FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

| | |
|---|---|
| **Název:** | Bezpečnostní analýza Linux Unified Key Setup (LUKS) |
| **Student:** | Jaroslav Kříž |
| **Vedoucí:** | Ing. Josef Kokeš |
| **Studijní program:** | Informatika |
| **Studijní obor:** | Bezpečnost a informační technologie |
| **Katedra:** | Katedra počítačových systémů |
| **Platnost zadání:** | Do konce letního semestru 2019/20 |

## Pokyny pro vypracování

Seznamte se s problematikou šifrování diskových oddílů.
Zaměřte se na nástroj Linux Unified Key Setup (LUKS). Nastudujte jeho možnosti a porovnejte je s vybranými konkurenčními programy.
Proveďte analýzu uživatelského prostředí LUKS. Vyznačte části, které mohou mít významnější bezpečnostní dopady.
Prostudujte zdrojový kód aplikace se zaměřením na nalezená riziková místa. Vyhodnoťte bezpečnost použitých algoritmů a jejich implementace.
Ověřte shodu implementace a dokumentace kryptografických primitiv pomocí jejich nezávislé reimplementace knihovnami třetích stran.
Proveďte útok hrubou silou na heslo ke svazku a změřte jeho účinnost (počet hesel s danými parametry, počet výpočtů za jednotku času).
Diskutujte své výsledky.

## Seznam odborné literatury

Dodá vedoucí práce.

prof. Ing. Pavel Tvrdík, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 11. února 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

# Bezpečnostní analýza Linux Unified Key Setup (LUKS)

***Jaroslav Kříž***

Katedra počítačových systémů
Vedoucí práce: Ing. Josef Kokeš

16. května 2019

# Poděkování

V první řadě bych chtěl poděkovat vedoucímu práce Ing. Josefu Kokešovi za odborné rady, vedení a velkou trpělivost během tvorby této práce. Dále bych chtěl poděkovat Ing. Elišce Šestákové a Ing. Tomáši Nováčkovi za poskytnutí korektury mé práci a cenné rady. V neposlední řadě bych chtěl poděkovat svým přátelům ze speciální skupiny Čtvrtkohraní za morální podporu během posledního semestru a své rodině za podporu během celého studia.

# Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 16. května 2019 ……

### Odkaz na tuto práci

Kříž, Jaroslav. *Bezpečnostní analýza Linux Unified Key Setup (LUKS)*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

# Abstrakt

Tématem této práce je analýza nástroje pro šifrování disků Linux Unified Key Setup (LUKS), jeho výhody, nevýhody a porovnání s konkurenčními nástroji napříč platformami. Práce obsahuje úvod do problematiky šifrování pevných disků a použitých kryptografických primitiv. Výsledkem práce je ověření funkčnosti zkoumaného nástroje jeho nezávislou reimplementací v jazyce C a zároveň útok hrubou silou na heslo v LUKS kontejneru, kdy je změřena jeho účinnost.

**Klíčová slova** bezpečnostní analýza, šifrování disku, LUKS, Linux, programovací jazyk C

# Abstract

The topic of this work is the security analysis of the Linux Unified Disk Encryption (LUKS), its advantages, disadvantages, and comparison with other tools across the platforms. The thesis also contains an introduction to the hard drive encryption and cryptographic primitives used in the tool. The result of this work is verification of the functionality of the investigated tool by its independent reimplementation in C language and also brute force attack on the password in LUKS container where its efficiency is measured.

**Keywords** security analysis, disc encryption, LUKS, Linux, programming language C

# Obsah

# Seznam obrázků

# Seznam tabulek

# Seznam algoritmů

# Seznam výpisů kódu

# Úvod

V dnešní době, kdy je ochrana dat skloňována téměř na denním pořádku, je potřeba svá data uchovávat v bezpečí. Představme si situaci, kdy ztratíme notebook s citlivými daty – může se jednat o dokumenty, fotografie apod. Šifrování dat sice nezajistí jejich vrácení, ale aspoň se k nim útočník nedostane, a nebude je zveřejňovat či nás nějakým způsobem vydírat. Netýká se to navíc jen dat, která máme u sebe. V čím dál větší míře data migrují na cloudové služby, kde nad nimi nemáme kontrolu už vůbec.

