

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Kryptografia v Pythone

Bakalárska práca

2022

Patrik Zeleňák

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Kryptografia v Pythone

Bakalárska práca

Študijný program:	Počítačové siete
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra elektroniky a multimediálnych telekomunikácií (KEMT)
Školiteľ:	prof. Ing. Miloš Drutarovský, CSc.

Košice 2022

Patrik Zeleňák

Abstrakt v SJ

Bakalárska práca sa zaoberá analýzou, testovaním a porovnaním vybraných existujúcich kryptografických knižníc pre programovací jazyk Python. Porovnanie knižníc je zamerané najmä z pohľadu rýchlosťi a rozsahu podpory kryptografických algoritmov a protokolov. Teoretická časť je venovaná opisu Merklovej hašovacej stromovej konštrukcie, výhodám a využitiu tejto konštrukcie. Samotná pozornosť je venovaná detailnému opisu novej vysoko-paralizovateľnej hašovacej funkcie Blake3. Praktická časť bakalárskej práce je rozdelená do dvoch častí. Prvá praktická časť je zameraná na analýzu dostupných kryptografických knižníc pre jazyk Python a prípravy demonštračných aplikácií pre porovnanie a analýzu daných knižníc. Druhá praktická časť je venovaná prepojeniu externej optimalizovanej hašovacej funkcie Blake3 napísanej v jazyku Rust s jazykom Python. Záver práce obsahuje výkonnostné porovanie prepojenej implementácie hašovacej funkcie Blake3 s jej inými implementáciami.

Klúčové slová v SJ

Kryptografia, Python, Blake3, PyO3, Dynamická Python knižnica

Abstrakt v AJ

This Bachelor thesis focuses on analysis, testing and comparison between existing cryptographic libraries for programming language Python. The comparison of cryptographic libraries is mainly focused on performance and range of supported cryptographic algorithms and protocols. Theoretical part of the paper discusses about Merkle hash tree structure, it's advantages and it's use in practice. A special attention is paid to detailed description of new highly parallelizable hash function Blake3. The practical part of the thesis is divided into two sections. The first section is focused on analysis of available cryptographic Python libraries and creation of demo application, needed for comparison of libraries. The second section focuses on binding of external Rust hash function Blake3 to Python. The conclusion of the thesis includes performance comparison of binded Blake3 implementation and some other implementations of hash function Blake3.

Klúčové slová v AJ

Cryptography, Python, Blake3, PyO3, Dynamic Python Library

Bibliografická citácia

ZELENÁK, Patrik. *Kryptografia v Pythone* . Košice: Technická univerzita v Košiciach, Fakulta elektrotechniky a informatiky, 2022. 80s. Vedúci práce: prof. Ing. Miloš Drutarovský, CSc.

TECHNICKÁ UNIVERZITA V KOŠICIACH
FAKULTA ELEKTROTECHNIKY A INFORMATIKY
Katedra elektroniky a multimediálnych telekomunikácií

**ZADANIE
BAKALÁRSKEJ PRÁCE**

Študijný odbor: **Informatika**
Študijný program: **Počítačové siete**

Názov práce:

Kryptografia v Pythonе
Cryptography in Python

Študent: **Patrik Zeleňák**
Školiteľ: **prof. Ing. Miloš Drutarovský, CSc.**
Školiace pracovisko: **Katedra elektroniky a multimediálnych telekomunikácií**
Konzultant práce:
Pracovisko konzultanta:

Pokyny na vypracovanie bakalárskej práce:

Analyzujte, otestujte a porovnajte vybrané existujúce kryptografické knižnice pre programovací jazyk Python. Porovnanie realizujte najmä z pohľadu rýchlosťi vykonávania kryptografických algoritmov a rozsahu podporovaných kryptografických algoritmov a protokolov. Zamerajte sa predovšetkým na moderné a perspektívne algoritmy, pre ktoré pripravte sady jednoduchých demonštračných aplikácií, ktoré budú demonštrovať ich využitie v Pythone. Na príklade hašovacej funkcie Blake 3 overte a demonštrujte možnosť využitia externých optimalizovaných modulov, vytvorených v inom programovacom jazyku, na zrýchlenie finálnej aplikácie vytvorennej v programovacom jazyku Python.

Jazyk, v ktorom sa práca vypracuje: slovenský
Termín pre odovzdanie práce: 27.05.2022
Dátum zadania bakalárskej práce: 29.10.2021



10. 
.....
prof. Ing. Liberios Vokorokos, PhD.
dekan fakulty

Čestné vyhlásenie

Čestne vyhlasujem, že som bakalársku prácu na tému: Kryptografia v Pythone vypracoval samostatne, pomocou uvedenej literatúry pod odborným vedením prof. Ing. Miloša Drutarovského, CSc.

Košice, 13.5.2022

.....

Vlastnoručný podpis

Podčakovanie

Rád by som sa podčakoval vedúcemu mojej bakalárskej práce, prof. Ing. Milošovi Drutarovskému, CSc. za metodické usmernenie, ochotu a odborné rady, ktoré mi poskytol pri vypracovaní bakalárskej práce.

Obsah

Zoznam skratiek	xvi
Zoznam Symbolov	xviii
1 Úvod do Pythonu	3
1.1 Python	3
1.1.1 Inštalácia Pythonu	4
1.1.2 PIP - inštalácia Python modulov	4
1.1.3 Import Python modulov	5
1.2 Binárna reprezentácia dát v jazyku Python	5
1.3 Spravne spracovavanie suborov pre kryptografiu	7
2 Kryptografické knižnice jazyka Python	11
2.1 Knižnica PyCryptodome	11
2.2 Knižnica Cryptography	12
2.3 Knižnica Hashlib	14
3 Vysoko paralizovateľná hašovacia funkcia Blake3	16

3.1	Merklova hašovacia stromová štruktúra	17
3.1.1	Rozdelenie vstupných dát na segmenty C_i	18
3.1.2	Hašovanie segmentov C_i	19
3.1.3	Tvorba a hašovanie rodičovských uzlov P	19
3.1.4	Hašovanie koreňa stromu	20
3.2	Paralizovateľná hašovacia funkcia Blake3	21
3.2.1	Kompresná funkcia $Kf()$	22
3.2.2	G funkcia	24
3.2.3	Paralelizmus s využitím SIMD inštrukcií	28
3.2.4	Permutácia $P()$	31
3.2.5	Finálna transformácia stavovej matice $Fu()$	31
3.2.6	Spracovanie uzlov a listov	32
3.2.7	Blake3 strom	36
4	Využitie paralizovateľnej Rust hašovacej funkcie Blake3 v jazyku Python	38
4.1	Prepájacie a mapovacie nástroje pre vytvorenie dynamickej Python knižnice	39
4.2	Inštalácia programovacieho jazyka Rust	40
4.3	Inicializácia vývojového prostredia a spustenie kódu	41
4.4	Inštalácia Rust modulov (crates)	42
4.5	Tvorba dynamickej Python knižnice z externého Rust modulu	43

4.6 Prepojenie Rust implementácie hašovacej funkcie Blake3 s jazykom Python	44
4.6.1 Inicializácia vývojového prostredia jazyka Rust	44
4.6.2 Import Rust modulov	45
4.6.3 Mapovenie Rust modulu s jazykom Python	46
4.6.4 Vytvorenie Python dynamického modulu	48
5 Experimentálne výsledky	53
5.1 Vyhodnotenie experimentálneho porovnania kryptografických knižíc jazyka Python	54
5.2 Experimentálne porovnanie viacerých implemetácií hašovacej funkcie Blake3	57
6 Záver	61
A Práca s binárnymi dátami	63
A.1 Základné operácie s bajtmi	63
A.2 Konverzia dátovych typov Bytes-Bytearray-List	65
B Inštalácia jazyka Python	67
B.1 Inštalácia jazyka Python pre OS Windows	67
B.2 Inštalácia a práca s viacerými verziami jazyka Python	69
B.2.1 Manuálna inštalácia	69
B.2.2 Inštalácia viacerých verzií jazyka Python pomocou nástroja pyenv	72

C Inštalácia jazyka Rust	73
Literatúra	76
Zoznam príloh	81

Zoznam obrázkov

3.1	Marklov hašovací strom	18
3.2	Hašovací strom Blake3	22
3.3	Bloková schéma kompresnej funkcie	23
3.4	G operácie	27
3.5	Princíp spracovania bloku dát M kompresnou funkciou	28
3.6	Paralelné spracovanie stĺpcov operáciami G_{0-3}	29
3.7	Rotácia vektorov a následné paralelné spracovanie stĺpcov operáciami G_{4-7}	29
3.8	Inverzná rotácia vektorov	30
3.9	Sekvenčné spracovanie listu binárneho stromu v hašovacej funkcií Blake3	36
3.10	Budovanie hašovacieho stromu	37
4.1	Výpis z terminálu 1.	41
4.2	Výpis z terminálu 2.	41
4.3	Bloková schéma mapovania	43
B.1	Oficiálna webová stránka jazyka Python	67

B.2	Výber požadovanej verzie jazyka Python	68
B.3	Proces inštalácie jazyka Python	68
B.4	Výpis nainštalovanej verzie Pythonu v príkazovom riadku	69
B.5	Inštalácia jazyka Python 2.7.18	70
C.1	Inštalačný balíček jazyka Rust dostupný	73
C.2	Terminálový výpis pri inštalácii jazyka Rust	74
C.3	Terminálový výpis chyby pri inštalácii jazyka Rust	74
C.4	Priebeh inštalácie softvéru Microsoft C++ build tool	75

Zoznam tabuliek

1.1	Módy pre prácu so súbormi v Pythone.	8
2.1	Feasible triples for a highly variable Grid	13
2.2	Porovnanie vybraných kryptografických modulov jazyka Python z hľadiska ich veľkosti	15
3.1	Premenné stavovej matice SM	26
3.2	Inicializačné vektory pre Blake3	26
3.3	Permutácia bloku dát $P(M)$, pričom $M = m_0, \dots, m_{15}$	31
3.4	Finálna transformácia $Fu()$	32
3.5	Tabuľka hodnôt premennej d	34
4.1	Verzie použitých programovacích jazykov a modulov	39
4.2	Premenovanie dynamického modulu v závislosti od operačného systému	51
5.1	Výsledky merania rychlosných testov, vykonaných na procesore I5-8250U a OS Windows 10, pre symetrické prúdové šifry	54
5.2	Výsledky merania rychlosných testov, vykonaných na procesore I5-8250U a OS Windows 10, pre symetrické blokové šifry	55

5.3	Výsledky merania rychlostných testov, vykonaných na procesore I5-8250U a OS Windows 10, pre digitálny podpis DSA a ECDSA	55
5.4	Výsledky merania rychlostných testov, vykonaných na procesore I5-8250U a OS Windows 10, pre RSA-OAEP s rôznou veľkosťou modula	56
5.5	Výsledky merania rychlostných testov,, vykonaných na procesore I5-8250U a OS Windows 10, pre hašovacie funkcie	56
5.6	Výsledky merania rychlostných testov pre, vykonaných na procesore I5-8250U a OS Windows 10, hašovacie funkcie	57
5.7	Rýchlosťny test hašovacej funkcie Blake3, vykonaných na procesore I5-8250U a OS Windows 10, pre rôzne implementácie funkcie Blake3	58
5.8	Porovnanie vybraných konvenčných hašovacich funkcií, podporovaných kryptografickým modulom Cryptography s prepojenou hašovaou funkciou Blake3 na procesore I5-8250U	59

Zoznam skratiek

3DES	Triple Data Encryption Algorithm.
AES	Advanced Encryption Standard.
ARC4	Alleged Rivest Cipher 4.
ARM	Advanced RISC Machine.
AVX-512	Advanced Vector Extensions 512.
AVX2	Advanced Vector Extensions 2.
CPU	Central Processor Unit.
CTR	Counter Mode.
DDR4	Double Data Rate 4.
DH	Diffie Hellman Algorithm.
DSA	Digital Signature Algorithm.
ECC	Elliptic Curve Cryptography.
ECDH	Elliptic-curve Diffie–Hellman.
ECDSA	Elliptic Curve Digital Signature Algorithm.
ECIES	Elliptic Curve Integrated Encryption Scheme.
GOST	Government Standard.
MD4	Message-digest algorithm 4.
MD5	Message-digest algorithm 5.
NIST	National Institute of Standards and Technology.
OS	Operačný systém.
PIP	Package Installer for Python.
RAM	Random Access Memory.
RSA	Rivest–Shamir–Adleman šifra.

- SHA** Secure Hash Algorithm.
- SHA2** Secure Hash Algorithm 2.
- SHA3** Secure Hash Algorithm 3.
- SIMD** Single instruction, multiple data.
- SSD** Solid State Drive.

Zoznam Symbolov

\parallel	zreťazenie objektov (concatenation)
C_i	Segment vstupných dát
CV_i	Zreťazovacia hodnota v binárnom strome
P	Rodičovský uzol
R	Koreň binárneho stromu
SM	Stavová matica
M	Blok vstupných dát
IV	Inicializačný vektor
Gk	G operácia, sfunkcie G()
v_{0-15}	Aliases stavovej matice
\oplus	Binárna operácia XOR
$+$	Sčítanie modulo 2^{32}
\ggg	Rotácia bitov vpravo
h_{0-7}	Vstupné zreťaovacie hodnoty stavovej matice
$T = t_0, t_1$	Počítadlo segmentov
b	Počítadlo bajtov v bloku
d	Bity slúžiace na bitovu separáciu domén

Úvod

Digitalizácia a dopad internetovej éry drasticky zmenil všetko vrátane bankovníctva, internetovej a sietovej bezpečnosti, elektronickej pošty, zábavy a dokonca aj vzdelávania. Kryptografia sa stala jedným z najdôležitejších nástrojov moderného sveta. Masívnym vývojom informačno-komunikačných technológií a elektronických systémov, spolu s dopytom po bezpečnom spracovaní a ukladaní dát, sa nároky na kryptografickú bezpečnosť zvýšili. Vzniklo veľké množstvo metód a kryptografických systémov, ktorými dokážeme naše dáta chrániť a uchovávať v bezpečí pred nepovolanými osobami. Jednou z týchto metód je šifrovanie. V tejto práci si bližšie opíšeme základne konverzné techniky a režimy práce so súbormi, ktoré sú potrebné pre prácu s kryptografickými modulmi jazyka Python. Podrobne sa zameriame na analýzu kryptografických modulov jazyka Python z hľadiska rýchlosťi vykonávania kryptografických úkonov ale aj z hľadiska rozsahu podporovaných kryptografických algoritmov a protokolov.

Využitie jazyka Python je pre kryptografické účely omnoho jednoduchšie ako napríklad C/C++. Hoci existujú populárne, voľne dostupné knižnice ako OpenSSL, ich používanie môže byť komplexné a náročné. Python eliminuje tieto problémy a disponuje mnohými knižnicami (modulmi) ako napríklad Cryptography, PyCryptodome, Hashlib a mnoho ďalších. Jednou z vlastností jazyka Python je jeho jednoduchá integrácia a prepojenie s jazykmi ako C/C++, Rust, Java, Fortran alebo .NET.

Nosnou časťou bakalárskej práce je detailný opis a analýza nového, rýchleho a vysoko paralizovateľného kryptografického stavebného bloku, ktorým je hašovacia funkcia Blake3. Funkcia Blake3 je zaujímavá najmä svojou štruktúrou, ktorá je založená na princípe Merklového stromu. Práve vďaka Merklovej stromovej ha-

šovacej konštrukcii je funkcia Blake3 vhodná pre rýchle hašovanie veľkých súborov s teoreticky nekonečnou možnosťou paralelizmu, či verifikáciu streamovaných dát. Optimalizovaná a plne paralizovateľná implementácia hašovacej funkcie Blake3 je napísaná v programovacom jazyku Rust. Využitie jednej z vlastností jazyka Python, možnosť prepojenia externého modulu napísaného v inom programovacom jazyku, demonštrujeme na príklade prepojenia externého Rust modulu Blake3 s jazykom Python.

Prvá kapitola obsahuje stručný úvod do programovacieho jazyka Python, čo zahŕňa opis jazyka Python, jeho inštaláciu a základnú prácu s bitovými dátovými typmi, či prácu so súbormi. Opísaný je princíp inštalácie a používania Python modulov pomocou Python inštalátora balíčkov (PIP).

Druhá kapitola je zameraná na opis vybraných kryptografických knižníc z hľadiska rozsahu podporovaných kryptografických algoritmov a protokolov. Opis zahŕňa výhody a nevýhody jednotlivých kryptografických knižníc ale aj princípy implementácie a štruktúru daných knižníc.

Detailným opisom hašovacej funkcie Blake3 ale aj opisom Merklovej stromovej štruktúry sa zaoberá tretia kapitola tejto bakalárskej práce. Táto kapitola opisuje výhody Merklovej hašovacej štruktúry, jej využitie a princíp budovania binárneho Merklového stromu. Súčasťou tejto kapitoly je detailný opis nosnej časti tejto práce, ktorou je hašovacia funkcia Blake3. Opísaný je vnútorný stav funkcie Blake3, princíp paralelizmu a využitia SIMD inštrukcií, či budovanie binárneho hašovacieho stromu funkcie Blake3.

Štvrtá kapitola zahŕňa princíp prepojenia externého modulu napísaného v programovacom jazyku Rust s jazykom Python. Samotné prepojenie je realizované na hašovacej funkcií Blake3, ktorej optimalizovaná a plne paralizovateľná implementácia je napísaná v jazyku Rust.

Posledná kapitola je venovaná experimentálnemu overeniu a porovnaniu kryptografických modulov jazyka Python. Porovnanie vybraných kryptografických modulov je realizované na základe rýchlosťi kryptografických algoritmov. Súčasťou tejto kapitoly je porovnanie rôznych implementácií hašovacej funkcie Blake3 spolu s prepojenou implementáciou funkcie Blake3 v jazyku Python.

1 Úvod do Pythonu

1.1 Python

Python je vysoko úrovňový, open-source programovací jazyk, podporovaný viacerými operačnými systémami. Python sa pokladá za procedurálny, funkcionálny ale aj objektovo orientovaný programovací jazyk, no môže byť použitý aj na rôzne scriptovacie účely [1]. Prvé implementácie programovacieho jazyka Python boli realizované v roku 1989 vývojárom Guidom van Rossumom¹ a prvá stabilná verzia jazyka Python (Python 1.0) bola predstavená v roku 1991. V roku 2000 bola predstavená verzia Python 2.0, ktorá obsahovala niekoľko vylepšení ako napríklad garbage collector s funkcionalitou zachytávania slučiek a podporu kódovacej sady Unicode. Verzia Python 3.0, ktorá bola predstavená v roku 2008 obsahovala veľké množstvo vstavaných knižníc a nových funkcionalít, ktoré verzia Python 2.0 neponúkala. Najväčšie rozdiely medzi verziou Python 2.0 a 3.0 predstavuje zmena výpisu na štandardný výstup, kde bol príkaz `print` nahradený funkciou `print()`, úprava vstavaných knižníc alebo zmena syntaxe. Aktuálna verzia jazyka Python je verzia Python 3.10.4. Garbage collector, ktorý je súčasťou Pythonu, zabezpečuje automatickú správu pamäte, čo sa môže javiť ako výhoda oproti štandardu pre kryptografiu, ktorým je jazyk C. Syntax jazyka Python je pomerne jednoduchá, čo nám umožňuje písanie kód rýchlo a prehľadne. Filozofia dizajnu jazyka Python kladie dôraz na čitateľnosť kódu s využitím odsadenia blokov kódu. Spúšťanie programu napísaného v Pythone si nevyžaduje špeciálny preklad a prítomnosť compilera. Program sa spracováva pomocou interpreta podobne ako v jazyku PERL alebo

¹https://en.wikipedia.org/wiki/Guido_van_Rossum

PHP. Na druhú stranu, Python má značné nedostatky z hľadiska spotreby pamäte [2][3]. Štandardne je Python pomalší ako jazyk C, čo predstavuje jednu z jeho nevýhod [4]. Využívanie viacerých jadier CPU môže byť obtiažné a problematické [5]. Tieto nevýhody je možné redukovať prepojením externých modulov napísaných v inom programovacom jazyku [6].

1.1.1 Inštalácia Pythonu

Pre väčšinu Linuxových distribúcii alebo macOS je Python vopred predinstalovaný. Rôzne verzie Python interpreta je možné stiahnuť (www.Python.org/ Python) a nainštalovať z oficiálneho zdroja. Inštalácia Python interpreta pre operačný systém Windows vyžaduje inštalačný balíček s príponou .msi. Výberom správnej verzie predídeme rôznym problémom s kompatibilitou modulov. V tejto práci budeme využívať Python verziu 3.8.10. Detailný popis inštalácie Python interpreta na iných platformách môžeme nájsť v dokumentácii² alebo v knihe Shannona W. Braya [7] resp. v prílohe **B**. Overiť správnosť inštalácie je možné príkazom `Python --version` v príkazovom riadku alebo termináli, ktorý vypíše aktuálne nainštalovanú verziu Pythonu.

1.1.2 PIP - inštalácia Python modulov

PIP³ (package installer for Python) je inštalátor Python balíčkov, ktorý budeme využívať na inštaláciu rôznych kryptografických modulov. PIP je súčasťou inštalačného balíčka Python interpreta. V prípade, že PIP nie je súčasťou inštalačného balíčka, je možné ho explicitne stiahnuť (<https://bootstrap.pypa.io/get-pip.py>) a doinštalovať príkazom `py get-pip.py`. Inštalácia balíčkov prebieha v príkazovom riadku alebo termináli príkazom `pip install nazov-balíčka`.

²<https://wiki.Python.org/moin/BeginnersGuide/Download>

³<https://pip.pypa.io/en/stable/>

1.1.3 Import Python modulov

Jedným z cieľov tejto práce je analýzovať a otestovať kryptografické moduly jazyka Python a porovnať jednotlivé moduly na základe rýchlosťi a rozsahu podporovaných kryptografických algoritmov a protokolov. Je preto potrebné uviesť niekoľko spôsobov, ktorými dokážeme tieto moduly importovať do Python kódu.

Zdrojový kód 1.1: Štandardný import modulu random v jazyku Python

```
import random
#Volanie funkcie
random.urandom(16)
```

Zdrojový kód 1.2: Import modulu s využitím aliasu v jazyku Python

```
import random as r
#Volanie funkcie
r.randint(1,10)
```

Zdrojový kód 1.3: Import konkrétnej triedy funkcie v jazyku Python

```
from random import randint
#Volanie funkcie
randint(1,10)
```

Obrovskou výhodou jazyka Python je možnosť interakcie s externými modulmi napísanými v inom programovacom jazyku [6]. Zaujímavosťou je, že niektoré vstavené funkcie jazyka Python sú navrhnuté v programovacom jazyku C [8]. Extérne moduly sú využívané najmä z dôvodu eliminácie slabých stránok jazyka Python. Súčasťou tejto práce je vytvorenie externého modulu, napísaného v programovacom jazyku Rust a jeho prepojenie s Pythonom. Viac o integrácii externých modulov nájdeme v kap. 4.

