html.
[5] M. Szydlo, "Merkle tree traversal in log space and time," [online] [cit. 2022-05-17], Dostupné z: https://link.springer.com/content/pdf/10.1007/978-3-540-24676-3_32.pdf.
[6] J. O'Connor, J.-P. Aumasson, S. Neves, and Z. Wilcox-O'Hearn, "Blake3," [online] [cit. 2022-05-17], Dostupné z: https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf.
[7] "Rust language," [online] [cit. 2022-05-17], Dostupné z: https://www.rust-lang.org/.
[8] "Pyo3," [online] [cit. 2022-05-17], Dostupné z: https://docs.rs/pyo3/latest/pyo3/.