Práce má za cíl seznámit čtenáře s problematikou šifrování diskových oddílů s konkrétním zaměřením na nástroj Linux Unified Key Setup (LUKS), nastudováním jeho možností a porovnáním s vybranými konkurenčními programy.

Dále je cílem práce provedení analýzy uživatelského prostředí nástroje LUKS. Následně vyznačení částí, které mohou mít významnější bezpečnostní dopady, k čemuž se váže prostudování zdrojového kódu aplikace se zaměřením na nalezená riziková místa. Výsledkem této části je vyhodnocení bezpečnosti použitých algoritmů a jejich implementace.

První kapitola této bakalářské práce popisuje samotný nástroj LUKS, ať už se jedná o jeho historii, poznání jeho výhod a nevýhod, či porovnání s konkurenčními programy. Druhá kapitola čtenáře důkladně seznamuje s nástrojem LUKS, hlavičkou označující LUKS kontejner a nejdůležitějšími operacemi v nástroji. Je zde zkoumána i funkčnost nástroje za okrajových podmínek. Třetí kapitola popisuje šifry, jejich operační módy a hashovací funkce, které jsou použity pro šifrování nástrojem LUKS. Je zde vysvětlen princip antiforenzních stripů. Čtvrtá kapitola se zaměřuje na ověření shody implementace a dokumentace kryptografických primitiv pomocí jejich nezávislé reimplementace knihovnami třetích stran. Celá bakalářská práce je završena pátou kapitolou, ve které je proveden útok hrubou silou na heslo ke svazku a je změřena jeho účinnosti (počet hesel s danými parametry, počet výpočtů za jednotku času).

Kapitola 1

# LUKS – principy a historie

Tato kapitola obsahuje základní informace o nástroji LUKS, bezplatném nástroji, který umožňuje šifrovat obsah pevného disku. Jsou zde obsaženy i informace o jeho vzniku, vývoji, možnostech a porovnání s dalšími podobnými nástroji.

## 1.1 Jak šifrovat

Přirozeným řešením pro šifrování dat je použití jednoho klíče a následně tímto klíčem vždy data rozšifrovat a přistupovat k nim. Co když ale k datům chce přistupovat více osob? Sdělíme jim tento tajný klíč? A co když ho oni poví i dalším osobám? Pokud se navíc rozhodneme heslo změnit, budeme muset přešifrovat celý disk. Toto rozhodně není bezpečná a správná cesta, uvážíme-li fakt, že se uprostřed přešifrovávání – které může trvat i hodiny – objeví nějaký nečekaný problém. Velmi snadno tak můžeme přijít o všechny své drahocenné informace.

Na toto má LUKS (i mnohé další podobné nástroje) odpověď – použít tzv. *master key*, klíč, který je náhodně vygenerovaný a neznáme ho. Navíc je obtížné (takřka nemožné) ho bez přešifrování měnit. Co ale známe, jsou uživatelské klíče, které můžeme libovolně vytvářet, měnit a odstraňovat. Nemusí tak docházet k přešifrování celého disku, a získáme tím možnost používat více různých klíčů k šifrovaným datům.

Nástroj LUKS používalo v roce 2017 na distribucích Debian a Ubuntu 2,5–5 % uživatelů. Toto číslo bude pravděpodobně velmi podobné nebo vyšší i napříč ostatními distribucemi, protože u ostatních distribucí je obvyklá větší gramotnost v počítačové bezpečnosti. Při uvážení počtu uživatelů internetu čítajících zhruba 4,5 miliardy, kdy Linux používají 2 % z nich, se dostaneme na zhruba 100 miliónů uživatelů [1, 2]. Dá se tedy říci, že LUKS je dnes jedním ze standardů pro šifrování pevných disků na Linuxu.