1.2 Binárna reprezentácia dát v jazyku Python

V kryptografii sú dátá typicky spracovávané vo forme bajtov, čomu sú prispôsobené aj rôzne kryptografické nástroje. Je preto dôležité dbať na správnu

reprezentáciu dát a prácu s nimi. Python poskytuje niekoľko spôsobov, ktorými je možné vyjadriť dátu v binárnom tvare. Prvým spôsobom je tzv. bytes-literals. Ide o binárnu reprezentáciu dát, ktorej syntax je podobná refazcu (string), avšak od bežného refazca sa líši prefixom b. Bytes-literals povoľuje iba základných 127 ASCII znakov. Všetky ostatné znaky je nutné spracovať iným spôsobom.

```
Bytes_literals = b"Reprezentacia bajtov vo forme bytes-literals."
```

Pri pokuse o konverziu refazca na bytes-literals napr. b"Toto je skúška!" nám Python interpreter vypíše chybové hlásenie, pretože tento výraz obsahuje znaky, ktoré nie sú z podporovaného rozsahu ASCII znakov. Ide o znaky "ú" alebo "š". V takomto prípade je nutné použiť metódu encode(), ktorú nám ponuka jazyk Python. Metóda encode() podporuje rôzne kódovacie sady. Predvolenou kódovacou sadou je UTF-8. Analogicky je možné využívať metódu decode() na dekódovanie bajtov.

```
Bytes_encoded = "Reprezentacia bajtov s využitím funkcie encode!".encode()
```

V praxi sa môžeme stretnúť s formátom bytes-arrays, ktorý reprezentuje list bajtov. Tento formát môže byť v niektorých kryptografických API preferovaný, pretože poskytuje väčšiu flexibilitu práce s bajtmi [9]. Bytes-arrays je základným binárnym typom v jazyku Python. V tomto bloku si opíšeme základne techniky konverzie bajtov, listov a bytes-arrays. Viac informácií o dátových typoch a vstavaných funkciách je možné nájsť v dokumentácii jazyka Python[8].

Pre generovanie a kopírovanie bajtov je vhodná vstavaná funkcia bytes(), ktorej vstupným parametrom môže byť celé číslo, binárny objekt alebo aj funkcia range().

Zdrojový kód 1.4: Práca s bajtmi

```
Bytes_zero = bytes(10)
Duplicated_bytes = bytes(Bytes_zero)
```

V Zdrojovom kóde 1.4 môžeme vidieť demonštráciu funkcie `bytes()`, kde sme najprv vygenerovali 10 nulových bajtov a neskôr sme tieto bajty skopírovali a priradili premennej s názvom `Duplicated_bytes`.

Veľmi často sa môžeme stretnúť s hexadecimálnou reprezentáciou reťazca. Ide o reťazec, ktorý obsahuje hexadecimálne vyjadrenie dát, vždy po dvojiciach znakov pričom každý znak môže nadobúdať hodnoty od 0 po f napr. reťazec `"2ef0 f1f2"`. Takýto hexa-string je možné previesť do bajtov a naopak.

```
Hex_to_bytes = bytes.fromhex("2ef0 f1f2")
Bytes_to_hex = b"\x2e\xf0\xf1\xf2".hex()
```

Jedným z možných formátov pre uchovávanie bajtov je uloženie bajtov do listu, kde je každý bajt vyjadrený ako celé číslo. List v takomto tvare je možné previesť do tvaru bytes-literal pomocou funkcie `bytes()` alebo do bytes-arrays pomocou `bytearray()`. Spätná konverzia bajtov na list je možná prostredníctvom funkcie `list()`.

```
Test_list = [116, 101, 115, 116]
Bytes_from_list = bytes(Test_list)
Test_bytes = b"test"
List_from_bytes = list(Test_bytes)
```

Práca s dátovým typom bytes-array je podobná ako s dátovým typom bytes. Vyžíva rovnaké funkcie na konverziu medzi inými dátovými typmi. Dátový typ bytes-array je tzv. mutable, čo znamená, že je možné meniť hodnotu jeho obsahu. Na druhej strane dátový typ bytes je immutable. Viac demonštračných príkladov a detailnejšie opisanie jednotlivých konverzných metód môžeme nájsť v prílohe **A** alebo v Python dokumentácii[8].

1.3 Spravne spracovavanie suborov pre kryptografiu

Ako sme už naznačili, spracovávanie dát v binárnom tvare je pre kryptografické systémy kľúčové, preto si v tomto bloku opíšeme niekoľko spôsobov spracovania súborov. Otvoriť a čítať súbory je v Pythone možné viacerými spôsobmi, no

najčastejšie sa využíva vstavaná funkcia `open()`. Táto funkcia obsahuje niekoľko vstupných parametrov ako `file`, `mode`, `encoding` a podobne [8]. `File` určuje názov, cestu alebo celočíselnú hodnotu deskriptora daného súboru. `Mode` predstavuje mód, v ktorom daný súbor otvárame. V prípade, že potrebujeme pracovať so súborom v binárnom tvaru, nastavíme túto hodnotu na "b". Predvolená hodnota argumentu `mode` "r" je nastavená na čítanie súboru v textovom režime. V Tabuľke 1.1 môžeme vidieť ďalšie podporované režimy. Jednotlivé režimy je možné kombinovať napr. "rb", "wr", "w+" a pod. [10].

Tabuľka 1.1: Módy pre prácu so súbormi v Pythone.

Režimy otvárania súborov.

Režim	Popis
r	režim pre čítanie, textový súbor
w	režim pre zápis, textový súbor; v prípade, že existuje rovnomený súbor, tak bude prepísaný
a	režim pre zápis na koniec súboru, textový súbor
r+	režim pre zápis na koniec súboru, textový súbor
w+	režim pre čítanie, zápis a prípadne prepisovanie v prípade rovnomeného súboru, textový súbor
rb	režim pre čítanie a zápis na konci súboru, binárny súbor
wb	režim pre zápis, v prípade rovnomeného súboru aj prepisovanie, binárny súbor
ab	režim pre zápis na koniec súboru, binárny súbor
rb+	režim pre čítanie a zápis, binárny súbor
wb+	režim pre zápis, čítanie a prípadne prepisovanie pri existencii rovnomeného súboru, binárny súbor
ab+	režim na zápis a čítanie na konci súboru, binárny súbor.

Zdrojový kód 1.5 demonštruje otvorenie súboru s názvom "`filename.txt`" v móde "rb", teda čítanie súboru v binárnom móde. V takomto prípade je nutné operáciu

ukončiť metódou `close()`, ktorá uvoľní dátu z RAM pamäte a správne zatvorí súbor. Metódou `read()` je možné získať obsah daného súboru. Po uzavorení súboru už nie je možné sa na súbor späťne odkazovať.

Zdrojový kód 1.5: Štandardná práca so súborom v jazyku Python

```
my_file = open("filename.txt", "rb")
data_from_file = my_file.read()
my_file.close()
```

Ďalším spôsobom je čítanie súboru pomocou klúčového slovíčka `with`, kde nie je potrebné súbor uzavárať. `with` sa postará o správne zatvorenie súboru a jeho uvoľnenie z pamäte. Odkazovanie sa na súbor je možné iba v bloku `with`.

Zdrojový kód 1.6: Práca so súborom s využitím príkazu `with` v jazyku Python

```
with open("filename.txt", "rb") as my_file:
    data_from_file = my_file.read()
```

Metóda `read()` obsahuje argument `size`, ktorý vyjadruje koľko bajtov súboru chceme prečítať. Hodnota argumentu `size` je predvolene nastavená na hodnotu `-1`, čo znamená prečítanie celého súboru. Je nutné podotknúť, že metóda `read()` vrácia dátu, či už v textovej forme alebo vo forme bajtov. Tieto dátu štandardne priradzujeme premennej, čo v preklade znamená, že dátu zapisujeme do operačnej pamäte. V prípade, že pracujeme s veľkým súborom (väčším ako je dostupná operačná pamäť), program nám môže „spadnúť“ a Python interpreter vypíše chybové hlásenie. Jedným z možných riešení je nastaviť hodnotu argumentu `size` na fixnú hodnotu a následne dátu spracovávať po blokoch s danou veľkosťou. Pri viacnásobnom volaní funkcie `read()` tak postupne prečítame celý obsah súboru. Zdrojový kód 1.7 demonštruje jednu z metód spracovania a hašovania veľkého súboru s využitím hašovacej funkcie MD5 [11].

Zdrojový kód 1.7: Čítanie veľkého súboru v Pythone

```
import hashlib
md5_object = hashlib.md5()
block_size = 128*md5_object.block_size
a_file = open("large_file", 'rb')

chunk = a_file.read(block_size)
```

```
while chunk:  
    md5_object.update(chunk)  
    chunk = a_file.read(block_size)  
  
md5_hash = md5_object.hexdigest()
```

2 Kryptografické knižnice jazyka Python

Kryptografia je veda, ktorá skúma matematické metódy utajovania obsahu a preukázateľnosti pôvodu prenášaných správ [12]. Jej cieľom je utajíť obsah prenášaných správ, čo zahŕňa riešenie základných bezpečnostných otázok ako je dôvernosť a integrita dát, či autentizácia a autorizácia užívateľov. Bezpečný prenos a utajenie prenášaného obsahu je možné dosiahnuť s využitím kryptografických algoritmov a protokolov (digitálne podpisy, výmena kľúčov, autentizácia správ, a pod.), ktorých základ tvoria kryptografické primitíva. Cieľom tejto kapitoly je analyzovať, otestovať a porovnať vybrané existujúce kryptografické knižnice (moduly) pre programovací jazyk Python, pričom porovnanie jednotlivých kryptografických modulov bude zamerané na spôsob implementácie, veľkosť modulov a rozsah podporovaných kryptografických algoritmov a protokolov.

2.1 Knižnica PyCryptodome

PyCryptodome (pôvodne známy ako PyCrypto) je kryptografický modul jazyka Python, obsahujúci nízko úrovňové kryptografické primitíva ako napríklad AES, Salsa20, RSA, DSA, SHA3, Blake2s, Blake2b, ElGamal, HMAC a mnoho ďalších. Súčasťou modulu PyCryptodome je aj algoritmus pre generovanie kľúčov na báze eliptických kriviek (NIST krivky P-192, P-224, P-256, P-384 a P-521). Hoci modul ponúka generovanie kľúčov na báze ECC, samotný algoritmus pre šifrovanie na báze ECC v module chýba. Úplný zoznam podporovaných kryptografických

primitív a protokolov je dostupný v dokumentácii¹. PyCryptodome nie je prepojovacím obrazom žiadnej kryptografickej knižnice jazyka C, ako je OpenSSL. Algoritmy sú v maximálnej možnej miere implementované v čistom Pythone. Iba časti, ktoré sú mimoriadne dôležité pre výkon (napr. blokové šifry), sú implementované ako rozšírenia jazyka C, teda ich implementácia je založená na možnosti externého prepojenia jazyka C s jazykom Python. V budúcich verziach modulu PyCryptodome môžeme očakávať pridanie kryptografických algoritmov ako ECIES, ECDH, Camellia, GOST, Diffie-Hellman, bcrypt, argon2. Viac informácií ohľadom modulu PyCryptodome je možné nájsť v dokumentácii².

2.2 Knižnica Cryptography

Cryptography je komplexný vysoko úrovňový (Fernet) ale aj nízko úrovňový (Hazmat) kryptografický modul jazyka Python, ktorý ponúka veľké množstvo kryptografických primitív a protokolov. Oproti modulu PyCryptodome je modul Cryptography masívnejší, komplexnejší a jeho implemenácia závisí od knižnice OpenSSL (napísanej v jazyku C) pre všetky kryptografické operácie. OpenSSL je de facto štandard pre kryptografické knižnice a poskytuje vysoký výkon spolu s rôznymi certifikátmi, ktoré môžu byť relevantné pre vývojárov. Podrobny opis všetkých podporovaných kryptografických algoritmov a protokolov je možné nájsť v online dokumentácii³. Tento modul obsahuje základné algoritmy pre ECC ako napríklad ECDSA či ECDH. Oproti modulu PyCryptodome ponúka modul Cryptography väčšiu podporu eliptických kriviek⁴. Tvorcovia tohto modulu kladú pri vývoji veľký dôraz na bezpečnú implementáciu všetkých kryptografických algoritmov a protokolov.

Tabuľka 2.1 poskytuje stručný prehľad základných kryptografických primitív a algoritmov, ktoré sú podporované kvalitnými kryptografickými Python modulmi PyCryptodome a Cryptography. Okrem týchto algoritmov, môžeme v oboch kryptogra-

¹<https://www.pycryptodome.org/en/latest/src/features.html>

²<https://www.pycryptodome.org/en/latest/src/installation.html>

³<https://cryptography.io/en/latest/hazmat/primitives/>

⁴<https://cryptography.io/en/latest/hazmat/primitives/asymmetric/ec/>

fických moduloch nájst protokoly ako Secret sharing scheme [13], alebo funkcie pre odvodzovanie kľúčov (key derivation functions) [14]. Z Tabuľky 2.1 alebo aj z dokumentácií daných kryptografických modulov je vidieť, že modul Cryptography je z hľadiska rozsahu podporovaných kryptografických primitív a algoritmov rozsiahlejší a robustnejší. Z pohľadu bezpečnej implementácie algoritmov je modul Cryptography aj odolnejší voči postranným útokom (side-channel attacks)[15].

Tabuľka 2.1: Stručný zoznam základných kryptografických primitív a algoritmov, podporovaných Python modulmi PyCryptodome a Cryptography

Podporované kryptografické algoritmy pre vybrané Python moduly			
Kategória	Kryptografické primitívum	PyCryptodome	Cryptography
Symetrické šifry	AES	✓	✓
	DES/3DES	✓	✓
	CAST-128	✓	✓
	RC2	✓	✗
	Camellia	✗	✓
	ChaCha20	✓	✓
	Salsa20	✓	✗
	RC4	✓	✓
	SEED	✗	✓
	SM4	✗	✓
Tradičné módy symetrických šifier	Blowfish	✗	✓
	IDEA	✗	✓
	ECB	✓	✓
	CBC	✓	✓
	CFB	✓	✓
Špeciálne módy symetrických šifier a autentizačné šifrovanie	OFB	✓	✓
	CTR	✓	✓ - iba pre AES
	OpenPGP	✓	✗
	CCM	✓	✓
	EAX	✓	✗
Kryptografia s verejným kľúčom	SIV	✓	✓
	OCB	✓	✓
	GCM	✓	✓
	ChaCha20-Poly1305	✓	✓
	XTS	✗	✓
	RSA(OAEP,PSS,PKCS1v15)	✓	✓
	DSA	✓	✓
	ECDSA	✓	✓
	ECDH	✗	✓
	ElGamal	✓	✗
	Diffie-Helman	✗	✓
	SHA1	✓	✓
	SHA2	✓	✓

Tabuľka 2.1 – pokračovanie

	SHA3	✓	✓
	Blake2	✓	✓
	MD5	✓	✓
	SM3	✗	✓
	KangarooTwelve	✓	✗
	RIPEMD-160	✓	✗
MAC	HMAC	✓	✓
	CMAC	✓	✓
	KMAC	✓	✗
	Poly1305	✓	✓

2.3 Knižnica Hashlib

Hashlib je hašovací Python modul, ktorý je súčasťou inštalácie jazyka Python. Teda modul Hashlib je aktualizovaný a udržiavaný samotnými vývojármi jazyka Python. Tento modul ponúka základné hašovacie funkcie ako napríklad MD4, MD5, funkcie z rodiny SHA, funkciu Blake2s, Blake2b a pod. Implementácia niektorých hašovacích funkcií ako napríklad funkcie z rodiny SHA3 alebo SHAKE sú založené na kryptografickej knižnici OpenSSL. Výhodou tejto knižnice je jednoduchá manipulácia a stručná dokumentácia⁵.

Za zmienku stoja aj špecializované moduly PyNaCl a tinyec. Modul PyNaCl je kryptografický modul zameraný predovšetkým na oblasť sieťových protokolov a algoritmov (<https://pypi.org/project/PyNaCl/>). Ide o profesionálny rozširovací modul kryptografickej knižnice libsodium vytvorennej Danielom J. Bernsteinstom⁶. Modul tinyec je relatívne malý kryptografický modul vytvorený na vykonávanie aritmetických operácií na eliptických krivkách. Hoci modul tinyec nie je vhodný pre produkčnú výrobu, slúži ako výborný edukačný nástroj pre hlbšie pochopenie vnútorného fungovania eliptických kriviek (<https://github.com/alexmgr/tinyec>).

Porovnanie vybraných kryptografických modulov možno vykonať z hla-

⁵<https://docs.python.org/3/library/hashlib.html>

⁶<https://cr.yp.to/djb.html>

diska veľkosti daných modulov. Tabuľka 2.2 demonštruje porovnanie veľkosti kryptografických modulov jazyka Python. Veľkosti vybraných kryptografických modulov z Tabuľky 2.2 zodpovedajú ich aktuálne najnovším verziám. Kryptografický modul Hashlib sa v Tabuľke 2.2 nenachádza, pretože tento modul je súčasťou jazyka Python a jeho veľkosť závisí nie len od implementácie kryptografických primitív, ale aj od samotnej štruktúry jazyka Python. Približná veľkosť implementácie kryptografických primitív modulu Hashlib pre Python 3.8 je 1,2 MB, čo zahŕňa aj kryptografické primitíva, ktorých implementácia je založená na prepojení kryptografických primitív z kryptografickej knižnice jazyka C, OpenSSL 1.1.0. Kryptografické moduly PyCryptodome, Cryptography a Hashlib boli experimentálne otestované z hľadiská rýchlosťi vykonávania kryptografických algoritmov. Výsledky rýchlostných meraní je možné pozorovať v kapitole 5.1.

Tabuľka 2.2: Porovnanie vybraných kryptografických modulov jazyka Python z hľadiska ich veľkosti

Porovnanie veľkosti vybraných kryptografických modulov		
Python Modul	Verzia modulu	Veľkosť
Cryptography	37.0.1	74,1 MB
PyCryptodome	3.14.1	39,5 MB
PyNaCl	1.5.0	28,6 MB
tinyec	0.4.0	64,4 kB

Zvyšná časť tejto bakalárskej práce je venovaná novému kryptografickému stavebnému bloku Blake3. Blake3 je perspektívna vysoko paralizovateľná hašovacia funkcia, postavená na základoch hašovacej funkcie Bao [16] a Blake2 [17], ktorá je zaujímavá z hľadiska extrémne rýchleho spracovania veľkého množstva dát. Blake3 sa čoraz častejšie využíva v blockchainových technológiach, peer-to-peer protokoloch a kryptomenách ako napríklad Solana a Suacoin [18][19]. V závere tejto práce demonštrujeme akým spôsobom možno využiť hašovaciu funkciu Blake3, ktorej optimalizovaná a paralizovateľná implementácia je napísana v programovacom jazyku Rust, pre efektívne odhašovanie veľkého množstva dát v jazyku Python.

3 Vysoko paralizovateľná hašovacia funkcia Blake3

Hašovacie funkcie majú obrovský význam v oblasti autentizácie a autorizácie dát. Jedným z možných využití hašovacích funkcií je overenie integrity dát t.j. garancia, že dátá neboli zmenené napríklad počas prenosu cez prenosový kanál. Predstavme si, že odosielateľ A a adresát B spolu nadviazali komunikáciu a chcú si medzi sebou vymeniť dátá. Odosielateľ A dátá zahašuje pomocou jednej z dostupných hašovacích funkcií. Výsledkom hašovania dát je hašovací kód s fixnou dĺžkou napr. 256 bitov. Takýto hašovací kód budeme nazývať referenčný hašovací kód. Referenčný kód je známy pre adresáta B . Odosielateľ A odošle dátá cez prenosový kanál adresátovi B . Overenie integrity dát sa vykoná na strane adresáta B a to tak, že adresát B zahašuje prijate dátá a porovná svoj hašovací kód s referenčným hašovacím kódom. Ak sa tieto kódy zhodujú, prenos dát prebehol úspešne a bez akejkoľvek zmeny dát počas ich prenosu. Takéto overenie integrity sa zväčša realizuje sekvenčným spôsobom, čo môže predstavovať isté nevýhody.

Konvenčné hašovacie funkcie spracúvajú dátá sekvenčným spôsobom, to znamená, že sa vstupné dátá typicky rozdеля na bloky dát o určitej veľkosti a tie sa spracúvajú s istou vzájomnou závislosťou. Každý z blokov je spracovaný kompresiou funkciou. Výstup kompresnej funkcie sa stáva vstupom pre následujúcu kompresnú funkciu. To znamená, že pre spracovanie i-teho bloku, musíme zahašovať všetky predchadzajúce bloky dát. Hašovací kód, teda výstup z hašovacej funkcie, dostávame po aplikácii kompresnej funkcie na posledný blok dát [20].

Hašovacia funkcia Blake3, ktorej je venovaná táto kapitola, však pracuje

úplne inak. Blake3 je moderná hašovacia funkcia vyvinutá v roku 2020 skupinou vývojárov¹ za účelom čo najefektívnejšie a najrýchlejšie zahašovať dátu a overiť ich integritu. Blake3 nevyužíva sekvenčné spracovanie dát ako to je u konvenčných hašovacích funkcií ale svoju štruktúru buduje na základoch Merklového stromu [21]. Merklova stromová konštrukcia prináša niekoľko výhod ako napríklad neobmedzenú možnosť paraleлизmu [22], overovanie integrity streamovaných dát, či overenie integrity dát pri ich sťahovaní z viacerých zdrojov simultánne [23]. Rýchlosť hašovacej funkcie Blake3 značne ovplyvňuje jej štruktúra. Vďaka Merklovým stromom je Blake3 vysoko paralelizovateľný, každé vlákno CPU jednotky môže spracovať svoj blok dát nezávisle, a špeciálne navrhnutý pre podporu SIMD inštrukcií, čo má za následok zvýšenie rýchlosťi. Autori Blaku3 dbali aj na podporu efektívnych výpočtov realizovaných na rôznych menších 32-bitových platformách ale aj bežných 64-bitových platformách. Teda efektívnosť spracovania dát je zachovaná bez ohľadu na typ architektúry. Funkcia Blake3, kvôli jej schopnosti rýchleho hašovania dát, nie je vhodná pre uchovávanie hesiel v databázových systémoch práve.

3.1 Merklova hašovacia stromová štruktúra

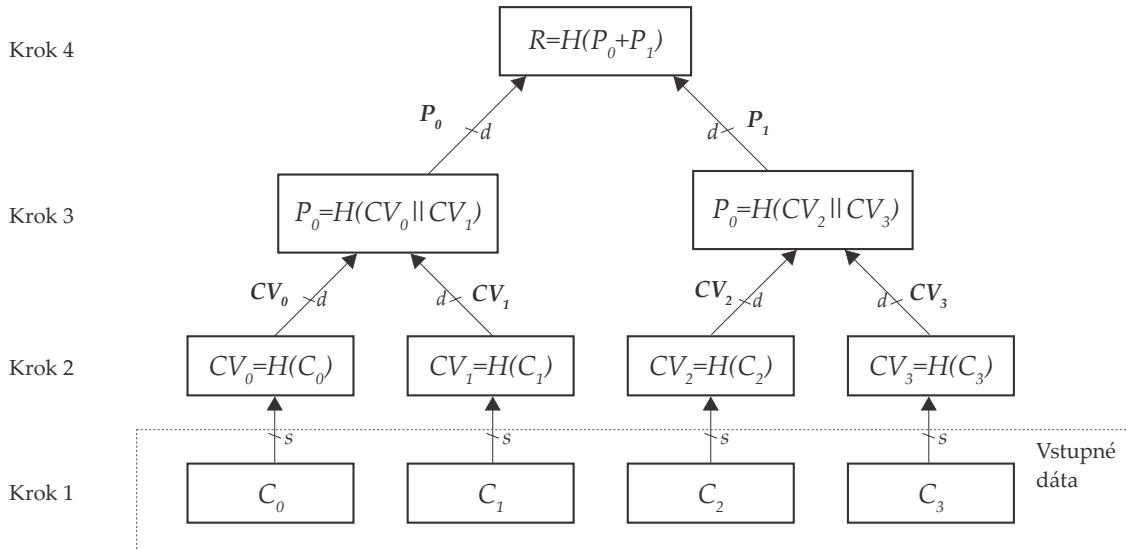
Merklov hašovaci strom je hašovacia konštrukcia vynájdená Ralphom Merklom[24], ktorá vykazuje výhodne vlastnosti pre efektívny paralelizovateľný výpočet integrity dát [25]. Ide o binárnu stromovú štruktúru, ktorej listy reprezentujú bloky vstupných dát C_i s veľkosťou s a každý rodičovský vrchol je odvodený od jeho dvoch potomkov. Na Obr. 3.1 môžeme vidieť príklad realizácie Merklového hašovacieho stromu.

Vo všeobecnosti môžeme realizáciu Merklového stromu popísať v niekolkých krokoch:

1. Rozdelenie vstupných dát do blokov C_i .
2. Hašovanie blokov dát C_i .

¹Tvorcovia Blake3: Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves a Zooko Wilcox-O'Hearn

3. Tvorba a hašovanie rodičovských uzlov P .
4. Hašovanie koreňa stromu R .



Obr. 3.1: Merklova stromova hašovacia konštrukcia

3.1.1 Rozdelenie vstupných dát na segmenty C_i

Vstupné dátá rozdelíme do blokov dát C_i o veľkosti s . Tieto bloky dát budeme označovať ako segmenty. V niektorých iných literatúrach môžu byť segmenty označované aj ako chunky. Z Obr. 3.1 je vidno, že vstupné dátá sú rozdelené do štyroch rovnako veľkých blokov C_0, C_1, C_2, C_3 . Veľkosť týchto blokov resp. segmentov sa môže v každej implementácii Merklovej hašovacej konštrukcie lísiť. Existujú určité hraničné prípady, kedy môže mať zlý výber veľkosti segmentu negatívne vlastnosti. J.Chapweske vo svojej práci [26] uvádzá, že z hľadiska efektívneho spracovania dát je veľmi dôležité voliť vhodnú veľkosť segmentu C_i .

Ak je veľkosť segmentu väčšia alebo rovná veľkosti vstupných dát, hašovacia štruktúra stráca svoje výhody stromovej štruktúry a proces hašovania je rovnaký ako pri konvenčnom hašovaní celého súboru. Teda ak veľkosť vstupných dát n je menšia alebo rovná veľkosti segmentu s , spracovávali by sme iba jeden blok dát a to konvenčným spôsobom, ktorý je naznačený v úvode tejto kapitoly.

Ak je veľkosť segmentu rovnaká ako veľkosť výstupu z hašovacej funkcie $H()$, teda kód hašovacej funkcie, ktorá spracováva dané segmenty vidľ. kap. 3.1.2, celková veľkosť hašovaných dát by mohla byť rovná až dvojnásobku veľkosti vstupu, pretože je potrebné hašovať nie len listy, ale aj príslušné rodičovské vrcholy.

Pre optimalizáciu Merklovej hašovacej stromovej štruktúry možno odvodiť záver, že veľkosť segmentu s by mala splňať základne požiadavky vyjadrené vzťahom 3.1

$$n > s > d. \quad (3.1)$$

Pričom n je veľkosť bloku vstupných dát, d predstavuje veľkosť hašovacieho kódu použitej hašovacej funkcie a s je veľkosť segmentu. Odporučaná veľkosť segmentu pre viaceré aplikáčné použitia, vrátane hašovacej funkcie Blake3, je 1024 bajtov.

3.1.2 Hašovanie segmentov C_i

V podkapitole 3.1.1 sme si opísali požiadavky potrebné pre optimálne rozdelenie vstupných dát do segmentov. Druhým krokom je spracovanie týchto segmentov C_i . Na každý segment sa aplikuje hašovacia funkcia $H()$. V praxi sa volí kvalitná a bezpečná hašovacia funkcia napr. niektorá z rodiny SHA-2. Hašovanie bloku prebehne klasickým spôsobom a výstupom z hašovacej funkcie $H()$ je zretežovacia hodnota v binárnom strome CV_i (ďalej len zretežovacia hodnota). Z Obr. 3.1 je zrejmé, že segment tvorí vstup pre hašovaciu funkciu $H()$. Teda segment C_0 je vstupom pre funkciu $H()$ a výstupom tejto funkcie je zretežovacia hodnota CV_0 .

3.1.3 Tvorba a hašovanie rodičovských uzlov P

V druhom kroku sme si opísali princíp hašovania segmentov pomocou hašovacej funkcie $H()$, čo viedlo k získaniu zretežovacích hodnôt. V tejto podkapitole si povieme akým spôsobom vznikajú rodičovské uzly a ako ich spracúvame. Každý rodičovský uzol má práve dvoch potomkov a to ľavého potomka a pravého potomka.

Teda každý rodičovský uzol pozostáva práve z dvoch hodnôt a to zo zratazovacej hodnoty ľavého potomka a zratazovacej hodnoty pravého potomka. Na Obr. 3.1 je vidieť, že rodičovský uzol P_0 pozostáva z jeho ľavého potomka C_0 a pravého potomka C_1 , konkrétnie z ich zreťazovacích hodnôt CV_0 a CV_1 . Hašovaním rodičovského uzla získavame opäť zreťazovaciu 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.

3.1.4 Hašovanie koreňa stromu

Vo všeobecnosti sa hašovanie koreňa stromu realizuje rovnakým spôsobom ako hašovanie rodičovských uzlov. Na Obr. 3.1 môžeme vidieť, že koreň stromu R má dvoch potomkov P_0 a P_1 . Zreťazovacie hodnoty potomkov koreňa tvoria vstup pre hašovaciu funkciu $H()$ a výstupom funkcie $H()$ získavame hašovací kód Merklovej hašovacej štruktúry. Teda po aplikácii $H(P_0 \parallel P_1)$ získavame hašovací kód stromovej štruktúry.

Stromová hašovacia štruktúra poskytuje možnosť paralelného spracovania dát ale aj rôzne iné výhody ako napríklad overenie integrity konkrétneho bloku dát, čo nie je možné pri konvenčnej metóde hašovania, práve z dôvodu sekvenčného spracovávania blokov. Pri stromovej hašovacej konštrukcii, je možné overiť integritu každej vetvy stromu, za predpokladu, že sú splnené určité požiadavky [26]. Teda je možné overiť integritu aktuálne získaných parciálnych dát, hoci strom ešte nie je plne dostupný. Táto vlastnosť poskytuje možnosti overovania integrity streamovaných súborov, či overovania integrity súborov pri ich stažovaní z viacerých zdrojov simultánne [23][27].

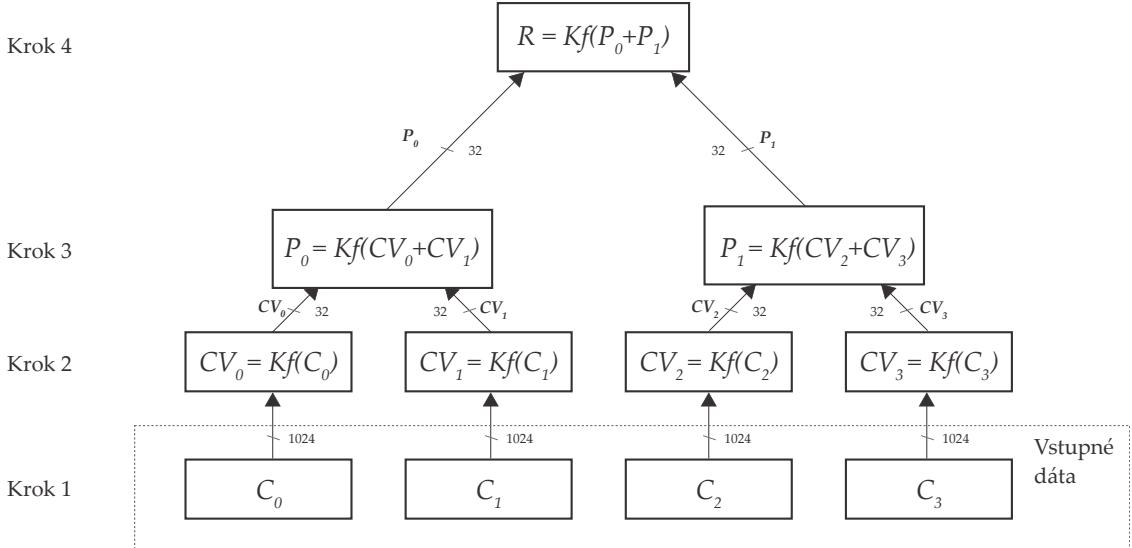
Merklové stromy tvoria jednu zo základných časti block-chainových technológií [28][29] a ich využitie možno nájsť u najväčších zastupiteľov kryptomien ako Bitcoin alebo Ethereum [30].

3.2 Paralizovateľná hašovacia funkcia Blake3

Blake3 využíva Merklovú hašovaciu konštrukciu, čo je jedným z hlavných dôvodov rýchlosťi a efektívnosti tejto hašovacej funkcie. Zvyšok tejto kapitoly bude venovaný podrobnému opisu funkcie Blake3 spolu s jej hašovacou stromovou štruktúrou.

Algoritmus hašovacej funkcie Blake3 spracúva údaje po blokoch s dĺžkou 1024 bajtov a produkuje hašovací kód s dĺžkou 64 bajtov, pričom je možné tento hašovací kód rozšíriť na ľubovoľnú veľkosť z rozsahu $0 \leq l < 2^{64}$. Maximálna veľkosť vstupných údajov je 2^{64} . Podobne ako bolo opísané v podkapitole 3.1 sa vstupné dátá rozdelia na segmenty C_i o veľkosti 1024 bajtov. Posledný segment môže byť aj menší no nikdy nie prázdný. V prípade, že je posledný segment menší bude mu pridaná vhodná výplňová schéma (padding). Tieto segmenty C_i tvoria listy binárneho hašovacieho stromu. Každý segment dát sa hašuje kompresnou funkciou $Kf()$, ktorej výsledkom je zretezovacia hodnota CV_i o veľkosti 32 bajtov. Princíp tvorby rodičovských uzlov je rovnaký ako v Merklovej stromovej štruktúre (kap. 3.1), teda každý rodičovský uzol je odvodený od jeho dvoch potomkov (pravý a ľavý potomok). Na ilustračnom Obr. 3.2 môžeme vidieť hierarchické spracovanie dát, kde rodičovský uzol P_0 je tvorený zretezovacou hodnotou ľavého potomka CV_0 a zretezovacou hodnotou pravého potomka CV_1 . Takýmto hierarchickým spôsobom pokračujeme v spracovaní dát smerom od listov ku koreňu stromu. Na Obr. 3.2 je taktiež vidieť, že kompresná funkcia požaduje dva vstupné parametre. V prípade, že hašujeme list stromu, vstupné parametre kompresnej funkcie tvorí stavová matica SM s veľkosťou 64 bajtov a segment C_i o veľkosti 1024 bajtov. Ak hašujeme rodičovský uzol, vstupné parametre tvorí opäť stavová matica $SM(64\text{B})$ a vstupné dátá tvorené zretezovacími hodnotami potomkov daného rodičovského uzla, teda hodnoty $CV_l \parallel CV_p$. Viac o kompresnej funkcií a stavovej matici nájdeme v podkapitole 3.2.1 a 3.2.2.

Aplikáciou kompresnej funkcie na koreň stromu, získame hašovací kód funkcie Blake3, ktorý je prednastavený na dĺžku 64 bajtov. Dĺžku hašovacieho kódu možno rozšíriť na ľubovoľný počet bajtov z rozsahu $0 \leq l < 2^{64}$ a prispôsobiť naším požiadavkám.



Obr. 3.2: Stromova konštrukcia hašovacej funkcie Blake3

3.2.1 Kompresná funkcia $Kf()$

Kompresná funkcia je funkcia, ktorá slúži na výpočet zretezovacích hodnôt. Každá zretezovacia hodnota slúži ako časť vstupných dát pre výpočet nadradeného rodičovského uzla. Príkladom môže byť zretezovacia hodnota CV_0 , ktorá reprezentuje prvú polovicu vstupných dát rodičovského uzla P_0 (Obr. 3.2).

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. Je nutné podotknúť, že listy a rodičovské uzly sa spracúvajú odlišným spôsobom. V prípade rodičovských uzlov je hodnota M rovná $CV_l \parallel CV_p$. Pre listy binárneho stromu platí, že sa list (1024 B) rozdelí na 64-bajtové subbloky M_0, \dots, M_{15} , ktoré sa následne spracúvajú sekvenčne. Viac o spracúvaní listov a rodičovských uzlov nájdeme v podkapitole 3.2.6.

Spracovanie rodičovského uzla možno vyjadriť rovnicou 3.2

$$CV_i = Kf(SM, M). \quad (3.2)$$

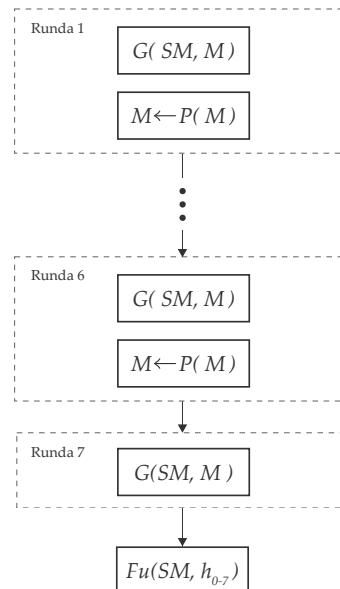
Pričom CV_i je zretezovacia hodnota daného rodičovského uzla a vstupné dátá $M = CV_l \parallel CV_p$. Spracovanie listu možno opísat vzťahom 3.3

$$CV_i = Kf(SM, M_n). \quad (3.3)$$

Pričom CV_i je zreťazovacia hodnota daného listu a M_n je subblok listu, pričom $n \in \{0, \dots, 15\}$. Vo všeobecnosti budeme tieto vstupné dátá označovať ako M .

Výstup kompresnej funkcie tvorí retazec o dĺžke 64 bajtov, ktorý je skrátený na požadovanú veľkosť 32 bajtov. Dĺžka retazca sa neskracuje iba v prípade, ak aplikujeme $Kf()$ na koreň stromu.

Kompresná funkcia $Kf()$ spracúva dátá v 7 rundách. Každá runda, okrem poslednej, je tvorená tvz. G funkciou a následnou permutáciou vstupných dát. 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 $Fu()$. Transformácia stavovej matice je realizovaná podľa Tabuľky 3.4. Aplikáciou $Fu()$ získame 64-bajtový výstup kompresnej funkcie. V prípade potreby úpravy výstupu $Kf()$ pre požadovanú veľkosť zreťazovacej hodnoty CV_i (32 B), sa realizuje skrátenie tohto výstupu. Skrátenie možno chápať ako zahodenie druhej polovice výstupu z $Kf()$. Teda skrátený výstup predstavuje iba prvých 32 bajtov z pôvodného 64-bajtového výstupu. Obr. 3.3 nám poskytuje grafické



Obr. 3.3: Bloková schéma kompresnej funkcie

znázornenie realizácie siedmich rund kompresnej funkcie $Kf()$, kde SM je stavová

matica (64×64), M je blok vstupných dát ($64 \times B$), ktorý v prípade rodičovského uzla predstavuje hodnotu $CV_l \parallel CV_p$ a v prípade listu sú to bloky M_{0-15} . $G()$ predstavuje G funkciu a $P()$ reprezentuje permutáciu vstupných dát podľa Tabuľky 3.3. Posledným krokom kompresnej funkcie je transformácia stavovej matice, označená ako $Fu()$, podľa Tabuľky 3.4.

3.2.2 G funkcia

V každej runde sa vykoná G funkcia, ktorej vstupný parameter SM reprezentuje stavovú maticu a M predstavuje blok vstupných dát. G funkcia má za úlohu zmeniť stav matice v závislosti od jej aktuálneho stavu a bloku vstupných dát. Stavovú maticu SM (64×64) možno zapísat v podobe šestnástich 32-bitových slov. Tieto slová možno vyjadriť v matici 4×4 následovne:

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

Matica vľavo symbolizuje inicializáciu stavovej matice SM a matica vpravo predstavuje aliasy parametrov stavovej matice. Inicializácia SM prebieha na začiatku spracovávania každého bloku dát M (pre rodičovské uzly je $M = CV_l \parallel CV_p$, v prípade listov vstupné dáta M predstavujú bloky M_0, \dots, M_{15}), teda pri každom volaní kompresnej funkcie $Kf()$. Hodnoty h_{0-7} predstavujú vstupné zreťazovacie hodnoty, ktoré sa v prípade spracovania prvého bloku listu (blok M_0) nastavia na korešpondujúce inicializačné vektory IV_{0-7} , ktoré sú zobrazené v Tabuľke 3.2. Pri spracúvaní nasledujúcich blokov listu (bloky M_1, \dots, M_{15}), sú parametre h_{0-7} nastavené na hodnoty zreťazovacích hodnôt predchádzajúcich blokov vid. podkapitola 3.2.6. Napríklad pri spracúvaní druhého bloku listu (blok M_1), teda $Kf(SM, M_1)$, sú parametre h_{0-7} tvorené skráteným výstupom (prvých 32 bajtov zo 64-bajtového výstupu) z kompresnej funkcie predchádzajúceho bloku $Kf(SM, M_0)$. V prípade

rodičovských uzlov sú parametre h_{0-7} nastavené v závislosti od zvoleného módu funkcie Blake3 viď. podkapitola 3.2.6.

Parameter t_0 (32 bitov) a t_1 (32 bitov) tvorí 64-bitový parameter $T = (t_0, t_1)$, ktorý slúži na dva účely, a to na indexáciu listov a ako premenlivý pravok pri rozšírení hašovacieho kódu funkcie Blake3. Indexácia listov spočíva v jednoduchej inkrementácii parametra T , teda pri spracúvaní všetkých blokov prvého listu je tento parameter nastavený na hodnotu $T=0$, pri spracovaní všetkých blokov druhého listu je $T=1$ a pod. V prípade rozšírenia hašovacieho kódu funkcie Blake3, ktorého veľkosť môže byť ľubovoľná z rozsahu $0 \leq l < 2^{64}$, slúži parameter T ako jediná premenlivá hodnota. Rozšírenie sa vykonáva opakovanej aplikáciou kompresnej funkcie $Kf()$ na koreň stromu, pričom každá takáto aplikácia $Kf()$, sa lísi iba parametrom T . Inkrementácia parametra T zabezpečí, že výsledkom každej aplikácie $Kf()$ na koreň stromu získame inú hodnotu reťazca. Spojením (concatenation) všetkých výstupných reťazcov získame výsledný rozšírený hašovací kód funkcie Blake3. Viac o rozšírenom výstupe je možné nájsť v dokumentácii [31]. V prípade rodičovských uzlov je parameter T vždy nastavený na hodnotu 0.

Parameter b označuje veľkosť spracovaného bloku dát M . Hodnota b je nastavená vždy na hodnotu 64. Jedinú výnimku predstavuje posledný blok v poslednom liste binárneho stromu, keďže jeho veľkosť môže byť aj menšia. Parameter d slúži na bitovú separáciu domén, čo možno chápať ako nastavenie špeciálnych vlastností daného bloku M . Modifikácia parametra d sa realizuje nastavením spodných 7 bitov podľa Tabuľky 3.5. Napríklad prvý blok v liste by mal obsahovať informáciu o tom, že je zároveň začiatkom listu. Podľa Tabuľky 3.5 možno túto informáciu informáciu vyjadriť ako $CHUNK_START$. V takomto prípade je hodnota d nastavená na $d=1$. Je zrejmé, že blok dát M môže obsahovať viacero vlastností, napr. koreň stromu môže byť zároveň aj posledným rodičovským uzlom, teda je potrebné nastaviť informáciu $PARENT$ a $ROOT$. Teda $d = 2^2 + 2^3$, čo je 12 resp. ...000 1100 binárne, zapísané na spodných 7 bitoch.

Matica vpravo predstavuje aliasy hodnôt stavovej matice (matice vľavo), ktoré slúžia na jednotný opis parametrov stavovej matice. Tieto aliasy budeme využívať v opise G funkcie.

Popis jednotlivých hodnôt spolu s ich veľkosťou v bitoch nájdeme v Tabuľke 3.1. Blok vstupných dát M možno taktiež zapísat v tvare šestnásťich 32-bitových slov m_0, \dots, m_{15} .

Tabuľka 3.1: Premenné stavovej matice SM

Prehľad premenných v matici 4x4.		
Premenná	Veľkosť v bitoch	Popis
h_{0-7}	256 (8x32)	Vstupné zreťazovacie hodnoty
m_{0-15}	512 (16x32)	Vstupné dátá
$T = t_0, t_1$	64 (32+32)	Počítadlo chunkov
b	32	Počet bajtov v bloku
d	32	Bity slúžiace na bitovu separáciu domén
IV_i	32	Inicializačný vektor

Blake3 využíva rovnakú sadu inicializačných vektorov ako jeden z jeho predchodcov Blake2s, hodnoty IV je možné nájsť v Tabuľke 3.2.

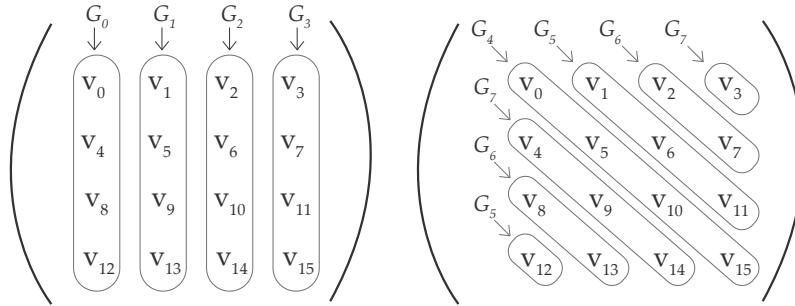
Tabuľka 3.2: Inicializačné vektory pre Blake3

IV_0	0x6A09E667
IV_1	0xBB67AE85
IV_2	0x3C6EF372
IV_3	0xA54FF53A
IV_4	0x510E527F
IV_5	0x9B05688C
IV_6	0x1F83D9AB
IV_7	0x5BE0CD19

Po inicializácii stavovej matice SM nasleduje jej spracovanie spolu s blokom dát M , v siedmich rundách kompresnej funkcie a finálnej transformácii $Fu()$. Pri spracovaní dát v jednotlivých rundách budeme týchto šestnásť 32-bitových slov stavovej matice označovať aliasmi, teda parametrami v_0, \dots, v_{15} . Blok dát M môžeme taktiež zapísat v podobe šestnásť 32-bitových slov m_0, \dots, m_{15} .

G funkcia pozostáva z 8 operácií G_{0-7} a je aplikovaná na každý stĺpec matice 4×4 a potom aj na každú diagonálu. Spracovanie stĺpcov a diagonál prebieha v 8 operáciách G_{0-7} paralelne. Na Obr. 3.4 môžeme vidieť grafické znázornenie ale aj matematické vyjadrenie týchto G operácií.

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



Obr. 3.4: Grafické znázornenie a matematické vyjadrenie realizácie G operácií

Každá operácia označovaná ako $G_k(a, b, c, d, M)$, kde $k \in \{0, \dots, 7\}$ a parametre a, b, c, d predstavujú slová matice vyjadrené aliasmi (v_0, \dots, v_{15}) , je definovaná Algoritmom 1. Parameter M predstavuje blok vstupných dát, ktorý je možno zapísat ako šestnásť 32-bitových slov m_0, \dots, m_{15} . Symbol \oplus predstavuje operáciu XOR, $+$ je znakom sčítania modulo 2^{32} a \ggg predstavuje rotáciu bitov vpravo.