## 1.2 Historie a vývoj

LUKS je de facto nadstavba nástroje cryptsetup, který využívá dm-crypt, nativní součást linuxového jádra pro šifrování. První ostrá verze tohoto nástroje byla vydána na počátku roku 2005, kdy byla představena svým autorem, Clemensem Fruhwirthem, společně s myšlenkou na bezpečné mazání klíčů [3]. Funkční prototyp byl představen v létě předchozího roku.

Od roku 2009 [4] se údržbě a dalšímu vývoji věnuje Čech Milan Brož, který společně s Ondřejem Kozinou vytvořil nový standard pojmenovaný LUKS2 (nebo LUKS verze 2), který je zpětně kompatibilní a opravuje mírné neduhy v návrhu původního standardu (např. chybějící záložní hlavičky, chybějící kontrolní součty) [5]. Oba dva jsou v současnosti zaměstnáni v české pobočce společnosti Red Hat.

V době psaní tohoto textu (květen 2019) je LUKS i nadále vyvíjen a jsou podporovány obě verze hlaviček. Navíc se díky volně přístupným kódům může na vývoji podílet takřka kdokoliv.

## 1.3 Možnosti

Linux Unified Key Setup byl vyvíjen, jak už název vypovídá, pro Linux. A jak je na této platformě zvykem, je nabízen zdarma. Existuje však i verze pro Windows [6]. Nástroj samotný nám umožňuje najednou používat až 8 různých hesel a dává možnost jejich snadné správy bez nutnosti přešifrování celého disku.

Díky své hlavičce a magické konstantě[1] je velmi snadno rozpoznatelné, že zařízení obsahuje zašifrovaný kontejner (tedy za předpokladu, že jsme hlavičku neoddělili a neuchovali na jiném zařízení – což LUKS také podporuje). Nástroj jako takový neimplementuje žádná kryptografická primitiva, spoléhá se na kernel a jeho cryptoAPI a správa klíčů podléhá aplikaci cryptsetup běžící v uživatelském prostoru [7].

Za zmínku stojí užití antiforenzního filtru, který zajišťuje nemožnost zrekonstruovat obsah uloženého *key slotu* již jednou smazaného klíče pomocí forenzní analýzy disku. Ke smazání může dojít například v případě automatické realokace vadného sektoru samotným diskem. Tento sektor tak již není možné konvenčními metodami přepsat. V případě, že by takovýto sektor obsahoval klíč, mohlo by dojít k obnovení informace, o které se předpokládalo, že je již úspěšně zničena. AF-filtr zajišťuje, že jsou informace rozloženy na disku takovým způsobem, že při spolehlivém odstranění – byť jen části – uchovaného klíče znemožní jeho následnou rekonstrukci [8].

Ačkoliv se LUKS používá z příkazové řádky, stále se jedná o poměrně intuitivní a uživatelsky přívětivý nástroj.

[1] sekvence bajtů na začátku LUKS kontejneru, která značí přítomnost zašifrovaných dat

## 1.4 Porovnání

Žádný nástroj není dokonalý a každý vyniká v jiné oblasti, tudíž každému může vyhovovat něco jiného. V tabulce 1.1 je proto vidět přehled porovnávaných nástrojů a některých funkcionalit, které nabízejí.

| | **VeraCrypt** | **Bitlocker** | **LUKS** |
|---|---|---|---|
| Skrytý kontejner | ANO | NE | NE |
| Více klíčů | ANO | ANO | ANO |
| TPM | NE | ANO | Částečně |
| Filesystem | Závisí na OS | NTFS | Závisí na OS |
| Licence | TrueCrypt License 3.0, Apache License 2.0 | Proprietární | Open source |
| Operační systém | Win, Linux | Win | Win, Linux |
| Dostupnost | Zdarma | Součást OS | Zdarma |
| Udržovaný | ANO | ANO | ANO |

Tabulka 1.1: Porovnání vybraných nástrojů a funkcionalit

### 1.4.1 VeraCrypt

VeraCrypt [9] je ve své podstatě velmi podobný nástroji LUKS – vytváří kontejnery, v nichž jsou data šifrována a přístupná jsou jen tehdy, když je zadáno správné heslo.

Navíc ale umožňuje vytvořit tzv. *hidden volume* (skrytý svazek), kdy v kontejneru existují dva souborové systémy a každý z nich je opatřen jiným heslem. Druhý svazek se pro okolí navíc tváří, že je naplněn náhodnými daty[2]. Tento svazek může – a zároveň nemusí – uživatel využívat. Pokud nějaká státní moc uživatele vyzve, aby vydal hesla ke svazku, uživatel může odevzdat heslo pouze k jednomu z nich. Vzhledem k tomu, že druhý (skrytý) svazek vypadá jako shluk náhodných bajtů, může uživatel argumentovat jeho neexistencí. Mnohdy neexistuje možnost prokázat opak. Toto je problematika tzv. hodnověrného popření[3].

Nástroj má velmi zajímavou historii, vznikl totiž jako fork[4], neboť vývojáři původního programu TrueCrypt [10] vydali na svých stránkách dne 28. května 2014 oznámení o ukončení vývoje společně s verzí 7.2, která už neumožňuje vytvořit nové kontejnery, ale dovoluje spravovat již existující.

Zpráva o ukončení přišla naprosto nečekaně, na stránkách projektu se objevil návod jak přejít ke konkurenčnímu nástroji Bitlocker [11]. Vše je doplněno varováním o možných bezpečnostních slabinách v TrueCryptu.

[2] ať už existuje, nebo nikoliv

[3] v angličtině známější pod názvem *plausible deniability*

[4] oddělená alternativní větev programu, kterou vyvíjí někdo jiný

TrueCrypt má své zdrojové kódy volně přístupné, i když byly z oficiálních stránek staženy. Navazující projekt VeraCrypt však z těchto kódů vychází, rozvíjí je a opravuje bezpečnostní chyby. Jelikož TrueCrypt není open source v pravém slova smyslu [12] a má svůj vlastní způsob licencování, musely původní kódy zachovat tuto licenci, nově připsané části ale používají licenci Apache License 2.0.

Dnes je VeraCrypt populárním nástrojem, odkaz svého předchůdce dodržuje stále velmi důstojně.

### 1.4.2 Bitlocker

Bitlocker je nativní řešení pro šifrování disků vyvinuté společností Microsoft a používané v jejich produktech, ať už na straně serverové (kdy je Bitlocker dostupný pro všechny verze), tak i na straně klientské. (U Windows 7 a 8 jde o edice *Enterprise* a *Ultimate*, u Windows 10 o edice *Pro* a *Enterprise*.)

Na rozdíl od nástroje LUKS umožňuje BitLocker využívat TPM[5] v plném rozsahu buď samostatně, nebo v kombinaci s PINem nebo klíčem. Pro LUKS je třeba si požadované funkcionality napsat sám a nebo se spolehnout na externí knihovnu od třetích stran [13]. V případě, že nechceme nebo nemůžeme TPM využívat, je zde stále možnost použít heslo nebo USB flash disk.

[5]Trusted Platform Module, zabezpečený kryptoprocesor, obvykle uložený na základní desce počítače, na který se mohou uložit šifrovací klíče

Kapitola 2

# LUKS – realizace

Tato kapitola seznamuje čtenáře s hlavičkou nástroje LUKS a porovnává rozdíly mezi první a druhou verzí nástroje. Dále jsou zde popsány nejdůležitější operace pro práci s tímto nástrojem. U stěžejních operací je popsán i jejich průběh.

Zdrojové kódy jsou volně dostupné v repozitáři GitLabu (viz [14]). Veškeré informace ohledně vnitřní implementace a použitých algoritmů jsou tedy veřejně dohledatelné a prověřitelné.

## 2.1 Struktura hlavičky

Diskový oddíl zašifrovaný nástrojem LUKS začíná hlavičkou a ihned za ní je tzv. *key material*. V těchto dvou částech jsou uloženy nutné informace k úspěšnému dešifrování disku. Následující oblast už obsahuje samotná zašifrovaná data. Toto rozložení dodržují obě verze nástroje.