Algoritmus 1 Implementácia jednotlivých operácií $G_k(a, b, c, d, M)$

Vstup: Slová stavovej matice (a, b, c, d) , blok vstupných dát $M = \{m_0, \dots, m_{15}\}$ a index $k \in \langle 0; 7 \rangle$, pričom index k je zviazaný s indexom G operácie

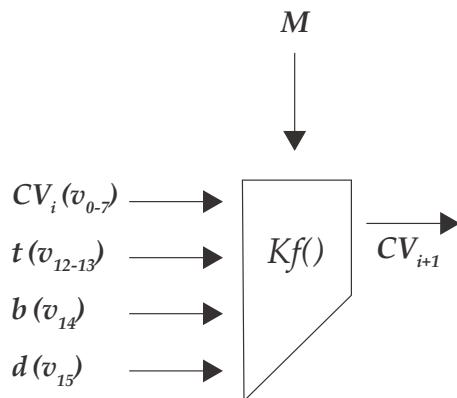
Výstup: Modifikované slová stavovej matice (a, b, c, d)

- 1: $a \leftarrow a + b + m_{2k+0}$
 - 2: $d \leftarrow (d \oplus a) \ggg 16$
 - 3: $c \leftarrow c + d$
 - 4: $b \leftarrow (b \oplus c) \ggg 12$
 - 5: $a \leftarrow a + b + m_{2k+1}$
 - 6: $d \leftarrow (d \oplus a) \ggg 8$
 - 7: $c \leftarrow c + d$
 - 8: $b \leftarrow (b \oplus c) \ggg 7$
-

Algoritmus 1 znázorňuje sekvenčné vykonávanie operácie $G_k(a, b, c, d, M)$, kde k nadobúda hodnoty 0–7 a slová a, b, c, d predstavujú slová stavovej matice vyjadrené aliasmi v_0, \dots, v_{15} . M je blok vstupných dát o veľkosti 64 B, ktorý možno napísat ako m_0, \dots, m_{15} , čo predstavuje slová m_{2k+0} a m_{2k+1} v Algoritme 1, kde index

k je zviazaný s indexom operácie G_k a može nadobúdať hodnoty $k \in \{0, \dots, 7\}$. Teda operácia $G_0(a, b, c, d, M)$ pracuje so slovami $m_{2.0+0}$ a $m_{2.0+1}$ a pod. Každá operácia $G_k()$ realizuje transformáciu štyroch vstupných slov a, b, c, d matice SM , na štyri výstupné slova a, b, c, d .

Obr. 3.5 ilustruje základný princíp spracovania bloku dát kompresnou funkciou, kde môžeme vidieť reprezentáciu vstupných parametrov kompresnej funkcie. Parameter M predstavuje blok vstupných dát a hodnoty CV_i, t, b, d predstavujú hodnoty stavovej matice. V zátvorkach pri jednotlivých hodnotách stavovej matice môžeme vidieť aj ich aliasy v_{0-15} . Všimnime si, že inicializačné vektory IV_{0-3} , teda slova v_{8-11} nie sú v Obr. 3.5 naznačené. Je to preto, že tieto hodnoty sú fixné a sú prítomne v každej aplikácii $Kf()$.



Obr. 3.5: Princíp spracovania bloku dát M kompresnou funkciou $Kf()$

3.2.3 Paralelizmus s využitím SIMD inštrukcií

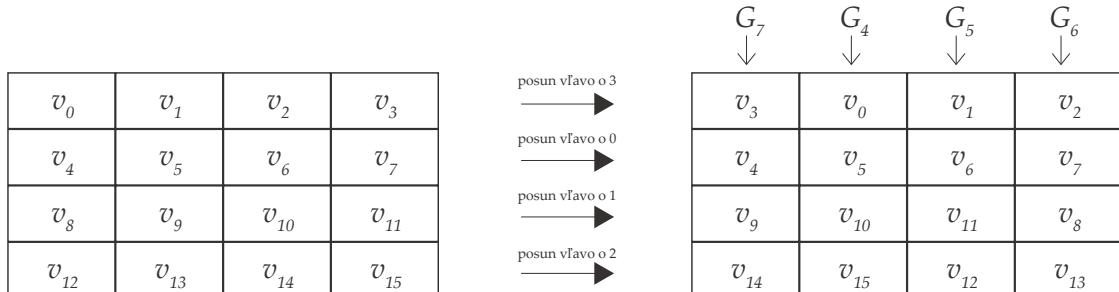
Funkcia Blake3 využíva niekoľko metód, ktorými je schopná zrýchliť proces hašovania. Jednou z týchto metód je využitie SIMD inštrukcií. Existujú 3 rôzne spôsoby využitia SIMD inštrukcii v hašovacej funkcií Blake3. Prvý spôsob je podobný ako u jeho predchodcu Blake2b, resp. Blake2s. Slová matice sa rozdelia po riadkoch do štyroch 128-bitových vektorov následovne. Prvý vektor obsahuje slová v_{0-3} , druhý vektor obsahuje slová v_{4-7} a pod. Slová v_{0-15} predstavujú aliasy parametrov stavovej matice SM . Následne sa vykoná vektorizovaná G funkcia (ktorá pracuje s vektorovými inštrukciami) na všetkých 4 stĺpoch paralelne. Po spracovaní stíp-

cov následuje diagonalizácia vektorov, čo znamená, že sa vykoná vhodná rotácia vektorov tak, aby všetky slová z pôvodných diagonál ležali vhodne pod sebou. Diagonalizáciou dosiahneme stav, kedy môžeme opäť použiť vektorizovanú G funkciu na jednotlivé stĺpce. Posledným krokom je inverzná rotácia vektorov do pôvodného stavu [32]. Takúto realizáciu paralelného spracovania dát s využitím SIMD inštrukcií možno vnímať následovne:

G_0	G_1	G_2	G_3
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}

Obr. 3.6: Paralelné spracovanie stĺpcov operáciami G_{0-3}

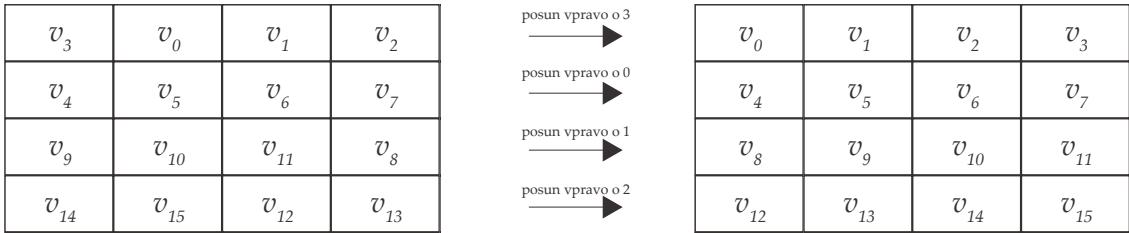
Obr. 3.6 naznačuje vektorizované spracovanie stĺpcov operáciami G_{0-3} paralelne.



Obr. 3.7: Rotácia vektorov a následné paralelné spracovanie stĺpcov operáciami G_{4-7} operácií

Z Obr. 3.7 je vidno, že vhodnou rotáciou vektorov je možné opäť vykonať spracovanie stĺpcov, tentokrát pre diagonálnej stavovej matice SM . Teda je možné vykonať operácie G_{4-7} paralelne po stĺpcoch. Rotácia vľavo je aplikovaná na každý vektor následovne. Nultý vektor rotujeme o 3 miesta vľavo. Prvý vektor rotujeme o 0 miest vľavo resp. nerotujeme vôbec, druhý o 1 miesto vľavo a tretí o 2 miesta vľavo. Po diagonalizácii prebehne spracovanie stĺpcov paralelne.

Posledným krokom je úprava vektorov do pôvodného stavu s využitím inverznej rotácie. Z Obr. 3.8. je vidno, že inverzná rotácia (vpravo) sa aplikuje na každý



Obr. 3.8: Inverzná rotácia vektorov

vektor následovne. Nultý vektor rotujeme o 3 miesta vpravo. Prvý vektor rotujeme o 0 miest vpravo resp. nerotujeme vôbec, druhý o 1 miesto vpravo a tretí o 2 miesta vpravo.

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.

Prvým spôsobom využitia SIMD inštrukcií pre veľé množstvo vstupných dát je metóda, ktorá je veľmi podobná spôsobu využitia SIMD inštrukcií v hašovacej funkcií Blake2bp resp. Blake2sp. V tejto metóde sa viacero listov spracúva paralelne s využitím vektorov a to tak, že každý vektor obsahuje jedno 32-bitové slovo stavovej matice každého listu. Teda prvý vektor bude obsahovať slová stavovej matice v_0 (čo je alias prvého slova stavovej matice) z každého stavu aktuálne spracúvaných listov. Druhý vektor bude obsahovať slova v_1 , tretí v_2 a pod. Šírku takto vytvorených, celkovo 16 vektorov, určuje počet aktuálne spracúvaných listov. Teda ak paralelne spracúvame 4 listy, šírka vektorov bude rovná 128 bitom (4×32 bitov). Následne sa vykoná G funkcia, ktorá spracúva dáta po jednom stĺpci resp. diagonále naraz, pre všetky stavy SM v každom liste, bez potreby diagonalizácie vektorov. Výhodou tohto spôsobu využitia SIMD inštrukcií je aj možnosť využitia inštrukčných sad ako AVX2 a AVX-512 [33][34][35].

Druhý spôsob spracovania dát pomocou SIMD inštrukcií pre väčšie množstvo vstupných dát je veľmi podobný prvému spôsobu (pre spracovanie malých vstupov), kde štvor-slovné (napr. v_0, v_1, v_2, v_3) riadky matice SM tvoria jeden zo 4 vektorov. V tomto prípade je možné využiť väčšie vektory, ktoré budú obsahovať viacero takýchto štvor-slovných riadkov, pričom každy z riadkov patrí inej stavovej matici

SM. Ďalej je princíp rovnaký ako v prvej metóde, t.j. paralelne spracovanie stlpcov a následne vykonaná diagonalizácia vektorov. Po diagonalizácii vektorov sa opäť vykoná paralelne spracovanie stlpcov a úprava vektorov do pôvodného stavu s využitím inverznej rotácie. Hoci je tento spôsob spracúvania dát v hašovacej funkcií Blake3 zriedkavý, jeho využitie môže byť vhodné pre rôzne mikroarchitektúry [31].

Viac informácií o paralelizme a využití SIMD inštrukcií je možné nájsť v dokumentácii hašovacej funkcie Blake3 [31].

3.2.4 Permutácia $P()$

Permutácia bloku vstupných dát M (64 B), zapísaných ako šestnásť 32 bitových slov m_0, \dots, m_{15} , prebieha v každej runde (okrem poslednej) a je realizovaná podľa Tabuľky 3.3. Permutácia sa vykonáva na slovách m_0, \dots, m_{15} , kde indexy slov m_0, \dots, m_{15} reprezentujú hodnoty 0–15 z Tabuľky 3.3.

Tabuľka 3.3: Permutácia bloku dát $P(M)$, pričom $M = m_0, \dots, m_{15}$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	6	3	10	7	0	4	13	1	11	12	5	9	14	15	8

3.2.5 Finálna transformácia stavovej matice $Fu()$

Posledným krokom kompresnej funkcie $Kf()$ je transformácia stavovej matice SM podľa Tabuľky 3.4, kde parametre v'_{0-15} predstavujú aliasy stavovej matice po poslednej aplikácii G funkcie (po 7 runde) a h_{0-7} predstavuje zreťazovacie hodnoty. Teda h_{0-7} predstavuje prvých 8 slov z pôvodne inicializovanej SM pre daný blok (prvých 8 slov matice, ktorých hodnoty zodpovedajú hodnotám matici v prvej runde).

Hodnoty h'_{0-15} , ktoré získame finálnou transformáciou stavovej matice SM , predstavujú 64-bajtový výstup kompresnej funkcie $Kf()$.

Tabuľka 3.4: Finálna transformácia $Fu()$

$h'_0 \leftarrow v'_0 \oplus v'_8$	$h'_8 \leftarrow v'_8 \oplus h_0$
$h'_1 \leftarrow v'_1 \oplus v'_9$	$h'_9 \leftarrow v'_9 \oplus h_1$
$h'_2 \leftarrow v'_2 \oplus v'_{10}$	$h'_{10} \leftarrow v'_{10} \oplus h_2$
$h'_3 \leftarrow v'_3 \oplus v'_{11}$	$h'_{11} \leftarrow v'_{11} \oplus h_3$
$h'_4 \leftarrow v'_4 \oplus v'_{12}$	$h'_{12} \leftarrow v'_{12} \oplus h_4$
$h'_5 \leftarrow v'_5 \oplus v'_{13}$	$h'_{13} \leftarrow v'_{13} \oplus h_5$
$h'_6 \leftarrow v'_6 \oplus v'_{14}$	$h'_{14} \leftarrow v'_{14} \oplus h_6$
$h'_7 \leftarrow v'_7 \oplus v'_{15}$	$h'_{15} \leftarrow v'_{15} \oplus h_7$

3.2.6 Spracovanie uzlov a listov

Merklov hašovací strom funkcie Blake3 pozostáva z dvoch hlavných prvkov, a to listov s veľkosťou 1024 bajtov a rodičovských uzlov s veľkosťou 64 bajtov. Spracovanie týchto stavebných prvkov hašovacieho stromu Blake3 vedie k získaniu zreteľazovacej hodnoty, potrebnej pre budovanie nadradených rodičovských uzlov. Hoci zámer spracovania oboch prvkov je rovnaký, realizacia je odlišná.

Spracovanie uzlov

Spracovanie rodičovského uzla prebieha jednoduchou aplikáciou kompresnej funkcie. Vstupnými parametrami $Kf()$ sú stavova matica SM a blok dát M pričom $M = CV_l \parallel CV_p$, kde CV_l je zreteľazovacia hodnota ľavého potomka a CV_p je zreteľazovacia hodnota pravého potomka. Výsledkom spracovania rodičovského uzla je 32-bajtová zreteľazovacia hodnota.

Stavova matica rodičovského uzla môže vyzerat takto:

$$\begin{pmatrix} k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ IV_0 & IV_1 & IV_2 & IV_3 \\ t_0 & t_1 & b & d \end{pmatrix} \rightarrow \begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix}$$

Blake3 disponuje 3 hašovacími modmi $hash(input)$, $keyed_hash(key, input)$ a $derive_key(context, key_material)$. Základný mód $hash(input)$ berie na vstup dátu o ľubovoľnej veľkosti z rozsahu $0 \leq l < 2^{64}$ bajtov. Mód $keyed_hash(key, input)$ berie na vstup okrem vstupných dát aj 256-bitový kľúč k_{0-7} , ktorého prítomnosť ovplyvní inicializáciu stavovej matice SM a teda hašovací kód je závislý od daného kľúča. Hašovací mód $derive_key(context, key_material)$ berie na vstup retazec ($context$) a kľúčový materiál, obe vstupy môžu byť ľubovoľnej veľkosti z rozsahu rozsahu $0 \leq l < 2^{64}$ bajtov. Hašovací mód $derive_key$ pracuje v 2 fázach. V prvej fáze sa hašuje retazec ($context$) a v druhej fáze sa hašuje kľúčový materiál v závislosti od výstupu hašovacieho kódu z prvej fázy. Každý z týchto módov (okrem základného $hash(input)$) pridá parametru d (v stavovej matici) svoju špecifickú informáciu (flag), ktorá bude nastavená pri každom spracovaní bloku M podľa Tabuľky 3.5. Veľkosť hašovacieho kódu funkcie Blake3 môže byť ľubovoľná z rozsahu $0 \leq l < 2^{64}$ bajtov, čo platí pre všetky módy, avšak pri výstupe menšom ako je 256 bitov je nutné zvažovať isté bezpečnostné riziká [36][37].

V prípade výberu módu, ktorý na vstup berie kľúč ($keyed_hash$) sú hodnoty k_{0-7} rovné danému kľúču. Ak sme zvolili základný hašovací mód $hash(input)$, hodnoty k_{0-7} pozostávajú z korešpondujúcich inicializačných vektorov IV_{0-7} . V prípade módu $derive_key$ sa hašovanie realizuje dvakrát (v dvoch fázach). V prvej fáze hodnoty stavovej matice k_{0-7} pozostávajú z korešpondujúcich inicializačných vektorov IV_{0-7} , no v druhej fáze (pri spracovaní kľúčového materiálu) tvoria hodnoty k_{0-7} prvých 256 bitov z výstupu prvej fázy. Hodnoty k_{0-7} sa nastavujú vždy pri inicializácii stavovej matice SM , teda v každom rodičovskom prvku ale aj pri spracovaní prvého bloku listu M_0 .

Parametre $t0, t1$ a b sú pre rodičovský uzol fixne dané $t_0 = t_1 = 0$. Hodnota veľkosti vstupného bloku b je u rodičovského uzla vždy nastavená na 64. Hodnota d (32 bitov), slúžiaca na bitovú separáciu domén je vhodne nastavená podľa Tabuľky 3.5, pričom separačné bity sa nastavujú len na spodných 7 bitoch.

Tabuľka 3.5: Tabuľka hodnôt premennej d

Flag name	hodnota
<i>CHUNK_START</i>	2^0
<i>CHUNK_END</i>	2^1
<i>PARENT</i>	2^2
<i>ROOT</i>	2^3
<i>KEYED_HASH</i>	2^4
<i>DERIVE_KEY_CONTEXT</i>	2^5
<i>DERIVE_KEY_MATERIAL</i>	2^6

Je zrejmé, že hodnota d može nadobúdať rôzne hodnoty v závislosti od významu spracúvaných údajov. Na príklade 3.4 možno vidieť nastavenie hodnoty d pre koreň hašovacieho stromu podľa Tabuľky 3.5, ktorý je zároveň aj rodičovským uzlom,

$$d = 2^2 + 2^3. \quad (3.4)$$

Pri spracúvaní koreňového uzla je hodnota d obohatená o hodnotu *ROOT* teda 2^3 čo symbolizuje, že výstupom kompresnej funkcie nie je 32-bajtový reťazec ale 64. Výstupný reťazec kompresnej funkcie, aplikovanej na koreň stromu, je hašovacím kódom funkcie Blake3 a jeho veľkosť možno rozšíriť na ľubovoľnú dĺžku z rozsahu $0 \leq l < 2^{64}$ [31].

Spracovanie listov

Listy hašovacieho stromu Blake3 sa kvôli svojej veľkosti musia spracovať inak ako rodičovské uzly. Spracovanie listu prebieha následovne:

1. Blok dát o veľkosti 1024 bajtov sa rozdelí na 64-bajtové subbloky M_n . Posledný list stromu môže byť aj menší ako $1024B$ no nikdy nie prázdny. Na-

príklad ak má posledný list stromu $667 B$, bude rozdelený na 10 subblokov ($10 \times 64 B = 640 B$) a k zvyšným 27 bajtom sa pridá výplňová schéma (padding) v podobe nulových bajtov, čím vznikne celkovo jedenásť 64-bajtových subblokov.

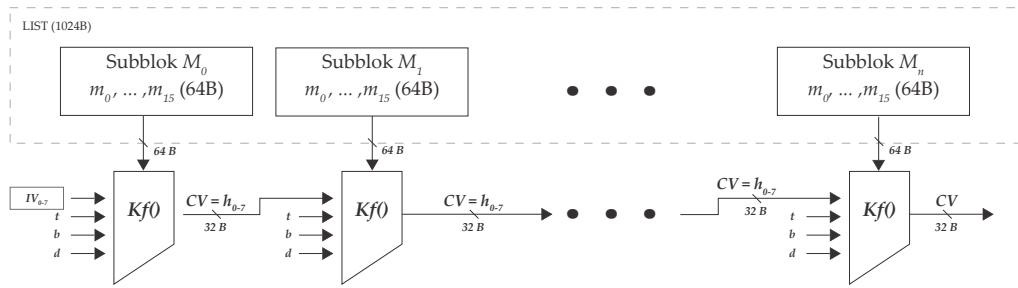
2. Každý z týchto subblokov M_n , kde $n \in <0; 15>$ možno zapísat ako šestnásť 32-bitových slov m_{0-15} . Subblock M_n predstavuje vstupné dátá kompresnej funkcie $Kf()$. Ďalším parametrom $Kf()$ je stavová matica SM , ktorá je vhodne upravená pre spracovanie daného bloku dát viď. Tabuľka 3.1 a Tabuľka 3.5. Matica SM sa inicializuje na začiatku spracovania každého bloku M_n . Pri spracovaní prvého bloku listu M_0 sa hodnoty h_{0-7} nastavia na korešpondujúce inicializačné vektory IV_{0-7} .

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

Počítadlo $T = t_0, t_1$ slúži ako index listu, teda všetky subbloky v prvom liste budú spracované indexom $T=0$, v druhom liste $T=1$ a pod.

3. Pre každý prvý subblock listu platí, že hodnota prvku d stavovej matice bude obsahovať informáciu $CHUNK_START$ (2^0) a každý posledný subblock listu $CHUNK_END$ (2^1).
4. Subbloky sa spracúvajú sekvenčným spôsobom, t.j. s určitou vzájomnou závislosťou. Výstup kompresnej funkcie pri spracovaní n-teho subblocku (subblock M_n) sa stáva vstupom pre spracovanie nasledujúceho subblocku M_{n+1} . Inak povedané, výstup z $Kf(SM, M_n)$ tvorí zreťazovaciu hodnotu h_{0-7} stavovej matice nasledujúceho subblocku viď. obr.3.9.

Bloková schéma na Obr. 3.9 naznačuje sekvenčný proces spracovania subblokov v liste. Všimnime si, že výstup z kompresnej funkcie $Kf()$, teda zreťazovacia hodnota subblocku, tvorí jeden zo vstupov $Kf()$ následujúceho subblocku. Z tejto blokovej



Obr. 3.9: Sekvenčné spracovanie listu binárneho stromu v hašovacej funkcií Blake3

schémy je zrejmé, že zretežovaci hodnotu (slova stavovej matice h_{0-7}) prvého subbloku v liste (M_0) tvoria inicializačné vektory IV_{0-7} .

3.2.7 Blake3 strom

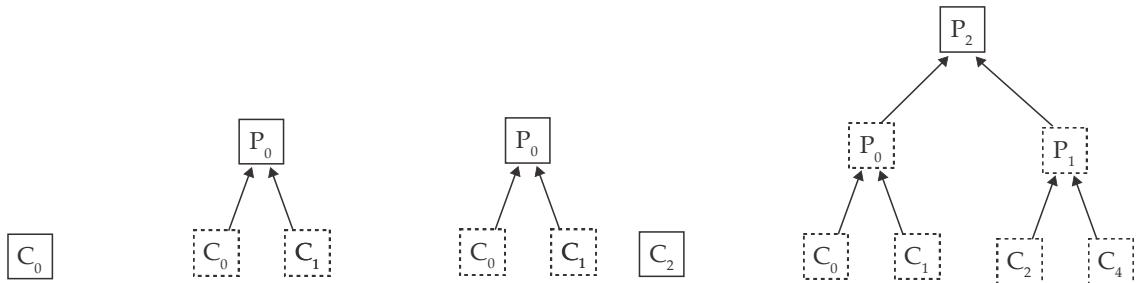
Princípom stromovej hašovacej konštrukcie je hierarchické získavanie zretežovacích hodnôt prvkov daného stromu. Zretežovacie hodnoty daných prvkov tvoria základ pre výpočet zretežovacích hodnôt ich nadriadeného rodičovského uzla. Výsledkom aplikácie kompresnej funkcie $Kf()$ na koreň stromu je hašovací kód funkcie Blake3.