### 2.1.1 LUKS1

Každý LUKS kontejner začíná tzv. *partition header* (phdr), za kterou je umístěn *key material* (na obrázku 2.1 označený jako KM1 až KM8 – je tedy možné uschovat až osm různých hesel). Následují samotná data zašifrovaná *master* klíčem.

| LUKS PHDR | KM1 | KM2 | ... | KM8 | samotná data |
|---|---|---|---|---|---|

Obrázek 2.1: Struktura diskového oddílu LUKS1

Informace obsažené v phdr obsahují záznam o existenci kontejneru LUKS, použité šifře a jejím operačním módu. V tabulce 2.1 je vidět kompletní rozložení jednotlivých oblastí hlavičky. Velikost phdr části hlavičky je 592 bajtů,

resp. 1 024 bajtů, pokud počítáme zarovnání oddílu. Velikost proměnné `key material` se odvíjí od několika parametrů při vytváření, například od počtu antiforenzních stripů. Hlavička jako taková obsahuje pouze zlomek velikosti celého kontejneru.

V tomto standardu neexistuje záloha hlavičky, je proto silně doporučováno hlavičku zaarchivovat a uložit na bezpečném místě, protože při změně, byť jediného bajtu, se může stát, že přijdeme o všechna data na disku.

LUKS1 používá následující datové typy, známé už z jazyka C; vícebajtové hodnoty jsou striktně v kódování big endian:

- `uint16_t` – 16bitový unsigned int
- `uint32_t` – 32bitový unsigned int
- `char[]` – kódované striktně v ASCII, ukončené binární nulou (`'\0'`)
- `byte[]` – sekvence bajtů
- `key slot` – struktura je popsána v tabulce 2.2.

| offset | jméno oblasti | délka | datový typ | popis |
|---|---|---|---|---|
| 0 | magic | 6 | `byte[]` | magická konstanta |
| 6 | version | 2 | `uint16_t` | LUKS verze |
| 8 | cipher-name | 32 | `char[]` | název šifry dle specifikace |
| 40 | cipher-mode | 32 | `char[]` | název operačního módu šifry |
| 72 | hash-spec | 32 | `char[]` | specifikace hashe |
| 104 | payload-offset | 4 | `uint32_t` | začátek zašifrovaných dat (v 512 bajtových sektorech) |
| 108 | key-bytes | 4 | `uint32_t` | délka klíče v bajtech |
| 112 | mk-digest | 20 | `byte[]` | kontrolní součet *master* klíče z PBKDF2 |
| 132 | mk-digest-salt | 32 | `byte[]` | sůl MK pro PBKDF2 |
| 164 | mk-digest-iter | 4 | `uint32_t` | počet iterací MK pro PBKDF2 |
| 168 | uuid | 40 | `char[]` | UUID oddílu disku |
| 208 | key-slot-1 | 48 | `key slot` | slot pro klíč 1 |
| … | … | … | … | … |
| 544 | key-slot-8 | 48 | `key slot` | slot pro klíč 8 |
| 592 | _padding | 432 | `char[]` | padding |

Tabulka 2.1: Rozložení phdr LUKS1

Prvních 6 bajtů diskového oddílu je vyčleněno na magickou konstantu, která určuje, že jde o LUKS kontejner. Jedná se o sekvenci bajtů

`'L', 'U', 'K', 'S', '0xBA', '0xBE'.`

Následuje číslo verze – v LUKS1 se jedná striktně o číslo 1. V polích označených jako `cipher-name`, `cipher-mode` a `hash-spec` je uložen název použité šifry, jejího módu a specifikace hashe. Proměnná `payload-offset` obsahuje informaci o začátku zašifrovaných dat. Tato proměnná se během existence LUKS kontejneru nemění, je ale stěžejní pro správné fungování. V případě nesprávné hodnoty může vést až k přepsání zašifrovaných dat. V určitých případech lze hlavičku úplně oddělit a skladovat na jiném zařízení, v tomto případě je pak offset nulový.

V proměnné `key-bytes` se skrývá délka *master* klíče, toto číslo je uložené v bajtech. Dále následují `mk-digest`, `mk-digest-salt` a `mk-digest-iter`, které uchovávají kontrolní součet *master* klíče po výstupu z funkce PBKDF2, sůl, která je použita jako vstup do té samé PBKDF2 funkce a počet iterací, kterým prochází *master* klíč. Více o fungování tohoto algoritmu je zmíněno v kapitole 3.1.3.