Štruktúru binárneho stromu v hašovacej funkcií Blake3 určujú tieto pravidlá:

1. Každý ľavý podstrom je úplný binárny strom. Všetký listy ľavého podstromu sú v rovnakej hĺbke a počet týchto listov je mocninou čísla 2.
2. Každý ľavý podstrom obsahuje rovnaký alebo väčší počet listov ako odpovedajúci pravý podstrom.

Obr. 3.10 demonštruje proces formovania hašovacieho stromu postupným spracúvaním listov na základe pravidiel uvedených vyššie. Každý z týchto listov C_i je spacúvaný kompresnou hašovacou funkciou, ktorej výstupom je zretežovacia hodnota. Zretežovacie hodnoty dvoch susedných potomkov tvoria vstupy pre výpočet zretežovacej hodnoty ich rodičovského uzla vid. podkapitola 3.2.6. Aplikáciou kompresnej funkcie na koreň stromu, získame hašovací kód funkcie Blake3 zakódovaný

vo formáte little-endian.



Obr. 3.10: Budovanie hašovacieho stromu Blake3

Kedže na budovanie hašovacieho stromu sú potrebné iba zretazovacie hodnoty daných prvkov, je potrebné tieto hodnoty uchovať v pamäti. Autori Blaku3 poukazujú na fakt, že nie je potrebné uchovať v pamäti všetky zretazovacie hodnoty ale iba tie, ktoré sú potenciálnymi kandidátmi pre výpočet zretazovacej hodnoty ďalšieho uzla. Túto skutočnosť demonštruje Obr. 3.10, kde pevné boxy predstavujú zretazovacie hodnoty, ktoré je potrebné uchovať a prerusované boxy predstavujú hodnoty, ktoré je možné uvoľniť z pamäte. Tvorcovia Blaku3 prišli s efektívnym riešením, uchovávať tieto hodnoty v dátovej štruktúre zásobník (stack). Zásobník zretazovacích hodnôt teda uchováva všetky potrebné zretazovacie hodnoty. O pridávanie a úpravu zásobníka sa stará funkcia `add_chunk_chaining_value()`, ktorú možno nájsť v referenčnom kóde². Referenčný kód však nie je optimalizovaný, slúži iba na edukačné účely. Podrobnejší opis hašovacej funkcie Blake3 je možné nájsť v jej dokumentácii [31].

²https://github.com/BLAKE3-team/BLAKE3/tree/master/reference_impl

4 Využitie paralizovateľnej Rust hašovacej funkcie Blake3 v jazyku Python

Python je v súčasnosti jedným z najpopulárnejších programovacích jazykov, ktorého dopyt stúpol za posledných 5 rokov až o 12,1% v porovnaní s inými programovacími jazykmi¹. Vďaka svojím výhodám, ako napríklad intuitívna syntax, jednoduchá interakcia s inými programovacími jazykmi, či široká developerská komunita, získal svoje prvenstvo v oblastiach dátovej vedy, strojového učenia, vývoja webových aplikácií a v mnohých ďalších odvetviach². Je nutné podotknúť, že hoci Python poskytuje veľké množstvo výhod, obsahuje aj niekoľko zásadných nevýhod ako napríklad vysoká spotreba operačnej pamäte alebo jeho nízka rýchlosť [3][2]. Python je pomerne pomalý programovací jazyk v porovnaní s nízko-úrovňovými programovacími jazykmi ako napríklad C, C++. Tieto nevýhody však vieme čiastočne eliminovať prepojením externých modulov, napísaných v inom rýchlejšom programovacom jazyku [6]. V tejto kapitole si demonštrujeme akým spôsobom je možné prepojiť časť kódu paralizovateľnej hašovacej funkcie Blake3, ktorá je napísaná v programovacom jazyku Rust s jazykom Python. Rust je vysoko-úrovňový programovací jazyk špeciálne navrhnutý pre bezpečnú správu pamäte a vývoj súbežných (concurrency) a paralizovateľných aplikácií [38]. Rust patrí k jedným z najrýchlejších programovacích jazykov a jeho rýchlosť je porovnatelná s jazykom C/C++.

Vhodným prepojením externých modulov napísaných v jazyku Rust s Pythonom

¹<https://pypl.github.io/PYPL.html>

²<https://www.crowdbotics.com/blog/python-vs-rust-which-is-better>

dosiahneme niekoľko násobne zrýchlenie a zefektívnenie nášho kódu, pričom si zachováme komfort jazyka Python.

4.1 Prepájacie a mapovacie nástroje pre vytvorenie dynamickej Python knižnice

V prípade prepojenia programovacích jazykov je nutné dbať na kompatibilitu všetkých modulov, programovacích jazykov a podobne. Verzie jednotlivých programovacích jazykov a prídavných modulov využitých v tejto práci sú zobrazené v Tabuľke 4.1.

Tabuľka 4.1: Verzie použitých programovacích jazykov a modulov

Programovací jazyk/modul	Verzia
Rust	rustc 1.56.1
Cargo	cargo 1.56.0
Python	Python 3.8.10
pip	pip 22.0.3
PyO3	0.15.1
Blake3	1.3.1
maturin	0.11.5

V prípade programovacích jazykov Python a Rust, nie je nutné využiť verzie z Tabuľky 4.1, no aj tu existujú minimálne požiadavky a odporúčania v rámci verzií, ktoré sa vzťahujú na modul PyO3. Rust modul PyO3, zodpovedný za mapovanie kódu napísaného v jazyku Rust do jazyka Python má isté minimálne požiadavky na verzie týchto programovacích jazykov. V prípade samotného programovacieho jazyka Rust je to verzia 1.48 alebo vyššia a v prípade Pythonu je to verzia 3.7 a vyššia. Existujú aj špeciálne verzie tzv. nightly versions modulu PyO3, ktoré sú kompatibilné so staršími verziami jazyka Python (2.7 alebo 3.5). V prípade, že pracujeme s inou verziou Rust modulu PyO3 ako je uvedené v Tabuľke 4.1, je vhodné skontrolovať tieto minimálne požiadavky na verzie programovacích jazykov v dokumentácii modulu (<https://pyo3.rs/>). Cargo je balíčkový manažér jazyka Rust, ktorý je zvyčajne súčasťou inštalácie programovacieho jazyka Rust, podobne

ako PIP (package installer for Python), ktorý je typicky súčasťou inštalácie jazyka Python. Pomocou Rust modulu PyO3 je teda možné namapovať zdrojový kód napísany v Ruste s jazykom Python, čo nám následne umožní vytvoriť Python modul. Zhotovenie Python modulu má na starosti terminalový Python nástroj maturin. Maturin je nástroj, ktorý slúži na rýchle a komfortné zostavenie Python modulov, bez zložitej konfigurácie. Modul maturin nie je striktne obmedzený jeho verziou, no odporúčaná verzia modulu maturin pre zachovanie kompatibility je 0.9.0 alebo akákoľvek jeho novšia verzia. Rust modul Blake3 obsahuje celkovo 11 verzii, pričom prvá predstavená verzia Blake3 0.3.3 bola predstavená 28. apríla 2020. Modul Blake3 je pravidelne udržiavaný a optimalizovaný z hľadiska výkonu a bezpečnosti. Súčasná verzia modulu Blake3 je verzia 1.3.1, ktorá bola predstavená 14. februára 2022. V tejto práci budeme pracovať s aktuálne najnovšou verziou Blake3 1.3.1, jednotlivé verzie sú dostupné v github repozitári [19].

4.2 Inštalácia programovacieho jazyka Rust

Programovací jazyk Rust je voľne dostupný a jeho stiahnutie a priebeh inštalácie je možné nájsť v online dokumentácií (<https://doc.rust-lang.org/book/ch01-01-installation.html>) resp. v prílohe C. V prípade operačného systému Linux alebo macOS sa inštalácia realizuje prostredníctvom terminálu. Pre operačný systém Windows je potrebné stiahnuť inštalačný balíček (<https://www.rust-lang.org/tools/install>). V prípade inštalácie programovacieho jazyka Rust na OS Windows, je nutné doinštalovať softvér *C++ build tools for Visual Studio 2013* alebo jeho akúkolvek novšiu verziu (<https://visualstudio.microsoft.com/visual-cpp-build-tools/>).

Počas inštalácie Rustu môžeme naraziť na niekolko terminálových výpisov, ktoré vyžadujú nášu pozornosť.

Prvý takýto vstup, ktorý znázornuje Obr. 4.1 informuje o potrebe inštalácie softvérku *Microsoft C++ build tools*. V tomto prípade je nutné softvér doinštalovať a inštaláciu Rustu spustiť opäťovne.

Ďalší takýto vstup, ktorý znázornuje Obr. 4.2 vyžaduje výber z 3 možnosti. V tomto

```
If you will be targeting the GNU ABI or otherwise know what you are  
doing then it is fine to continue installation without the build  
tools, but otherwise, install the C++ build tools before proceeding.  
  
Continue? (Y/n)  
>
```

Obr. 4.1: Výpis z terminálu 1.

```
Current installation options:  
  
default host triple: x86_64-pc-windows-msvc  
default toolchain: stable (default)  
profile: default  
modify PATH variable: yes  
  
1) Proceed with installation (default)  
2) Customize installation  
3) Cancel installation  
>
```

Obr. 4.2: Výpis z terminálu 2.

prípade zvolíme možnosť číslo „1“, teda predvolenú inštaláciu, v ktorej sa vykoná inštalácia programovacieho jazyka Rust, compilera pre jazyk Rust, balíčkového manažera Cargo a ďalších potrebných závislostí.

Správne nainštalovanie programovacieho jazyka Rust môžeme overiť zistením verzie Rustu a balíčkového manažera Cargo pomocou príkazov `rustc --version` (pre verziu Rustu) a `cargo --version` (pre verziu Carga).

4.3 Inicializácia vývojového prostredia a spustenie kódu

Prvým krokom vývoja našej aplikácie je inicializácia vývojového prostredia pomocou terminálového príkazu:

```
Cargo init
```

Počas inicializácie sa nám v adresári vytvorí niekoľko súborov. Súbor *Cargo.toml* uchováva všetky potrebné informácie o našom projekte, jeho modifikáciou je možné importovať všetky potrebné Rust moduly. Zložka *src* obsahuje súbor *main.rs*, ktorý po komplikácii a spustení na obrazovku vypíše „Hello world“. V zložke *src* by sa mali nachádzať všetky Rust súbory obahujúce zdrojový kód („prípona .rs“).

Preklad kódu (compile) je možné vykonať príkazom *cargo build*. Týmto príkazom spustíme preklad kódu v neoptimalizovanom (developerskom móde). Preklad kódu v optimalizovanom móde je možné vykonať príkazom *cargo build --release*. Spustenie kódu je možné vykonať príkazom *cargo run*, ktorý spustí náš kód tak tiež v neoptimalizovanom móde. Pre spustenie kódu v optimalizovanom móde je potrebné vykonať príkaz *cargo run --release*.

4.4 Inštalácia Rust modulov (crates)

Balíčkový manažer Cargo je zodpovedný za inštaláciu, update a vymazanie modulov jazyka Rust. V iných literatúrach sa môžeme stretnúť s pojmom crates, čo je typické označenie Rust modulov. Inštaláciu modulov možno realizovať príkazom:

```
cargo install [nazov_modulu]  
napr. cargo install b3sum
```

Importovanie modulov do kódu realizujeme pridaním modulu do súboru *Cargo.toml* v nainicializovanom prostredí.

Zdrojový kód 4.1: *Cargo.toml*

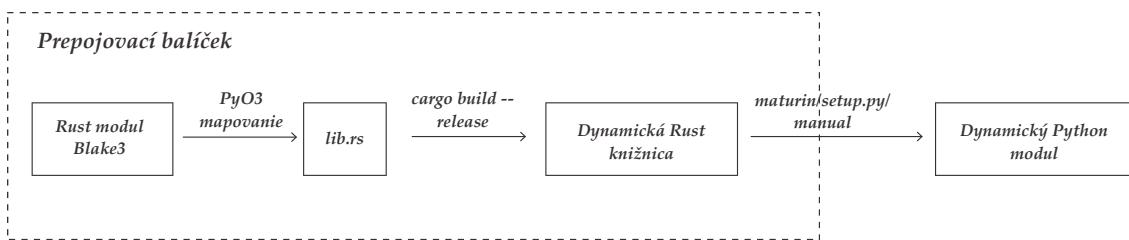
```
[dependencies]  
time = "*"  
chrono = "0.4"
```

V prípade, že moduly iba importujeme do súboru *Cargo.toml* (bez inštalácie), balíčkový manažér Cargo sa postará o ich inštaláciu pri preklade kódu.

4.5 Tvorba dynamickej Python knižnice z externého Rust modulu

Na prepojenie externého modulu vytvoreného v programovacom jazyku Rust s jazykom Python je potrebné namapovať tento externý modul do jazyka Python, čo vyžaduje konverziu objektov medzi jazykom Rust a Python. Mapovanie si môžeme predstaviť ako vytváranie funkcií, resp. tried a metód, ktoré na vstup budú brat premenné jazyka python. Tieto premenne sa pomocou mapovacieho modulu zviažu s typmi premenných programovacieho jazyka Rust a v tele funkcie prebehne nosná časť výpočtov (v našom prípade hašovanie). Cieľom mapovania je získať hodnotu, ktorú je Python schopný spracovať. Medzi najznámejšie mapovacie moduly pre mapovanie jazyka Rust a jazyka Python patrí modul PyO3 a modul cpython. V tejto práci budeme na mapovanie jazykov využívať mapovací modul PyO3, ktorého výhodou je jednoduchá integrácia a zrozumiteľná dokumentácia (<https://pyo3.rs/>). Prepojenie externého modulu možno rozdeliť do niekoľkých krokov:

1. Vytvorenie vývojového prostredia v jazyku Rust.
2. Importovanie potrebných Rust modulov (modifikácia súboru Cargo.toml).
3. Mapovanie Rust modulu a jazyka Python s využitím modulu PyO3.
4. Samotné prepojenie nášho modulu a jeho import do jazyka Python.



Obr. 4.3: Bloková schéma mapovania externého Rust modulu do dynamického Python modulu

Obr. 4.3 poskytuje grafické znázornenie celého procesu prepojenia externého modulu (Blake3) napísaného v jazyku Rust s jazykom Python a následne vytvorenie dynamického Python modulu, ktorý využijeme v demonštračnej aplikácii.

Prepojenie externého Rust modulu si demonštrujeme na príklade hašovacej funkcie Blake3, ktorej plne optimalizovaná a paralizovateľná verzia je napísaná práve v jazyku Rust. Prepojením hašovacej funkcie Blake3, napísanej v jazyku Rust, zabezpečíme to, že nosná časť výpočtov bude spracovaná jazykom Rust, čo má za následok niekoľko násobne zrýchlenie procesu hašovania. V závere tejto kapitoly zhodnotíme a porovnáme výsledky získané hašovaním súboru s využitím funkcie Blake3 pre jej rôzne implementácie. Porovnanie zahrňa aj našu prepojenú implementáciu hašovacej funkcie Blake3.

4.6 Prepojenie Rust implementácie hašovacej funkcie Blake3 s jazykom Python

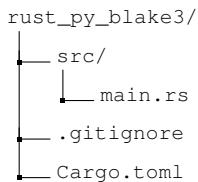
Táto podkapitola slúži ako návod pre vytvorenie dynamického Python modulu prepojením už existujúceho Rust modulu. V tomto prípade budeme prepájať hašovaciu funkciu Blake3 napísanu v programovacom jazyku Rust, ktorú v závere tejto kapitoly importujeme do našej demonštračnej Python aplikácie, následne modul otestujeme a porovnáme aj s inými implementaciami hašovacej funkcie Blake3.

4.6.1 Inicializácia vývojového prostredia jazyka Rust

Prvým krokom prepojenia externého Rust modulu je vytvorenie vývojového prostredia. Vývojové prostredie je možné nainicializovať príkazom *cargo new [názov_balíčka]*. Vykonaním tohto príkazu sa nám vytvorí zložka s názvom nášho prepájacieho balíčka, ktorá bude obsahovať niekoľko potrebných súborov s ktorými budeme pracovať.

Vykonaním príkazu *cargo new rust_py Blake3* vytvoríme nový adresár s názvom

rust_py_blaKE3, ktorý bude obsahovať nasledujúce súbory:



4.6.2 Import Rust modulov

Pridanie Rust modulov do nášho novo-vytvoreného prepájacieho balíčka *rust_py_blaKE3* realizujeme jednoduchou modifikáciou súboru *Cargo.toml*. Manuálna inštalácia prídavných modulov nie je potrebná, pretože o ich inštalácii sa postará balíčkový manažér Cargo pri preklade. Súbor *Cargo.toml* pre mapovanie hašovacej funkcie Blake3 môže vyzerat takto:

Zdrojový kód 4.2: Cargo.toml

```
[package]
name = "rust\py\blaKE3"
version = "0.1.0"
edition = "2021"

[lib]
name = "rust\py\blaKE3"
crate-type = ["cdylib"]

[dependencies.pyo3]
version = "0.15.1"
features = ["extension-module"]

[features]
neon = ["blaKE3/neon"]

[dependencies]
blaKE3 = { version = "1.3.1", features = ["rayon"] }
time = "*"
chrono = "0.4"
```

Sekcia *package* obsahuje základné informácie o našom prepájacom balíčku ako názov balíčka, verzia a edícia. Meno balíčka môže byť ľubovoľné, no malo by sa zhodovať s menom adresára v ktorom sa nachádza (teda *rust_py_blaKE3*). Toto meno sa zhoduje s názvom balíčka, ktorý sme zadávali v príkaze *cargo new rust_py_blaKE3*.

Sekcia *lib* obsahuje informácie o dynamickej knižnici, ktorá je výsledkom celého mapovania a následného prekladu. Túto mapovaciu dynamickú knižnicu importujeme do nášho python kódu a pomocou nej budeme hašovať súbor s využitím hašovacej funkcie Blake3 napísanej v jazyku Rust. Názov tejto dynamicej knižnice nemusí byť totožný s menom nášho balíčka, no pre vyhnutie sa akýmkoľvek komplikáciám ponecháme názov dynamickej knižnice zhodný s názvom prepájacieho balíčka.

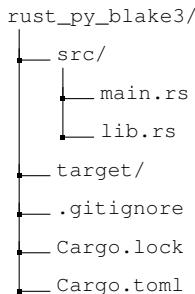
V sekcií *dependencies.pyo3* definujeme verziu rozširovacieho mapovacieho modulu PyO3. Verziu modulu PyO3 nastavíme na 0.15.1.

V prípade, že chceme hašovanie uskutočniť na ARM procesoroch pridáme do sekcie *features* funkcionality neon, čo predstavuje akési rozšírenie pre spracovanie SIMD instrukcií na ARM procesoroch (<https://www.arm.com/technologies/neon>). V tejto práci budeme na hašovanie dát využívať procesor od spoločnosti Intel, teda sekciu *features* necháme prázdnú resp. vynecháme úplne. Poslednou časťou súboru *Cargo.toml* je sekcia *dependencies*, ktorá obsahuje všetky prídavné moduly ako napr. modul Blake3 s využitím funkcie rayon, zodpovednej za paralyzovateľnosť, či moduly time a chrono, ktorými môžeme otestovať rýchlosť a efektívnosť hašovania funkcie Blake3 v jazyku Rust.

4.6.3 Mapovenie Rust modulu s jazykom Python

Na mapovanie modulu Blake3, ktorý je napísaný v jazyku Rust budeme využívať rozširovací modul PyO3. Podrobnejší opis tohto modulu ako aj návod na jeho používanie nájdeme v dokumentácii (<https://pyo3.rs/v0.15.1/>). Je potrebné dodať, že na mapovanie kódu resp. modulu musíme ovládať základne princípy programovacieho jazyka Rust. Odporúčaná literatúra pre efektívne štúdium jazyka Rust je jeho dokumentácia postavená na základe rôznych praktických príkladov (<https://doc.rust-lang.org/stable/rust-by-example/>).

Pre mapovanie modulu Blake3 je nutné vytvoriť v adresári *rust_py_blaKE3/src* súbor *lib.rs*, ktorý bude obsahovať samotné mapovanie. Teda štruktúra nášho adresára by mohla vyzerat takto:



Pridaním súboru *lib.rs* sa nám automaticky vytvorí zložka target spolu so súborom *Cargo.lock*. Mapovanie funkcie Blake3, napísanej v súbore *lib.rs*, môže vyzerať nasledovne:

Zdrojový kód 4.3: Mapovanie Python a Rust objektov v súbore lib.rs

```

use pyo3::prelude::*;
use pyo3::types::{PyAny, PyBytes};

#[pyfunction]
fn rustypy_blaKE(_py: Python, data:&[u8]) -> PyResult<[u8;32]>{
    let mut _hasher = blake3::Hasher::new();
    _hasher.update_rayon(data);
    let result_from_update = *_hasher.finalize().as_bytes();
    return Ok(result_from_update);
}

#[pymodule]
fn rustypy(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(rustypy_blaKE, m)?);
    Ok(())
}

```

Prvý riadok kódu naznačuje využívanie rozšírovacieho modulu *pyo3*, ktorý používa špecialné makrá ako napríklad `#[pyfunction]` alebo `#[pymodule]`. `#[pyfunction]` má za úlohu definovať našu python funkciu, ktorú budeme neskôr volať v našom python kóde. Funkcia *rustypy_blaKE()* realizuje hašovanie dát pomocou funkcie Blake3. Návratová hodnota funkcie *rustypy_blaKE()* je objekt *PyBytes*, ktorý je rozpoznateľný jazykom Python. Konverziu dátových typov potrebnú pre prepojenie rust objektov s jazykom python môžeme nájsť v dokumentácii rozšírovacieho modulu PyO3 (<https://pyo3.rs/v0.15.1/conversions/tables.html>). Dôležitou funkciou pre správne

importovanie našej knižnice do python kódu je funkcia `rust_py_blake3()`. Táto funkcia musí byť definovaná Macrom `#/[pymodule]` a jej názov musí byť zhodný s názvom knižnice, ktorý sme zadávali v súbore `Cargo.toml` (v našom prípade musí byť názov tejto funkcie „`rust_py_blake3`“). Ak by názov funkcie neboli totožné s názvom našej dynamickej knižnice, import nášho modulu do python aplikácie by neboli možné.

V prípade, že sme mapovanie dokončili môžeme pokračovať s prekladom nášho prepojovacieho balíčka `rust_py_blake3`. Preklad modulu vykonáme príkazom `cargo build --release` (preklad je nutné vykonať z adresára „`./rust_py_blake3`“). Prepínačom `--release` sme povedali, že preklad sa ma vykonať v optimalizovanom móde.

4.6.4 Vytvorenie Python dynamického modulu

Finálnym krokom, ktorý je potrebný vykonať pre správne využitie a otestovanie nášho prepájacieho balíčka „`rust_py_blake3`“, je vytvorenie dynamického Python modulu a následne importovanie tohto modulu do testovacej aplikácie napísanej v jazyku Python. Vytvoriť a následne použiť dynamický Python modul vytvorený z nášho prepojovacieho Rust balíčka je možné realizovať 3 spôsobmi:

1. Python modul maturin.
2. Python modul Setup-tools.
3. Manuálne vytvorenie dynamického modulu.

maturin

Najjednoduchší spôsob, ako prvýkrát vyskúšať náš prepajací modul generovaný rozširovacím modulom PyO3, je použiť maturin. Maturin je nástroj na vytváranie a publikovanie Python modulov založených na programovacom jazyku Rust. Maturin je známy tým, že poskytuje menšiu flexibilitu ako modul setup-tools

alebo manuálne vytvorenie dynamického modulu, no jeho používanie je nenáročné. Nasledujúce kroky nastavia virtuálne prostredie, do ktorého si nainštalujeme python modul maturin.

V adresári *rust_py_blaKE3* vytvoríme Python virtuálne prostredie (virtual environment) do ktorého si nainštalujeme modul maturin.

```
py -m venv .env  
.env\Scripts\activate
```

Týmito príkazmi platné pre OS Windows vytvoríme a spustíme virtuálne prostredie jazyka python. Pre vytvorenie a spustenie virtualného prostredia pre OS Linux alebo macOS možno postupovať podľa návodu (<https://docs.rs/pyo3/latest/pyo3/>). Následne nainštalujeme modul maturin pomocou príkazu:

```
pip install maturin
```

Samotné vytvorenie dynamickej knižnice vykonáme príkazom:

```
maturin develop --release
```

kde opäť použijeme prepínač *--release* pre vytvorenie modulu v optimalizovanom móde. Posledným krokom je import a testovanie novo vytvoreného dynamického Python modulu. Demonštračná aplikácia *test.py* (ktorá sa môže nachádzať v ľubovoľnom adresári) môže vyzerat takto:

Zdrojový kód 4.4: Import a otestovanie dynamickej Python knižnice v súbore *test.py*

```
import rust_py_blaKE3 as r  
print(r.rustypy_blaKE(b"Testuje funkciu BlaKE3"))
```

kde prvý riadok má za úlohu importovať naš novovytvorený dynamický Python modul (vytvorený pomocou modulu maturin) a druhý riadok demonštruje využitie konkrétnej funkcie *rustypy_blaKE()* ktorú sme definovali v súbore lib.rs. Spustenie súboru *test.py* je nutné vykonať z virtuálneho prostredia.

```
.\env\Scripts\activate  
py [cesta_k_app\test.py]
```

Setup-tools

Druhou metódou pre vygenerovanie dynamického Python modulu z nášho prepájacieho Rust modulu „rust_py_blake3“ je možné realizovať pomocou modulu *setup-tools*, ktorý je vo všeobecnosti flexibilnejší a ako *maturin*, no práca s ním je náročnejšia. Pre prácu s Python modulom *setup-tools* je potrebné nainštalovať niekolko python balíčkov.

```
pip install setuptools-rust  
pip install setuptools  
pip install wheel
```

V adresári „rust_py_blake3“ vytvoríme nový python súbor *setup.py* a adresár s názvom nášho prepájacieho balíčka teda „rust_py_blake3“, ktorý bude obsahovať súbor *__init__.py*. Vytvorením týchto súborov by náš adresár mohol vyzerat takto:

```
rust_py_blake3/  
└── rust_py_blake3/  
    └── __init__.py  
src/  
└── main.rs  
    └── lib.rs  
target/  
.gitignore  
Cargo.lock  
Cargo.toml  
└── setup.py
```

Použité súbory *setup.py* a *__init__.py* vytvorené pre túto prácu je možné nájsť v github repozitári [39]. Princípy tvorby súboru *setup.py* je môžé si naštudovať v dokumentácii (<https://setuptools-rust.readthedocs.io/en/latest/>). Spustenie modulu *setup-tools* pre vytvorenie požadovaného Python modulu možeme vykonať následovným príkazom

```
py setup.py install
```

Po vykonaní príkazu *py setup.py install* sa nám v adresári „rust_py_blake3“ vytvorí adresár „rust_py_blake3.egg-info“, ktorý obsahuje prídavne informácie o našom novo vytvorenom Python module. Po úspešnom vytvorení Python modulu

„rust_py_blake3“, môžeme tento modul využívať bez akéhokoľvek virtualného vývojového prostredia. Teda testovaciu aplikáciu *test.py*, ktorá obsahuje kód na overenie funkčnosti nášho modulu, môžeme spustiť následovne:

```
py cesta_k_suboru\test.py
```

Manuálne vytvorenie dynamického Python modulu

Metóda manuálneho vytvorenia dynamického Python modulu sa môže javiť ako relatívne rýchle riešenie, bez potreby inštalácie doplnkových modulov. Po preklade nášho mapovacieho Rust modulu, ktorý bol preložený príkazom *cargo build --release*, sa nám v adresári "rust_py_blake3/target/release" vytvoril súbor (dynamická knižnica) s názvom *rust_py_blake3.dll* resp. *librust_py_blake3.dll*. Tento súbor môže nadobúdať aj iné názvy (Tabuľka 4.2) v závislosti od nášho OS. Túto dynamickú knižnicu je potrebné nakopírovať do zložky s našou testovacou aplikáciou *test.py* a následne ju premenovať podľa Tabuľky 4.2, čím získame dynamický Python modul.

Tabuľka 4.2: Prenovanie dynamického modulu v závislosti od operačného systému

Prenovanie dynamického modulu		
Operačný systém	Názov dynamického modulu	Po premenovaní
macOS	rust_py_blake.dylib alebo librust_py_blake3.dylib	rust_py_blake3.so
Windows	rust_py_blake.dll alebo librust_py_blake3.dll	rust_py_blake3.pyd
Linux	rust_py_blake.so alebo librust_py_blake3.so	rust_py_blake3.so

Import dynamického Python modulu do testovacej aplikácie je vykonaný v prvom riadku súboru *test.py*. Spustenie testovacieho súboru vykonáme príkazom:

```
py cesta_k_suboru\test.py
```

Príklad prepojenia externého Rust modulu hašovacej funkcie Blake3, pre všetky vyššie uvedené metódy je možné nájsť v github repozitári [39]. Github repozitár taktiež obsahuje Python dynamickú knižnicu *rust_py_blake3.pyd*, ktorú môže čitateľ využiť a otestovať hned bez akejkoľvek ďalšej konfigurácie. Táto Python dynamická knižnica je vhodná pre operačný systém Windows. Jediná podmienka, ktorú je nutné splniť pre otestovanie modulu je vložiť súbor *rust_py_blake3.pyd* do adresára, v ktorom sa nachádza testovací kód. Experimentálne testovanie rýchlosťi rôznych implementácií hašovacej funkcie Blake3, vrátane našej prepojenej implementácie, je možné nájsť v kapitole 5.2.

5 Experimentálne výsledky

Táto kapitola obsahuje výsledky meraní, ktoré boli realizované v experimentálnej časti tejto bakalárskej práce. Kapitola obsahuje výsledky rýchlosných testov, vykonaných na demonštračných aplikáciách, ktoré majú za úlohu porovnať kryptografické moduly jazyka Python. Kapitola taktiež zahŕňa výsledky meraní rýchlosných testov, vykonaných na rôznych implementáciách hašovacej funkcie Blake3, vrátane nami vytvorennej prepojenej implementácie *rusty_py_blaKE.py*. Demonštračné aplikácie vytvorené v rámci experimentálnej časti tejto bakalárskej práce sú dostupné v prílohe **D** alebo v github repozitári [39]. Hardvérové vybavenie, na ktorom boli vykonané všetky merania je nasledovné. Centrálna procesorová jednotka (CPU) I5-8250U od spoločnosti Intel s celkovým počtom 4 jadier, resp. 8 logickejich vlákien so základnou frekvenciou 1,60 GHz (<https://ark.intel.com/content/www/us/en/ark/products/124967/intel-core-i58250u-processor-6m-cache-up-to-3-40-ghz.html>). Typ operačná pamäť (RAM) je DDR4 s frekvenciou 2400 MHz a veľkosťou 8 GB a pamäť SSD Micron 1100 256 GB s maximálnou rýchlosťou čítania 530 MB/s. Realizácia meraní bola vykonaná na 64-bitovom operačnom systéme Windows 10 od firmy Microsoft. Pri realizácii výkonnostných porovnávacích testov je vhodné postupovať spôsobom, ktorým získame konzistentné hodnoty merania a minimalizujeme odchýlky v meraniach. Jednou z najefektívnejších techník na dosiahnutie konzistentných výsledkov je vypnutie funkcionality turboboost a hyper-threading na centrálnej procesorovej jednotke [40].

5.1 Vyhodnotenie experimentálneho porovnania kryptografických knižníc jazyka Python

Pre vybrane kryptografické moduly jazyka Python sme vykonali rýchlosné testy, v ktorých sme testovali symetrické blokové šifry ako AES, 3DES, CAST5, Blowfish, symetrické prúdové šifry ChaCha20, ARC4, Salsa20 a hašovacie funkcie rodiny SHA2, SHA3 a funkcie Blake2s, Blake2b. Výsledky rýchlosných testov sú zobrazené v tabuľkách 5.1, 5.2, 5.3, 5.4, 5.5, 5.6. Výsledné hodnoty meraní znázornené v tabuľkách 5.1, 5.2, 5.5, 5.6 predstavuje priemernú hodnotu merania rýchlosi daného algoritmu pri celkovo 20 vykonaných meraniach. Testovací súbor, ktorý bol spracovávaný danými kryptografickými primitívami má veľkosť cca 100 MB. Zdrojové kódy demonštračných aplikácií, vytvorené pre testovanie týchto kryptograficých modulov, možno nájsť v Github repozitári [39] alebo v prílohe **D**. V repozitári sa nachádzajú demonštračné aplikácie, ktoré využívajú niektoré ďalšie algoritmy ako DH, RSA, či ECDSA, ktoré boli testované z hľadiska správnosti vykonávania kryptografických úkonov.

Tabuľka 5.1: Výsledky merania rýchlosných testov, vykonaných na procesore I5-8250U a OS Windows 10, pre symetrické prúdové šifry

Rýchlosný test – Symetrické prúdové šifry		
Algoritmus	Modul Cryptography	Modul PyCryptodome
ChaCha20	0,2418 s	0,6713 s
ChaCha20Poly1305	0,2401 s	0,9264 s
Salsa20	—	0,9926 s
ARC4	0,5525 s	0,4509 s

Tabuľka 5.1 znázorňuje rýchlosne porovnanie symetrických prúdových šifier, ktoré sú implementované v kryptografickom module Cryptography a PyCryptodome. Z Tabuľky 5.1 je vidieť, že modul Cryptography je niekoľko násobne rýchlejší ako PyCryptodome, avšak neobsahuje prúdovú šifru Salsa20. Šifrovanie symetrickými prúdovými šiframi bolo realizované na súbore s veľkosťou 100 MB a výsledne hodnoty predstavujú priemernú rýchlosť šifrovania pri celkovo 20 vykonaných meraniach.

Tabuľka 5.2: Výsledky merania rýchlosťných testov, vykonaných na procesore I5-8250U a OS Windows 10, pre symetrické blokové šifry

Rýchlosťný test – Symetrické blokové šifry				
Algoritmus	Mód - ECB	Mód - CBC	Mód - CTR	Modul
AES(128)	0,2257 s	0,3961 s	0,1798 s	Cryptography
	0,1758 s	0,5962 s	0,1678 s	PyCryptodome
3DES	7,4142 s	7,3613 s	—	Cryptography
	7,4813 s	7,9763 s	7,5829 s	PyCryptodome
CAST5	1,9021 s	1,8051 s	—	Cryptography
	2,1216 s	2,5302 s	2,1562 s	PyCryptodome
Blowfish	1,8336 s	1,9066 s	—	Cryptography
	1,6371 s	2,0676 s	1,6463 s	PyCryptodome

Z Tabuľky 5.2 je vidno, že modul Cryptography je opäť rýchlejší ako modul PyCryptodome, no jeho nevýhodou môže byt pravé limitácia CTR módu, ktorá je podporovaná iba pre šifru AES. Šifrovanie symetrickými blokovými šiframi bolo realizované na súbore s veľkosťou 100 MB a výsledne hodnoty predstavujú priemernú rýchlosť šifrovania pri celkovo 20 vykonaných meraniach.

Tabuľka 5.3: Výsledky merania rýchlosťných testov, vykonaných na procesore I5-8250U a OS Windows 10, pre digitálny podpis DSA a ECDSA

Rýchlosťný test – Digitálne podpisy				
Algoritmus	Veľkosť súboru	Generovanie kľúčov	Digitálny Podpis	Modul
DSA	100 MB	min: 0,271 s	0,493 s	Cryptography
	200 MB	max: 5,2748 s	1,022 s	Cryptography
	1 GB	avg: 0,998 s	5,421 s	Cryptography
DSA	100 MB	min: 5,0326 s	1,061 s	PyCryptodome
	200 MB	max: 41,476 s	2,139 s	PyCryptodome
	1 GB	avg: 14,527 s	10,562 s	PyCryptodome
ECDSA	100 MB	min: 0,002 s	0,479 s	Cryptography
	200 MB	max: 0,146 s	1,024 s	Cryptography
	1 GB	avg: 0,004 s	4,741 s	Cryptography
ECDSA	100 MB	min: 0,005 s	1,067 s	PyCryptodome
	200 MB	max: 0,011 s	2,143 s	PyCryptodome
	1 GB	avg: 0,006 s	10,378 s	PyCryptodome

Tabuľka 5.3 demonštruje rýchlosť generovania kľúčov a následne podpísanie súboru algoritmami DSA a ECDSA. Tabuľka 5.3 poskytuje minimálnu, maximálnu a priemernú dobu trvania generovania kľúčov (pri 100 vykonaných meraniach) a zároveň aj dobu podpisu súborov o veľkosti 100 MB, 200 MB a 1 GB. Modulus v algoritme

DSA je 3072, čo odpovedá ekvivalentnej kryptografickej bezpečnosti pre použitú hašovaciu funkciu SHA256. V prípade ECDSA je použitá taktiež hašovacia funkcia SHA256 a eliptická krivka P-256.

Tabuľka 5.4: Výsledky merania rychlostných testov, vykonaných na procesore I5-8250U a OS Windows 10, pre RSA-OAEP s rôznou veľkosťou modula

Rýchlosťny test – RSA-OAEP					
Veľkosť vstupu	Modulus	Cryptography		PyCryptodome	
		Generovanie kľúčov	Šifrovanie	Generovanie kľúčov	Šifrovanie
190 B	2048	0,348 s	0,00041 s	0,791 s	0,0042 s
250 B	4096	2,749 s	0,00047 s	21,919 s	0,0094 s
500 B	8192	5,058 s	0,00089 s	293,414 s	0,0276 s

Tabuľka 5.4 demonštruje RSA-OAEP šifrovanie pre dĺžky vstupov 190 B, 250 B a 500 B a generovanie klúčov pre moduly s o veľkosti 2048, 4096, 8192. Výsledne hodnoty meraní tvoria aritmetický priemer po celkovo 20 vykonaných meraniach.

Tabuľka 5.5: Výsledky merania rychlostných testov,, vykonaných na procesore I5-8250U a OS Windows 10, pre hašovacie funkcie

Rýchlosťny test – Hašovacie funkcie		
Algoritmus	Čas	Modul
SHA2-224	0,4735 s	Cryptography
	1,0186 s	PyCryptodome
	0,4781 s	Hashlib
SHA2-256	0,4807 s	Cryptography
	1,0107 s	PyCryptodome
	0,4803 s	Hashlib
SHA2-384	0,3274 s	Cryptography
	0,6428 s	PyCryptodome
	0,3162 s	Hashlib
SHA2-512	0,3211 s	Cryptography
	0,6449 s	PyCryptodome
	0,3166 s	Hashlib

Tabuľky 5.5 a 5.6 znázorňujú hašovanie súboru o veľkosti 100 MB s využitím rôznych hašovacích funkcií podporovaných Python kryptografickými modulmi Cryptography, PyCryptodome a Hashlib. Výsledné hodnoty meraní predstavujú priemernú rýchlosť šifrovania pri celkovo 20 vykonaných meraniach.

Na základe výsledkov vynonaných meraní možno odvodiť záver, že kryptografický

Tabuľka 5.6: Výsledky merania rýchlosných testov pre, vykonaných na procesore I5-8250U a OS Windows 10, hašovacie funkcie

Rýchlosný test – Hašovacie funkcie		
Algoritmus	Čas	Modul
SHA3-224	0,5608 s	Cryptography
	0,7921 s	PyCryptodome
	0,6278 s	Hashlib
SHA3-256	0,5901 s	Cryptography
	0,817 s	PyCryptodome
	0,6534 s	Hashlib
SHA3-384	0,761 s	Cryptography
	1,0595 s	PyCryptodome
	0,8518 s	Hashlib
SHA3-512	1,1051 s	Cryptography
	1,5101 s	PyCryptodome
	1,2274 s	Hashlib
Blake2b	0,3057 s	Cryptography
	0,5395 s	PyCryptodome
	0,3875 s	Hashlib
Blake2s	0,4865 s	Cryptography
	0,9105 s	PyCryptodome
	0,6105 s	Hashlib

modul Cryptography sa javí v priemere ako najrýchlejší a zároveň je aj najrobustnejší, teda podporuje najviac kryptografických algoritmov a protokolov. Jeho limitáciu v môžeme nájsť napríklad v čítačovom móde (CTR) pre blokové symetrické šifry, pretože tento mód je podporovaný iba pre šifru AES. Na druhej strane modul PyCryptodome je v tomto smere flexibilnejší, no výkonnostne sa javí ako najslabší. Zaujímavou voľbou pre hašovanie dát je kompaktný modul Hashlib, ktorý vďaka svojej jasnej a stručnej dokumentácii je najjednoduchší na používanie.

5.2 Experimentálne porovnanie viacerých implemetácií hašovacej funkcie Blake3

Táto kapitola je stručným zhodnotením testovania hašovacej funkcie Blake3 a porovnanie dosiahnutých meraní pre rôzne implementácie. Vykonanie rých-

lostných testov hašovacej funkcie Blake3 prebehlo na optimalizovanej Rust verzii funkcie Blake3¹, referenčnom Python kóde² a na našej demonštračnej prepojovavej implementácii [39]. Rýchlosťne porovnanie zahŕňa merania rýchlosťi pomocou hašovacieho a overovacieho softvéru TurboSFV (<https://www.turbosfv.com/>). TurboSFV je nástroj na výpočet a validáciu kontrolných súčtov pre súbory, či hašovanie daných súborov. Nástroj TurboSFV podporuje viacero hašovacích algoritmov ako napríklad MD5, hašovacie funkcie rodiny Blake2, SHA2, SHA3, či funkciu Blake3. Testovanie rýchlosťi týchto implementácií bolo vykonané na centrálnej procesorovej jednotke (CPU) I5-8250U od spoločnosti Intel (<https://ark.intel.com/content/www/us/en/ark/products/124967/intel-core-i58250u-processor-6m-cache-up-to-3-40-ghz.html>). Hašovanie prebehlo na testovacích súboroch o veľkostiach 100 MB, 200 MB a 1 GB. Výsledky testovania reprezentuje Tabuľka 5.7.

Tabuľka 5.7: Rýchlosťny test hašovacej funkcie Blake3, vykonaných na procesore I5-8250U a OS Windows 10, pre rôzne implementácie funkcie Blake3

Rýchlosťny test – Hašovacia funkcia Blake3 otestovaná na rôznych implementáciách					
Implementácia	Vlákna (Threads)/ Jadrá	100 MB	200 MB	1 GB	
Rust	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 vlákiem	0,019 s	0,031 s	0,165 s	
	6 vlákiem	0,015 s	0,029 s	0,153 s	
	7 vlákiem	0,014 s	0,028 s	0,142 s	
	8 vlákiem	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 vlákiem	0,021 s	0,034 s	0,171 s	
	6 vlákiem	0,018 s	0,032 s	0,167 s	
	7 vlákiem	0,017 s	0,031 s	0,156 s	
	8 vlákiem	0,016 s	0,029 s	0,146 s	
TurboSFV	1 jadro	0,388 s	0,768 s	3,874 s	
	2 jadrá	0,215 s	0,422 s	2,093 s	
	4 jadrá	0,127 s	0,236 s	1,105 s	
Python ref. impl.	1 vlákno	4224,296 s	14181,683 s	0,000 s	

Tabuľka 5.7 predstavuje výsledky rýchlosťného testu hašovacej funkcie Blake3 pre rôzne implementácie. V prípade oficiálnej optimalizovanej paralizovateľnej imple-

¹<https://docs.rs/blake3/latest/blake3/>

²https://github.com/oconnor663/pure_python_blaKE3

mentácie napisanej v programovacou jazyku Rust a našej demonštračnej prepojenej implementácii rust_py_blaKE3 bolo možné hašovanie otestovať z hľadiska rýchlosi na rôznom počte aktívnych vlakien procesora I5-8250U od spoločnosti Intel. Z meraní vyplýva, že výpočtovo najrýchlejsia implementácia je implementácia napísaná v jazyku Rust, ktorá je rýchlejsia ako naša demonštračná prepojená implementácia rust_py_blaKE3 v priemere o približne 6.7 %. Je nutné podotknúť, že naša domonštrašná implementácia nie je optimalizovaná z hľadiska spotreby operačnej pamäte a zároveň poskytuje iba základný hašovací mód funkcie Blake3, teda mód *hash(input)*. Rozšírenie našej demonštračnej implementácie o všetky hašovacie módy funkcie Blake3, spolu s verifikáciou streamovaných dát považujem za vhodné pre ďalšie štúdium. Nástroj TurboSFV podporuje hašovanie dát hašovacou funkciou Blake3 s využitím paralelizmu a to od verzie 9.20. Toto hašovanie sa však vzťahuje na počet dostupných jadier CPU, nie na počet vlakien ako to bolo u Rust a rust_py_blaKE3 implementácií [41]. Výsledky meraní paralelného hašovania dát s využitím hašovacej funkcie Blake3 nástrojom TurboSFV sú zobrazené v Tabuľke 5.7. Referenčná implementácia hašovacej funkcie Blake3, napísaná v programovačom jazyku Python, sa javí ako najpomalsia. Táto neoptimalizovaná implementácia slúži iba na edukačné účely a nepodporuje žiadnu formu paralelizmu.

Tabuľka 5.8: Porovnanie vybraných konvenčných hašovacich funkcií, podporovaných kryptografickým modulom Cryptography s prepojenou hašovacou funkciou Blake3 na procesore I5-8250U

Porovnanie vybraných hašovacich funkcií			
Súbor	SHA-256	SHA3-256	rust_py_blaKE3
20 MB	0,0958 s	0,1177 s	0,0051 s
50 MB	0,2327 s	0,2902 s	0,0094 s
100 MB	0,4807 s	0,5901 s	0,016 s
500 MB	2,3168 s	2,8908 s	0,0696 s
1,5 GB	7,0825 s	8,9933 s	0,2061 s
2 GB	10,0687 s	12,0961 s	0,2934 s

Tabuľka 5.8 zobrazuje porovnanie konvenčných hašovacích funkcií SHA-256 a SHA3-256 s prepojenou implementáciou hašovacej funkcie Blake3. Z Tabuľky 5.5 a 5.6 je vidieť, že kryptografický modul Cryptography dosahuje najlepšie vý-

sledky z hľadiska rýchlosťi vykonávania algoritmov pre jazyk Python, a preto je vhodné vykonať porovnanie konvenčných hašovacích funkcií s využitím práve tohto modulu. Z Tabuľky 5.8 je vidieť, že prepojená implementácia hašovacej funkcie Blake3, využívajúca všetkých 8 dostupných vlákien, je výrazne rýchlejšia ako konvenčné hašovacie funkcie a to niekoľko násobne. Na príklade hašovania súboru o veľkosti 20 MB je prepojená implementácia funkcie Blake3 v priemere 21-krát rýchlejšia ako konvenčné hašovacie funkcie. V prípade hašovania väčších súborov tento rozdiel narastá a na príklade hašovania súboru o veľkosti 2 GB je tento rozdiel v priemere až 38 násobný.