Dále phdr obsahuje 128bitovou hodnotu `uuid`[6], která identifikuje disk. Poté následují jednotlivé *key sloty*.

| offset | jméno oblasti | délka | datový typ | popis |
|---|---|---|---|---|
| 0 | active | 4 | `uint32_t` | stav *key slotu* aktivní/neaktivní |
| 4 | iterations | 4 | `uint32_t` | počet iterací pro PBKDF2 |
| 8 | salt | 32 | `byte[]` | sůl pro PBKDF2 |
| 40 | key-material-offset | 4 | `uint32_t` | začátek sektoru pro *key material* |
| 44 | stripes | 4 | `uint32_t` | počet antiforenzních stripů |

Tabulka 2.2: Rozložení key slotu LUKS1

Stav *key slotu* je definován konstantou `0x0000DEAD` pro neaktivní slot, popřípadě `0x00AC71F3` pro slot aktivní. Tato konstanta je proto uložena v proměnné `active`.

Počet iterací pro funkci PBKDF2 a sůl pro uživatelské heslo uchovávají `iterations` a `salt`. Proměnná `key-material-offset` určuje začátek sektoru, kde je uložený odpovídající *key material*. Počet antiforenzních stripů obsahuje proměnná `stripes`. Toto číslo je definované jako konstanta s hodnotou 4 000, v současné době ho nelze měnit. Jako poslední je v hlavičce `_padding`, který zarovnává strukturu do 512bajtových sektorů. Je vyplněn vždy nulami.

[6]Universally Unique Identifier

### 2.1.2 LUKS2

LUKS2 byl navržen tak, aby vyřešil nedostatky z předchozí verze – jedná se zejména o kontrolní součty (pro možnost detekovat poškození dat) a existenci záložní binární hlavičky pro opravu hlavičky poškozené.

Hlavičku u této verze můžeme rozdělit do tří samostatných částí:

- binární hlavička,
- oblast pro metadata (uložená v JSON[7] formátu)
- oblast s klíči.

Jak je vidět na obrázku 2.2, ihned za binární hlavičkou a metadaty následuje záložní hlavička. Oblast s klíči se vyskytuje pouze jednou. Velikost binární hlavičky je přesně 512 bajtů a je zaručeno, že se zapíše atomicky do jednoho sektoru disku. Data uložená v JSON části hlavičky jsou skladována jako řetězec jazyka C (ukončen binární nulou). Jeho velikost je nastavena přesně na 12 kB.


Obrázek 2.2: Struktura diskového oddílu LUKS2

V hlavičce druhé verze nástroje LUKS jsou použity následující datové typy, vícebajtové hodnoty jsou opět uloženy v kódování big endian:

- `uint8_t` – 8bitový unsigned int
- `uint16_t` – 16bitový unsigned int
- `uint64_t` – 64bitový unsigned int
- `char[]` – vyžadovány pouze ASCII hodnoty ukončené binární nulou
- `uint8_t[]` – pole bajtů

Z důvodu kompatibility napříč verzemi je nutné, aby bylo možné identifikovat LUKS kontejner. Proto jsou proměnné, jmenovitě magická konstanta a verze, totožné v obou verzích. V případě záložní hlavičky je konstanta `magic` zapsána obráceně (`'S'`, `'K'`, `'U'`, `'L'`, `'0xBA'`, `'0xBE'`). Obsah proměnné `version` je nastaven na 2.