6 Záver

Prudkým rozvojom informačno-komunikačných technológií začalo vznikať čoraz viac aplikácií, ktoré spracúvajú, uchovávajú alebo prenášajú často citlivé, osobné alebo tajné údaje. Bezpečné zaobchádzanie s údajmi rieši kryptografia. Vzhľadom na fakt, že programovací jazyk Python je jedným z najpopulárnejších programovačích jazykov a jeho popularita stále narastá, je voľba tohto programovacieho jazyka zaujímavou v oblasti vývoja takýchto aplikácií. Teda má zmysel, uvažovať o kryptografii v programovacom jazyku Python, čo je téma, ktorej sa venuje táto bakalárka práca.

Teoretická časť tejto bakalárskej práce má za úlohu detailne opísat nový kryptografický stavebný blok, ktorým je hašovacia funkcia Blake3. Hašovacia funkcia Blake3 ma obrovský potenciál vo svete kryptografie, pretože vďaka Merklovej stormovej hašovacej štruktúre a efektívnej implementácii je funkcia Blake3 niekoľko násobne rýchlejšia ako súčasné hašovacie funkcie. V tejto práci sme sa venovali problematike Merklových stromov, kde sme opísali ich štruktúru, využitie a výhody, ktoré nám táto konštrukcia ponúka. Následne sme podrobne opísali princípy hašovacej funkcie Blake3 ako aj jej využitie a výhody v porovnaní s konvenčnými hašovacími funkciami. Stručne sme naznačili využitie SIMD inštrukcií a parallelizmus vo funkcií Blake3.

Úlohou prvej praktickej časti bakalárskej práce je oboznámiť sa s programovacím jazykom Python a základnými operáciami, ktoré je potrebné ovládať v oblasti kryptografie. Predstavili sme si najpopulárnejšie a najrobustnejšie kryptografické Python moduly, pre ktoré sme pripravili sadu demonštračných aplikácií. Cieľom demonštračných aplikácií bolo otestovať dané kryptografické moduly z hľadiska rýchlosťi vykonávania kryptografických úkonov ale aj z hľadiska rozsahu pod-

porovaných kryptografických algoritmov a protokolov.

V druhej praktickej časti tejto práce sme sa venovali premosteniu hašovacej funkcie Blake3 do jazyka Python, pretože optimalizovaná a paralizovateľná implementácia funkcie Blake3 je napísaná v jazyku Rust. 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 rýchlosťi a bezpečnosti mohol predstavovať alternatívu programovacieho jazyka C resp. C++. Vývojári jazyka Rust propagujú jeho schopnosť pomáhať vývojárom vytvárať rýchle, bezpečné aplikácie a tvrdia, že Rust zabraňuje chybám segmentácie a zaručuje bezpečnosť vlákien pri vývoji aplikácií. Programovací jazyk Rust neobsahuje garbage collector, runtime alebo manuálnu správu pamäte, čo súvisí s jeho rýchlosťou a bezpečnosťou. Bezpečnosť Rustu je založená na troch striktných „ownership-based“ pravidlách, na ktoré dohliada prekladač. Prvé pravidlo hovorí o tom, že každá hodnota v Ruste má svoju premennú, ktorá je jej vlastníkom. Druhé pravidlo hovorí o tom, že každá hodnota môže mať práve jedného vlastníka a tretie pravidlo hovorí, že hodnota bude zahodená, keď sa jej vlastník dostane mimo rozsah (mimo blok) kódu [42][43]. Práve rýchlosť, bezpečnosť a podpora mnohých knižníc jazyka Rust viedla tvorcov hašovacej funkcie Blake3 k využitiu tohto jazyka pre vývoj Blaku3. Tieto výhody sme využili v bakalárskej práce pri prepojení hašovacej funkcie Blake3 s jazykom Python. V práci sme si vysvetlili princíp mapovania objektov jazyka Rust s jazykom Python. Vytvorili sme dynamický Python modul, ktorý sme neskôr otestovali v našej demonštračnej aplikácii a vykonali merania so zamraním na rýchlosť tejto implementácie. Rýchlosné meracie testy boli vykonané na rôznych implementáciach funkcie Blake3. Hoci naša implementácia je o čosi pomalšia ako optimalizovaná Rust implementácia, jej výhodou je zachovanie komfortu jazyka Python pri hašovaní dát.

Naša demonštračná implementácia hašovacej funkcie Blake3 má niekolko nevýhod, t.j. väčšia spotreba pamäte, či neúplná podpora všetkých funkcionalít, ktoré ponúka funkcia Blake3 ako napríklad verifikácia streamovaných dát. Optimalizáciu a úpravu zdrojového kódu je možné dosiahnuť s využitím vhodných programátorských techník a takéto vylepšenie považujem za vhodné na ďalšie štúdium.

A Práca s binárnymi dátami

A.1 Základné operácie s bajtmi

Bytes-literal je jedným z možných vyjadrení binárnych dát. Ide o formát, ktorý je štruktúrovo podobný retazcu (string), avšak obsahuje prefix `b`. Formát bytes-literal podporuje iba základných 127 ASCII znakov.

```
bytes_literal = b"ABCD"  
print("bytes_literal: ",bytes_literal)
```

V prípade, ak naše dáta obsahujú znaky, ktoré nie sú z podporovaného rozsahu 127 ASCII znakov, je vhodné použiť vstavanú metódu Python retazca `encode()`. Táto metóda podporuje rôzne kódovacie sady ako napríklad predvolenú sadu utf-8, či utf-16. Analogicky je možné použiť metódu `decode()` pre dekódovanie znakov.

```
string="ABČĎ"  
print("string.encode(): ",string.encode())
```

Bajt je v Python tzv. immutable dátovy typ. To znamená, že jeho hodnotu nie je možné meniť počas písania kódu. Zdrojový kód nižsie je domonštráciou snahy o zmenu premennej `change_bytes`, ktorá pôvodne obsahuje 4 nulové bajty. Pri spustení kódu, nám Python interpreter vypíše chybové hlásenie.

```
change_byte = b'0000'  
change_byte = bytes_zero[0:1]=b'2'  
print("bytes_zero: ",change_byte)
```

V jazyku python vieme bajty generovať a kopírovať, s využitím vstavanej funkcie `bytes()`. Táto funkcia podporuje generovanie nulových bajtov, inkrementujúcich bajtov (vstupný parameter je funkcia `range()`) alebo kopírovanie bajtov.

```

# generovanie nulovych bajtov
bytes_zero = bytes(10)
print("bytes_zero: ",bytes_zero)

# generovanie inkrementujucich bajtov
bytes_inc = bytes(range(20))
print("bytes_inc: ",bytes_inc)

# kopirovanie bajtov
bytes_copied = bytes(bytes_inc)
print("bytes_copied: ",bytes_copied)

```

V kryptografii sa môžeme stretnúť s potrebou konverzie reťazca hexadecimálnych znakov na bajty a naopak. Takúto konverziu je v Pythone možné vykonať využitím metódy `fromhex()` a `hex()`. Podmienkou je, že hexadecimálne znaky musia byť vyjadrené vždy po dvojiciach. To znamená, že ak by sme chceli zapísať znak vyjadrujúci číslo 4, správny formát zápisu do dvojice znakov by bol `0x04` nie `0x4`. Takúto konverziu je možné vykonať následovne.

```

#hex_string po dvojiciach (hex znaky, max. ff)
hex_to_bytes_literal=bytes.fromhex('feac')
print("bytes_to_hex: ",hex_to_bytes_literal)
# feac -> b'\xfe\xac'

bytes_literal_to_hex = b'\xfe\xac'.hex()
print("bytes_literal_to_hex: ",bytes_literal_to_hex)
# b'\xfe\xac' -> feac

```

Python podporuje dátový typ byte-array, ktorý veľmi podobný typu `bytes` a je s ním možné vykonávať rovnaké operácie ako s typom `bytes`. Výhodou dátového typu byte-array oproti typu `bytes` je to, že je tzv. `mutable`, teda je možné meniť jeho obsah v kóde.

```

bytearray_zero = bytearray(10)
# bytearray je mutable a je možné meniť jeho obsah
bytearray_zero[1:2]=b'\x01'
print(bytearray_zero)

```

S typom byte-array je možné pracovať rovnako ako s typom `bytes`, teda možeme generovať, kopírovať a formovať bajty z hexa-reťazca.

```

# generovanie nulovych byte-arrays
bytearray_zero = bytearray(10)
print(bytearray_zero)

```

```

# generovanie inkrement. byte-arrays
bytearray_inc = bytearray(range(10))
print(bytearray_inc)

#create bytearray from hex-string
hex_string = '2Ef0 f1f0'
print(bytearray.fromhex(hex_string))
# 2ef0f1f0 -> b'.\xf0\xf1\xf0'

# get hex from bytearray
temp_bytearray = bytearray(b'.\xf0\xf1\xf0')
print(temp_bytearray.hex())
# b'.\xf0\xf1\xf0' -> 2ef0f1f0

```

A.2 Konverzia dátovych typov Bytes-Bytearray-List

V tejto sekcií sa zameriame na základne konverzné techniky medzi dátovými typmi jazyka Python ako sú Bytes, Byte-array a List. Tieto konverzné techniky možno nájsť aj v našich demonštračných aplikáciach vytvorených pre testovanie kryptografických Python modulov.

Konverzia List ↔ Byte-array:

```

# list na byte-array
test_list = [116, 101, 115, 116]
test_byte_array = bytearray(test_list)
print("test_byte_array: ", test_byte_array)

# bytearray na list
test_byte_array = list(bytearray(test_list))
print(test_byte_array)

```

Konverzia List ↔ Bytes:

```

# list na bytes
bytes_from_list = bytes(list(range(20)))
print(bytes_from_list)

# bytes na list
list_form_bytes = list(bytes_from_list)
print(list_form_bytes)

```

Konverzia Bytesarray ↔ Bytes:

```
get_bytearray_from_bytes = bytearray(b'abcs')
print(get_bytearray_from_bytes)

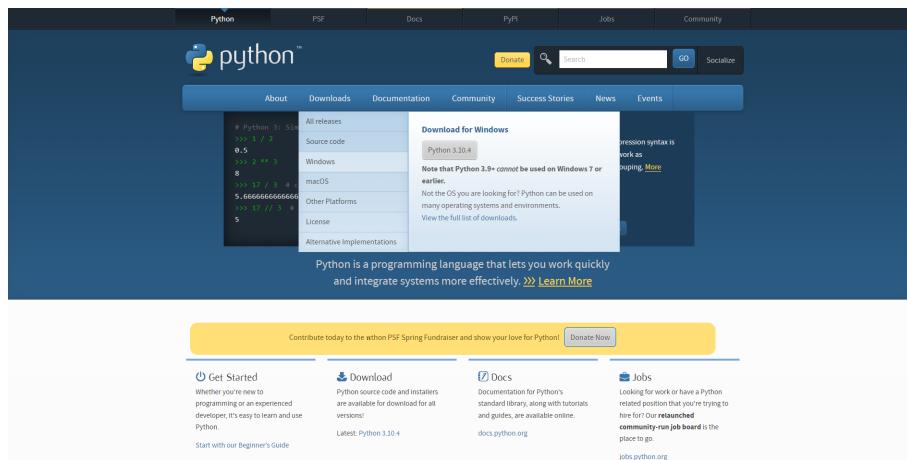
get_bytes_from_bytearray = bytes(bytearray(b'abcs'))
print(get_bytes_from_bytearray)
```

B Inštalácia jazyka Python

Táto príloha obsahuje detailný návod na inštaláciu programovacieho jazyka Python pre operačný systém Windows. Súčasťou prílohy je aj stručný návod inštalácie a využitia viacerých verzií programovacieho jazyka Python.

B.1 Inštalácia jazyka Python pre OS Windows

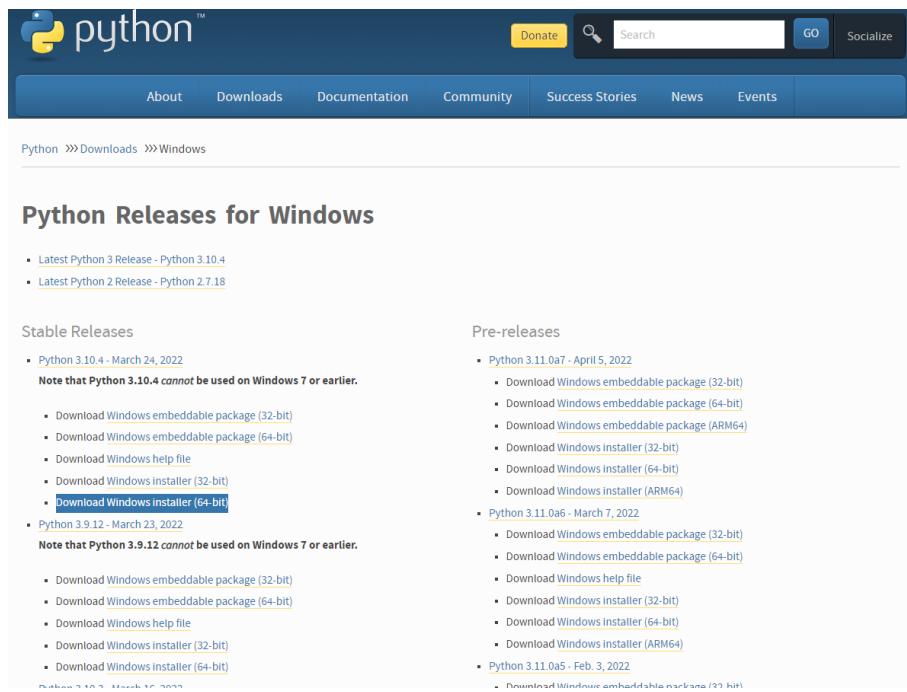
Inštalácia programovacieho jazyka Python pre OS Windows je realizovaná prostredníctvom inštalačného balíčka, ktorý je možný stiahnuť z oficiálnej stránky jazyka Python (<https://www.python.org/>) v sekcií **Downloads** (viď. Obr. B.1).



Obr. B.1: Oficiálna webová stránka jazyka Python

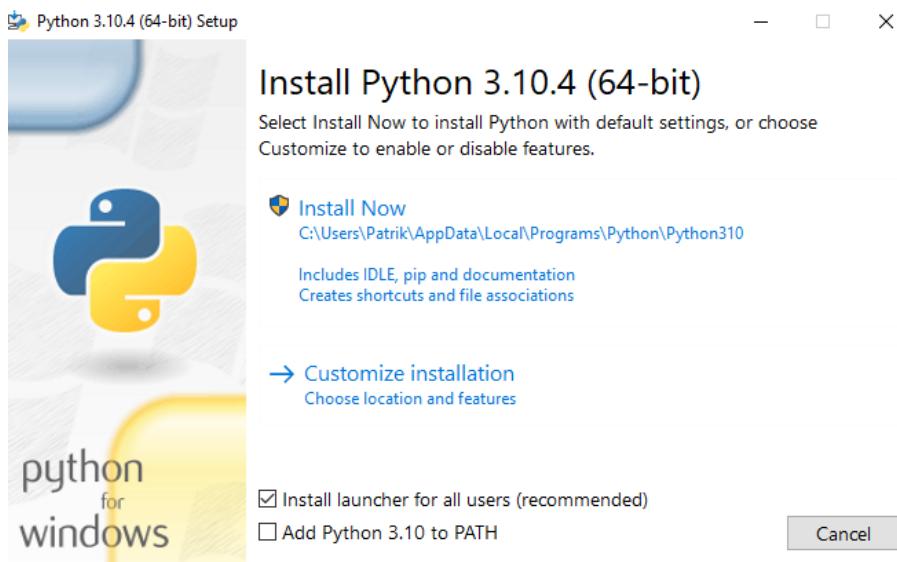
Následne vyberieme vhodnú verziu Pythonu a stiahneme inštalačný balíček (Obr. B.2). V tomto prípade si nainštalujeme aktuálne najnovšiu dostupnú verziu, teda verziu 3.10.4.

Po stiahnutí inštalačného balíčka spustíme inštaláciu Pythonu dvojklikom na inšta-



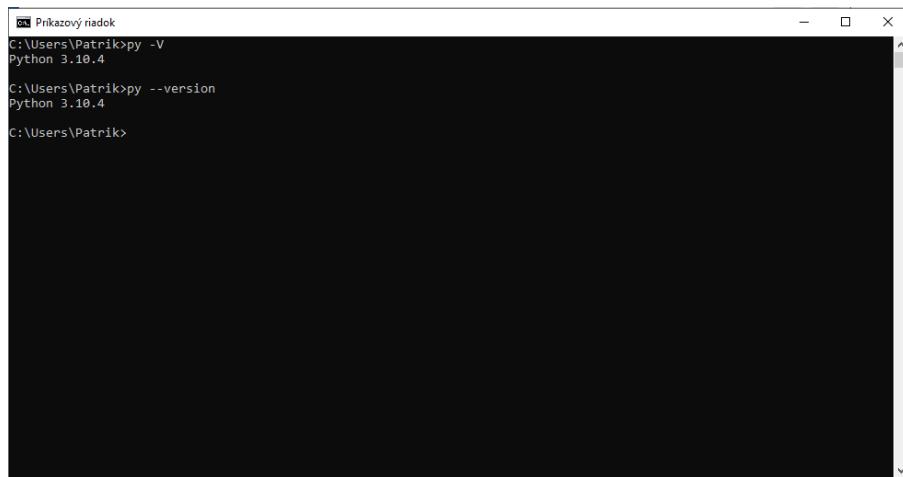
Obr. B.2: Výber požadovanej verzie jazyka Python

lačný balíček. Inštaláciu môžeme ponechať v preferovanom predvolenom nastavení



Obr. B.3: Proces inštalácie jazyka Python s využitím inštalačného balíčka

a pokračujeme s výberom možnosti **Install Now**, čo možno pozorovať na Obr. B.3. Touto voľbou sa nám nainštaluje verzia Pythonu 3.10.4 do predvoleného adresára, v tomto prípade C:\Users\Patrik\AppData\Local\Programs\Python\Python310. Správnosť inštalácie môžeme otestovať terminálovým príkazom `py -version` resp. `py -V` (vid. Obr. B.4).



```
Príkazový riadok
C:\Users\Patrik>py -V
Python 3.10.4
C:\Users\Patrik>py --version
Python 3.10.4
C:\Users\Patrik>
```

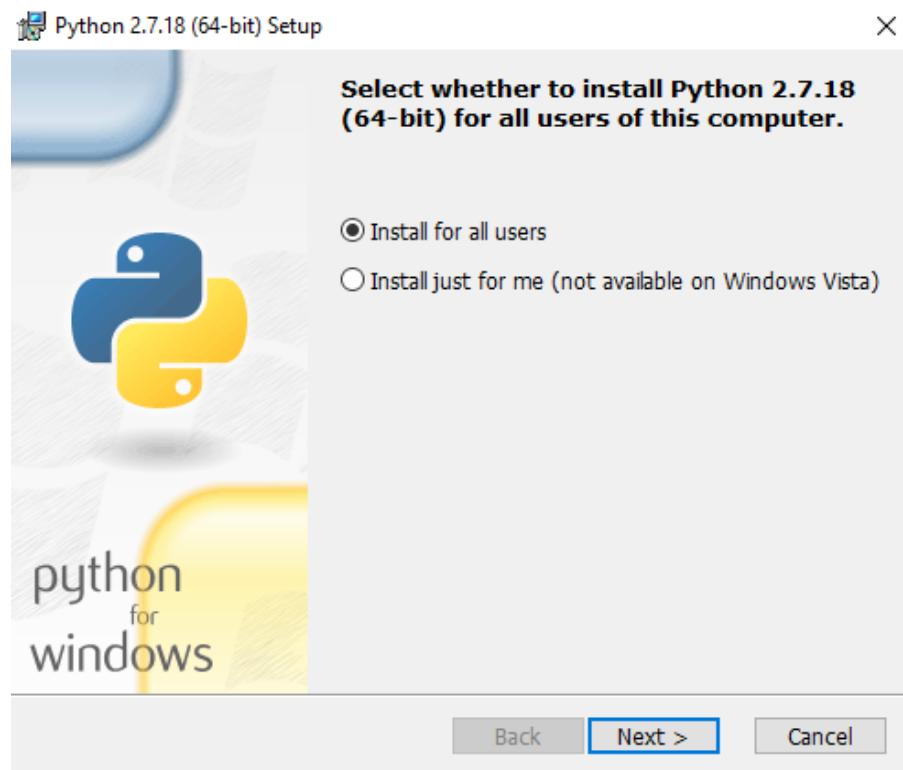
Obr. B.4: Výpis nainštalovanej verzie Pythonu v príkazovom riadku

B.2 Inštalácia a práca s viacerými verziami jazyka Python

Inštalácia viacerých verzií jazyka Python na jednom zariadení môže byť problematická. V tejto časti prílohy si uvedieme niekoľko metód inštalácie a použitia viacerých verzií jazyka Python pre OS Windows.

B.2.1 Manuálna inštalácia

Prvým, najrýchlejším a zároveň najjednoduchším spôsobom je manuálna inštalácia viacerých verzií jazyka Python. V predošej časti tejto prílohy sme si nainštalovali Python 3.10.4. V tejto časti demonštrujeme inštaláciu Pythonu 2.7.18. Inštalácia prebehne podobný spôsobom ako pri inštalácii Python 3.10.4. V prvej fáze (Obr. B.5) inštalácie ponecháme predvolené nastavenia a zvolíme možnosť **Next>**. Následne vyberieme a zvolíme cestu pre inštaláciu Python 2.7.18. Cestu zvolíme rovankú akú sme volili pri inštalácii Python 3.10.4, pričom koncový adresár premenujeme na Python27, teda výsledná cesta by mohla vyzeráť takto, C:\Users\Patrik\AppData\Local\Pro-grams\Python\Python27. Inštaláciu dokon-

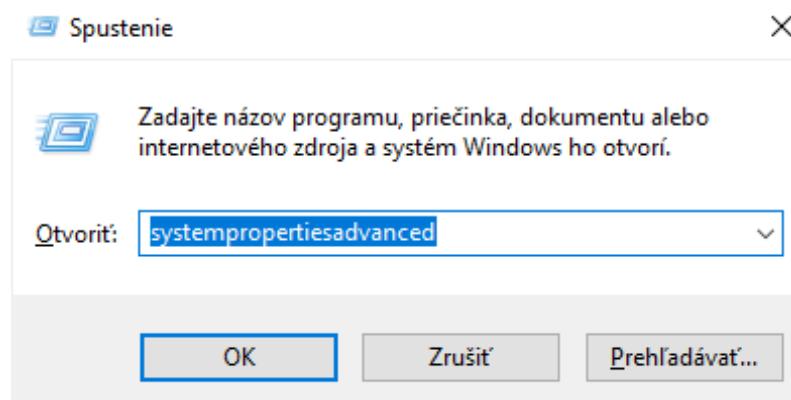


Obr. B.5: Inštalácia jazyka Python 2.7.18 k existujúcej verzii 3.10.4

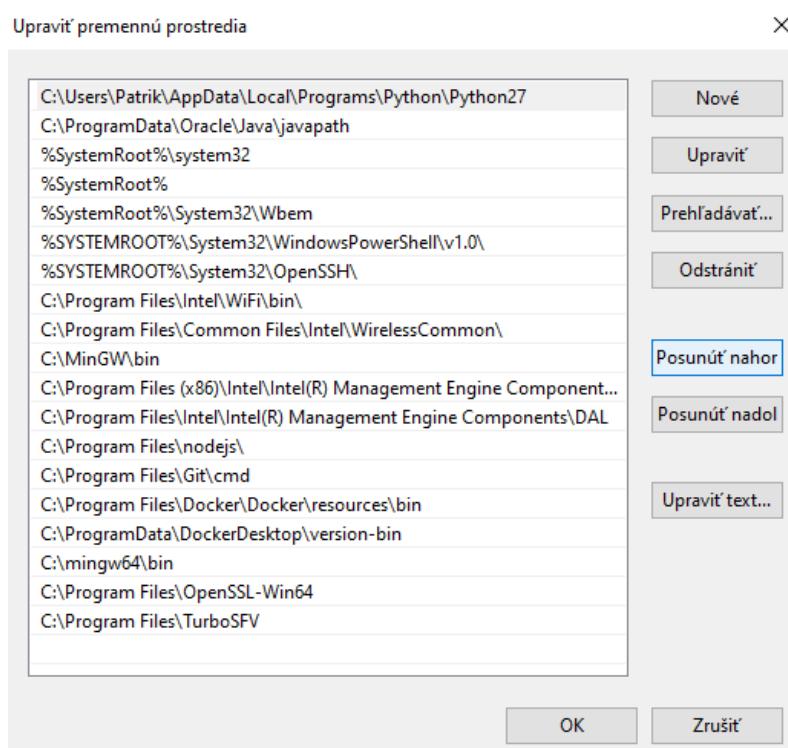
číme výberom možnosti **Next >**.