[7] JavaScript Object Notation

| offset | jméno oblasti | délka | datový typ | popis |
|---|---|---|---|---|
| 0 | magic | 6 | `char[]` | magická konstanta |
| 6 | version | 2 | `uint16_t` | verze LUKSu |
| 8 | hdr_size | 8 | `uint64_t` | v bajtech, zahrnuje JSON oblast |
| 16 | seqid | 8 | `uint64_t` | zvyšuje se s každým updatem |
| 24 | label | 48 | `char[]` | ASCII popis |
| 72 | csum_alg | 32 | `char[]` | alg. kontrolního součtu |
| 104 | salt | 64 | `uint8_t[]` | sůl |
| 168 | uuid | 40 | `char[]` | uuid zařízení |
| 208 | subsystem | 48 | `char[]` | volitelný sekundární popis |
| 256 | hdr_offset | 8 | `uint64_t` | offset od začátku v bajtech |
| 264 | _padding | 184 | `char[]` | vyplněno nulami |
| 448 | csum | 64 | `uint8_t[]` | kontrolní součet hlavičky |
| 512 | _padding4096 | $7 \cdot 512$ | `char[]` | padding, vyplněno nulami |

Tabulka 2.3: Rozložení binární phdr LUKS2

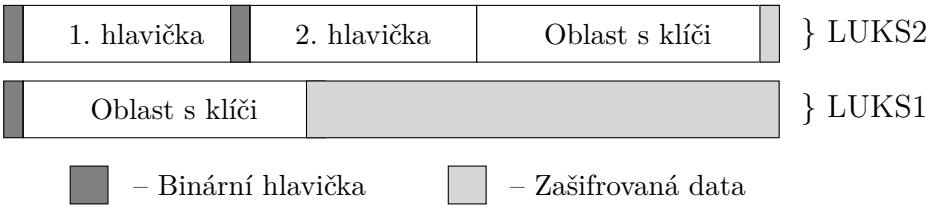
Proměnná `hdr_size` obsahuje velikost hlavičky včetně JSON oblasti. Přesnou velikost JSON oblasti tedy zjistíme odečtením velikosti binární hlavičky (4 096). Záložní hlavička musí obsahovat totožný údaj, proměnná slouží jako ochrana proti přepsání hlavičky jinou JSON oblastí.

Následuje `seqid`, což je vlastně čítač, který se zvyšuje s každou aktualizací hlavičky. Hlavička, která obsahuje vyšší číslo, je použita pro účel obnovení. Pokud se primární a sekundární hlavička rozchází v číslování, je automaticky použita ta s vyšším číslem.

Stejně jako u klasických disků je i zde umožněno uložit popis disku (či kontejneru), k tomu slouží proměnná `label`. V `csum_alg` je uložen řetězec obsahující název algoritmu pro kontrolní součet. Ten je vypočítán z binární hlavičky a i odpovídajícího JSONu. Výchozí algoritmus je SHA256. Zbývající bajty jsou doplněny nulami.

Sůl je vygenerována pro každou hlavičku zvlášť, tudíž každá z hlaviček obsahuje v proměnné `salt` unikátní hodnoty. V `uuid` je uloženo UUID zařízení. Jaké odsazení má hlavička od počátku zařízení je zmíněno v `hdr_offset`. Jedná se o číslo v bajtech. Následuje `_padding`, který je vyplněný binárními nulami.

Kontrolní součet hlavičky je vypočítán podle dříve definovaného algoritmu v `csum_alg` a uložen do `csum`. V případě kratšího kontrolního součtu je vše doplněno nulami. To vše uzavírá zarovnávací proměnná `_padding4096`, která zajišťuje, že binární hlavička bude mít přesně 4 096 bajtů. Je vyžadováno, aby padding obsahoval pouze binární nuly.




Obrázek 2.3: Porovnání velikostí struktur LUKS1 a LUKS2

V JSON oblasti jsou uložena data v JSON formátu. Jak již bylo zmíněno dříve, v současné době je velikost přesně 12 kB. Nepoužitá část této oblasti je ponechána vynulovaná.

## 2.2 Kompatibilita napříč verzemi

Zpětná kompatibilita je zaručena, budoucí verze nemusí být schopny vytvořit starší hlavičky, ale je garantováno, že je budou umět číst. Jak podotýká dokumentace [15], budoucí verze by při interpretaci neměla vrátit chybu. Výjimkou je samozřejmě nepřítomnost magické konstanty.