Následne je nutné zvoliť (prepnúť) používanú verziu Pythonu. Prvým krokom, ktorý je potrebný vykonať je pridanie cesty C:\Users\Patrik\AppData\Local\Programs\Python\Python27 do systémových premenných prostredia. Pridať cestu do systémových premenných prostredia môžno realizovať následovne:

1. Pomocou klávesovej skratky **Win+R** vyhľadáme názov **systempropertiesadvanced**.

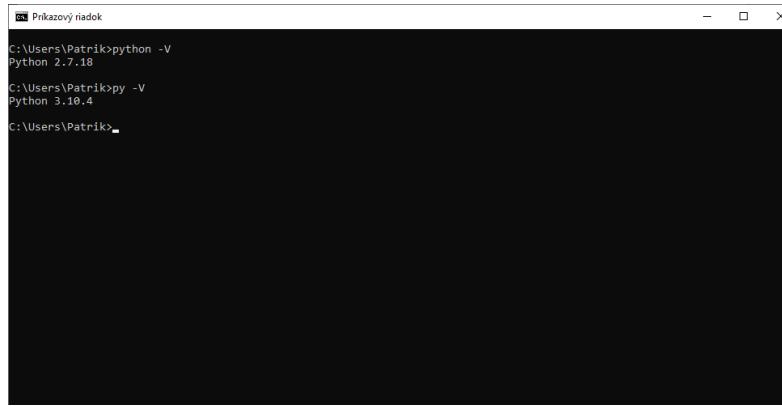


2. Zvolíme možnosť **Premenné prostredia**. V časti **Systémové premenné** zvolíme premennú **Path** a následne vyberieme možnosť **upraviť**.
3. Pridáme cestu do systémových premenných pomocou tlačidla **Nové**, kde vložíme cestu C:\Users\Patrik\AppData\Local\Programs\Python\Python27.
4. Je dôležité, aby naša novo-pridaná cesta k Pythonu 2.7.18 bola ako prvá v poradí, tým zabezpečíme „prepnutie“ verzie. Následne uložíme vykonané zmeny a spustíme príkazový riadok.



5. V príkazovom riadku spustíme príkaz `python -V`. Je veľmi dôležité použiť príkaz **python** a nie **py**, pretože príkaz **py** pracuje na globálne predvolenej verzii Python 3.10.4. Súbory, ktoré chceme spúštať z verzie, ktorú sme si zvolili ako prvú v poradí (v systémových premenných prostredia) budeme taktiež spúštať príkazom (`python`).

Takúto manuálnu inštaláciu viacerých verzií jazyka Python je možné vykonať aj pomocou video návodu (<https://www.youtube.com/watch?v=ggRsauJcEyE>).



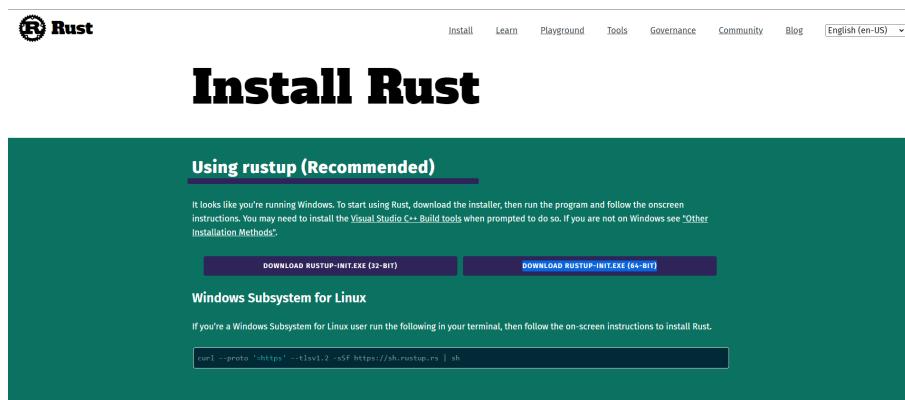
```
C:\Users\Patrik>python -V
Python 2.7.18
C:\Users\Patrik>py -V
Python 3.10.4
C:\Users\Patrik>
```

B.2.2 Inštalácia viacerých verzií jazyka Python pomocou nástroja pyenv

Inštaláciu viacerých verzií jazyka Python je možné vykonať oveľa komfortnejším a flexibilnejším spôsobom, pomocou nástrojov pyenv alebo pyenv-virtualenv. V prípade ak by sme chceli nainštalovať viacero verzií jazyka Python 3.x.x je vhodné zvoliť nástroj pyenv, ktorý je pre OS Windows dostupný v github repozitári (<https://github.com/pyenv-win/pyenv-win>). Nainštalovanie a práca s nástrojom pyenv je detailne opísaná v návode (<https://k0nze.dev/posts/install-pyenv-venv-vscode/>), resp. video návode (<https://k0nze.dev/posts/install-pyenv-venv-vscode/>). V prípade, ak chceme pracovať aj s verziou Python 2.x.x je lepšie zvážiť nástroj pyenv-virtualenv (<https://www.freecodecamp.org/news/manage-multiple-python-versions-and-virtual-environments-venv-pyenv-pyvenv-a29fb00c296f/>).

C Inštalácia jazyka Rust

Inštalácia programovacieho jazyka Rust pre operačný systém Windows pozostáva z niekoľkých krokov. Prvým krokom je stiahnutie inštalačného balíčka pre jazyk Rust (<https://www.rust-lang.org/tools/install>). Inštalácia vyžaduje prítomnosť softvéru **C++ build tools for Visual Studio 2013** alebo akúkoľvek novšiu verziu , ktorý je dostupný na linke: <https://visualstudio.microsoft.com/visual-cpp-build-tools/>. Inštalácia programovacieho jazyka Rust pre operačný systém Windows pozostáva z niekoľkých krokov. Prvým krokom je stiahnutie inštalačného balíčka pre jazyk Rust (<https://www.rust-lang.org/tools/install>). Inštalácia vyžaduje prítomnosť softvéru **C++ build tools for Visual Studio 2013** alebo akúkoľvek novšiu verziu, ktorý je dostupný na linke: <https://visualstudio.microsoft.com/visual-cpp-build-tools/>. Po



Obr. C.1: Inštalačný balíček jazyka Rust dostupný na oficiálnej webovej stránke Rustu

stiahnutí a nainštalovaní **C++ build tools for Visual Studio 2013**, môžeme pokračovať v inštalácii samotného Rustu. Dvojklikom na inštalačný balíček Rustu spusťime terminálový inštalačný proces. Obr. C.2 obsahuje výpis z terminálu, v ktorom si môžeme zvoliť jednu z 3 možnosti. Pokračujeme prednastaveným (defaultným)

```
The Cargo home directory located at:
C:\Users\Patrik\.cargo

This can be modified with the CARGO_HOME environment variable.

The cargo, rustc, rustup and other commands will be added to
Cargo's bin directory, located at:
C:\Users\Patrik\.cargo\bin

This path will then be added to your PATH environment variable by
modifying the HKEY_CURRENT_USER\Environment\PATH registry key.

You can uninstall at any time with rustup self uninstall and
these changes will be reverted.

Current installation options:

    default host triple: x86_64-pc-windows-msvc
        default toolchain: stable (default)
            profile: default
            modify PATH variable: yes

1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
>
```

Obr. C.2: Terminálový výpis pri inštalácii jazyka Rust

spôsobom a vyberieme možnosť 1. Zvolením tejto možnosti súhlasíme s predvolenými nastaveniami a začneme proces inštalácie jazyka Rust a jeho závislosti ako napríklad balíčkový manažér Cargo.

V prípade, že po spustení inštalačného balíčka Rust dostaneme terminálový výpis, ktorý je naznačený na Obr. C.3, je nutné stiahnuť a nainštalovať softvér **C++ build tools for Visual Studio 2013** (<https://visualstudio.microsoft.com/visual-cpp-build-tools/>). Obr. C.3 informuje o potrebe softvéru **Microsoft C++ build tools**

```
Rust Visual C++ prerequisites

Rust requires the Microsoft C++ build tools for Visual Studio 2013 or
later, but they don't seem to be installed.

The easiest way to acquire the build tools is by installing Microsoft
Visual C++ Build Tools 2019 which provides just the Visual C++ build
tools:
https://visualstudio.microsoft.com/visual-cpp-build-tools/

Please ensure the Windows 10 SDK and the English language pack components
are included when installing the Visual C++ Build Tools.

Alternately, you can install Visual Studio 2019, Visual Studio 2017,
Visual Studio 2015, or Visual Studio 2013 and during install select
the "C++ tools":
https://visualstudio.microsoft.com/downloads/

Install the C++ build tools before proceeding.

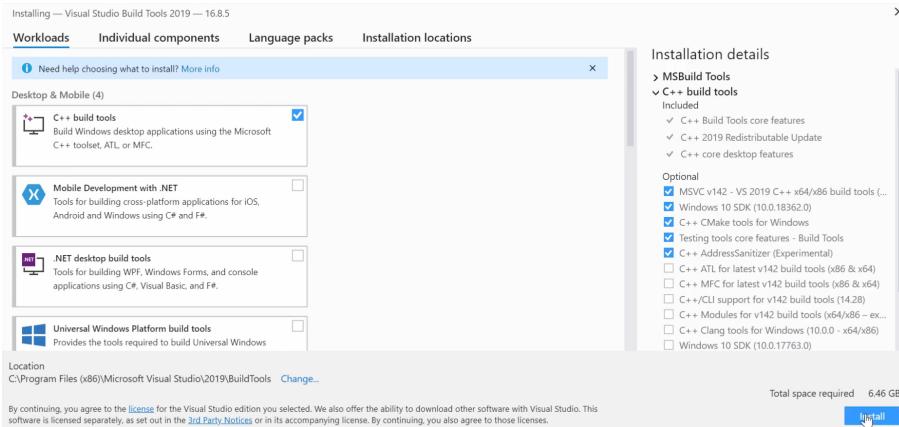
If you will be targeting the GNU ABI or otherwise know what you are
doing then it is fine to continue installation without the build
tools, but otherwise, install the C++ build tools before proceeding.

Continue? (Y/n) =
```

Obr. C.3: Terminálový výpis, ktorý informuje o tom, že sovtvér Microsoft C++ build tool, nie je dostupný

pre inštaláciu jazyka Rust. V tomto prípade je možné inštaláciu zrušiť a pokračovať s nainštalovaním **Microsoft C++ build tools**. Pri inštalácii **C++ build tools**

for Visual Studio 2013 je nutné zvoliť inštaláciu **C++ build tools**. Po úspešnom



Obr. C.4: Priebeh inštalácie softvéru Microsoft C++ build tools

nainštalovaní **C++ build tools** je možné opäť spustiť inštalačný balíček programovacieho jazyka Rust. Po spravnom nainštalovaní **C++ build tools** a opäťovnom spustení inštalácie jazyka Rust, dostaneme terminalový výpis zhodný s Obr. C.2, pri ktorom násedne zvolíme predvolenú možnosť inštalácie, teda zvolíme možnosť 1.

Literatúra

1. KUHLMAN, Dave. *A Python Book: Beginning Python, Advanced Python, and Python Exercises*. [online] [cit. 2022-01-28]. Dostupné z: https://web.archive.org/web/20120623165941/http://cutter.rexx.com/~dkuhlman/python_book_01.html.
2. BARANY, Gergö. *Python Interpreter Performance Deconstructed*. [online] [cit. 2022-01-29]. Dostupné z: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1083.2971&rep=rep1&type=pdf>.
3. CHUN, Wesley J. *Core Python Programming*. USA: Prentice Hall PTR, 2006. ISBN 978-01-3226-993-3.
4. FOURMENT, Mathieu; GILLINGS, Michael R. *A comparison of common programming languages used in bioinformatics*. [online] [cit. 2022-02-02]. Dostupné z: <https://link.springer.com/content/pdf/10.1186/1471-2105-9-82.pdf>.
5. TABBA, Fuad. *Adding Concurrency in Python Using a Commercial Processor's Hardware Transactional Memory Support*. [online] [cit. 2022-02-02]. Dostupné z: https://web.archive.org/web/20130115061039id_/http://www.cs.auckland.ac.nz/~fuad/parpycan.pdf.
6. JARAKACA. *Integrating Python With Other Languages*. [online] [cit. 2022-02-02]. Dostupné z: <https://wiki.python.org/moin/IntegratingPythonWithOtherLanguages>.
7. BRAY, Shannon W. *Practical Cryptography in Python: Learning Correct Cryptography by Example*. New York, 2019. 1. ISBN 9781484248997.

8. GUIDO VAN ROSSUM, and the Python development team. *The Python Library Reference*. [online] [cit. 2022-01-28]. Dostupné z: <https://docs.python.org/3/download.html>.
9. MICROSOFT. *System Security Cryptography Namespace*. [online] [cit. 2022-02-04]. Dostupné z: <https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography?view=net-6.0>.
10. SITIARIK, Patrik. *Elektronická Príručka Práce So Súbormi, Poliami A Zoznamami V Jazyku Python*. [online] [cit. 2022-02-07]. Dostupné z: <https://opac.crzp.sk/?fn=detailBiblioForm&sid=FB09C562C25683D6609B6652F34B>.
11. PANKAJ. *Reading Large Text Files in Python*. [online] [cit. 2022-02-08]. Dostupné z: <https://www.journaldev.com/32059/read-large-text-files-in-python>.
12. BURDA, Karel. *Kryptografia okolo nás*. Praha, 2019. 1. ISBN 9788088168492. Tiež dostupné na: https://web2.mlp.cz/koweb/00/04/52/75/91/kryptografia_okolo_nas.pdf.
13. BEIMEL, Amos. *Secret-Sharing Schemes: A Survey*. [online] [cit. 2022-02-09]. Dostupné z: <https://www.turbosfv.com/Blog?q=2&p0=68>.
14. CHEN, Lily et al. Recommendation for key derivation using pseudorandom functions. *NIST special publication*. 2008, roč. 800, s. 108. Tiež dostupné na: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-108.pdf>.
15. PYCA/CRYPTOGRAPHY. *Frequently asked questions*. [online] [cit. 2022-02-11]. Dostupné z: <https://cryptography.io/en/latest/faq/?highlight=side-channel#why-use-cryptography>.
16. O'CONNOR, Jack. *The Bao Spec*. [online] [cit. 2022-02-17]. Dostupné z: https://github.com/oconnor663/bao/blob/master/docs/spec_0.9.1.md.
17. AUMASSON, Jean-Philippe; NEVES, Samuel; WILCOX-O'HEARN, Zooko; WINNERLEIN, Christian. *BLAKE2: simpler, smaller, fast as MD5*. [online] [cit. 2022-02-17]. Dostupné z: <https://www.blake2.net/blake2.pdf>.

18. MACHADO, Armando; DOMINGUES, J. Augusto. *Suacoin: Ecological friendly proof-of-work cryptocurrency*. [online] [cit. 2022-02-10]. Dostupné z: <https://www.suacoin.com/Suacoin.pdf>.
19. O'CONNOR, Jack; AUMASSON, Jean-Philippe; NEVES, Samuel; WILCOX-O'HEARN, Zooko. *Blake3*. [online] [cit. 2022-02-10]. Dostupné z: <https://github.com/BLAKE3-team/BLAKE3#adoption--deployment>.
20. PENARD, Wouter; WERKHOVEN, Tim van. *On the Secure Hash Algorithm family*. [online] [cit. 2022-02-11]. Dostupné z: [https://blog.infocruncher.com/resources/ethereum-whitepaper-annotated/On%20the%20Secure%20Hash%20Algorithm%20family%20\(2008\).pdf](https://blog.infocruncher.com/resources/ethereum-whitepaper-annotated/On%20the%20Secure%20Hash%20Algorithm%20family%20(2008).pdf).
21. SZYDLO, Michael. *Merkle Tree Traversal in Log Space and Time*. [online] [cit. 2022-02-12]. Dostupné z: https://link.springer.com/content/pdf/10.1007/978-3-540-24676-3_32.pdf.
22. ATIGHEHCHI, Kevin; ROLLAND, Robert. *Optimization of Tree Modes for Parallel Hash Functions: A Case Study*. [online] [cit. 2022-02-12]. Dostupné z: <https://arxiv.org/pdf/1512.05864.pdf>.
23. CHAMPINE, Luke. *Streaming Merkle Proofs within Binary Numeral Trees*. [online] [cit. 2022-02-12]. Dostupné z: <https://eprint.iacr.org/2021/038.pdf>.
24. MERKLE, Ralph C. *METHOD OF PROVIDING DIGITAL SIGNATURES*. U.S. Patent 4 309 569, jan. 1982.
25. DAHLBERG, Rasmus; PULLS, Tobias; PEETERS, Roel. *Efficient Sparse Merkle Trees Caching Strategies and Secure (Non-)Membership Proofs*. [online] [cit. 2022-02-12]. Dostupné z: <https://eprint.iacr.org/2016/683.pdf>.
26. CHAPWESKE, J. *Tree Hash EXchange format (THEX)*. 2003. Dostupné z: <https://adc.sourceforge.io/draft-jchapweske-thex-02.html>.
27. SCHRODER, Dominique; SCHRODER, Dominique. *Verifiable Data Streaming*. [online] [cit. 2022-02-12]. Dostupné z: <https://eprint.iacr.org/2013/038.pdf>.

28. CASTELLON, Cesar; ROY, Swapnoneel; KREIDL, Patrick; DUTTA, Ayan; BÖLÖNI, Ladislau. Energy Efficient Merkle Trees for Blockchains. In: *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 2021, s. 1093–1099. Dostupné z DOI: [10.1109/TrustCom53373.2021.00149](https://doi.org/10.1109/TrustCom53373.2021.00149).
29. MOHAN, Arun Prasad; MOHAMEDASFAK, R.; GLADSTON, Angelin. Merkle Tree and Blockchain-Based Cloud Data Auditing. *Int. J. Cloud Appl. Comput.* 2020, roč. 10, s. 54–66.
30. CRYPTOPEDIA. *Merkle Trees and Merkle Roots Help Make Blockchains Possible*. [online] [cit. 2022-02-09]. Dostupné z: <https://www.gemini.com/cryptopedia/merkle-tree-blockchain-merkle-root#section-merkle-trees-in-bitcoin-and-beyond>.
31. O'CONNOR, Jack; AUMASSON, Jean-Philippe; NEVES, Samuel; WILCOX-O'HEARN, Zooko. *BLAKE3 one function, fast everywhere*. 2020.
32. O'CONNOR, Jack. *Blake3/asm*. [online] [cit. 2022-02-10]. Dostupné z: [http://github.com/BLAKE3-team/BLAKE3/blob/e17743e8fdf2845be6dc85ad339bf45feeefc564/asm/asm.py#L331-L370](https://github.com/BLAKE3-team/BLAKE3/blob/e17743e8fdf2845be6dc85ad339bf45feeefc564/asm/asm.py#L331-L370).
33. GEPNER, Paweł. *Using Avx2 Instruction Set To Increase Performance Of High Performance Computing Code*. [online] [cit. 2022-02-17]. Dostupné z: https://147.213.75.178/ojs/index.php/cai/article/viewFile/202017_5_1001/851.
34. INTEL. *Intel AVX-512 Instructions*. [online] [cit. 2022-02-17]. Dostupné z: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html?wapkw=avx>.
35. WATANABE, Hiroshi; NAKAGAWA, Koh M. SIMD vectorization for the Lennard-Jones potential with AVX2 and AVX-512 instructions. *Computer Physics Communications*. 2019, roč. 237, s. 1–7. Dostupné z DOI: <https://doi.org/10.1016/j.cpc.2018.10.028>.
36. GUNSING, Aldo. *Block-Cipher-Based Tree Hashing*. [online] [cit. 2022-02-22]. Dostupné z: <https://eprint.iacr.org/2022/283.pdf>.

37. O'CONNOR, Jack. *Blake3/c/Security Notes*. [online] [cit. 2022-02-10]. Dostupné z: <https://github.com/BLAKE3-team/BLAKE3/tree/master/c#security-notes>.
38. BLANDY, J.; ORENDORFF, J. *Programming Rust: Fast, Safe Systems Development*. O'Reilly Media, 2017. ISBN 9781491927281.
39. ZELEŇÁK, Patrik. *CryptographyInPythonThesis*. [online] [cit. 2022-03-17]. Dostupné z: https://github.com/Alg0ritmus/CryptographyInPython_Thesis.
40. BAKHVALOV, DENIS. *How to get consistent results when benchmarking on Linux?* [online] [cit. 2022-04-28]. Dostupné z: <https://easyperf.net/blog/2019/08/02/Perf-measurement-environment-on-Linux>.
41. TURBOSFV. *TurboSFV v9.20*. [online] [cit. 2022-03-22]. Dostupné z: <https://www.turbosfv.com/Blog?q=2&p0=68>.
42. FULTON, Kelsey R.; CHAN, Anna; VOTIPKA, Daniel; HICKS, Michael; MAZUREK, Michelle L. *Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study*. [online] [cit. 2022-04-30]. Dostupné z: https://obj.umiacs.umd.edu/securitypapers/Rust_as_a_Case_Study.pdf.
43. APODACA, Richard L. *Rust Ownership by Example*. [online] [cit. 2022-04-30]. Dostupné z: <https://depth-first.com/articles/2020/01/27/rust-ownership-by-example/#:~:text=For%20as%20long%20as%20a%20variable%20remains%20in,owner%20is%20no%20longer%20used%20within%20the%20scope..>

Zoznam príloh

Príloha A Práca s binárnymi dátami.

Príloha B Inštalácia jazyka Python.

Príloha C Inštalácia jazyka Rust.

Príloha D CD médium - bakalárska práca v elektronickej podobe, zdrojové kódy demonštračných aplikácií, rýchlosťné testy algoritmov, testovanie kryptografických knižníc jazyka Python, prepojenie externého Rust modulu Blake3 s jazykom Python.