Navíc je možné pomocí příkazu `cryptsetup convert --type luks2` převést starší hlavičku na novou, tedy za předpokladu, že se na disku nachází dostatek volného místa. Pokud LUKS2 používá kompatibilní nastavení s LUKS1 (PBKDF2, bez ochrany integrity, bez tokenů), lze ho naopak převést zpět. Na to využijeme stejný příkaz, tentokrát však s jiným parametrem přepínače `cryptsetup convert --type luks1`. Na obrázku 2.3 je vidět rozdílná velikost struktur.

## 2.3 Inicializace

Průběh celého procesu inicializace je popsán v algoritmu 1. Pro vytvoření nového kontejneru a jeho inicializaci zadáme příkaz `cryptsetup -v luksFormat <zařízení>`. Ihned dostaneme varování o přepsání dat a musíme ho potvrdit. Varování je zachycené na obrázku 2.4.

```
WARNING
=======
This will overwrite data on <device> irrevocably.
Are you sure? (type uppercase yes):
Enter passphrase:
Verify passphrase:
```

Obrázek 2.4: Varování o přepsání oblasti

masterKeyLength ← definováno uživatelem
masterKey ← vygeneruj náhodný vektor,
délka: masterKeyLength

phdr.magic ← LUKS_MAGIC
phdr.version ← 1
phdr.cipher-name ← zadáno uživatelem
phdr.cipher-mode ← zadáno uživatelem
phdr.hash-spec ← zadáno uživatelem
phdr.key-bytes ← masterKeyLength
phdr.mk-digest-salt ← vygeneruj náhodný vektor,
délka: LUKS_SALTSIZE

phdr.mk-digest-iter ← vypočítáno dle uživatelského vstupu
phdr.mk-digest ← PBKDF2(masterKey,
phdr.mk-digest-salt,
phdr.mk-digest-iter,
LUKS_DIGESTSIZE)

$$\text{baseOffset} \leftarrow \left\lfloor \frac{\text{velikost phdr}}{\text{LUKS\_SECTOR\_SIZE}} + 1 \right\rfloor$$

stripes ← LUKS_STRIPES nebo zadáno uživatelem

$$\text{keyMaterialSectors} \leftarrow \frac{\text{stripes} * \text{masterKeyLength}}{\text{LUKS\_SECTOR\_SIZE}} + 1$$

**for** *ks* ← 0 **to** *LUKS_NUMKEYS* **do**
ks.active ← LUKS_KEY_DISABLED

//zarovnat na násobky LUKS_ALLIGN_KEYSLOTS
baseOffset ← zaokrouhli_nahoru(baseOffset,
LUKS_ALLIGN_KEYSLOTS)
ks-key-material-offset ← baseOffset
baseOffset ← baseOffset + keyMaterialSectors
ks.stripes ← stripes
**end**

phdr.payload-offset ← baseOffset
phdr.uuid ← vygeneruj uuid

zápis phdr na disk

Algoritmus 1: Inicializace kontejneru LUKS1 [15]

O vytvoření verze 1, nebo 2 rozhoduje verze balíku cryptsetup. Do verze programu 2.1.0 je jako výchozí LUKS zvolen LUKS1, od novějších verzí LUKS2 [16]. Nicméně si lze přepínačem `--type <luks1/luks2>` explicitně zvolit verzi.

Dochází k přemazání veškerých dat v celé délce hlavičky a nastavení hodnot na binární nuly. Zatímco velikost phdr části hlavičky je za všech okolností konstantní, *key material* část je závislá na délce *master* klíče a počtu antiforenzních stripů.

Pro rozpoznání validní LUKSové hlavičky je potřeba, aby se inicializovala magická konstanta a nastavila správná verze. Pro naše účely budeme předpokládat, že inicializujeme LUKS1.

Následně dochází k volbě šifry a jejího operačního módu uživatelem. Výchozím nastavením je šifra AES s operačním módem XTS a inicializačním vektorem plain64. Proměnná `cipher-name` je tedy vyplněna řetězcem `"aes"` a proměnná `cipher-mode` řetězcem `"xts-plain64"`.

Následně je z pseudonáhodného zdroje vygenerován řetezec o parametrizovatelné délce konstanty `masterKeyLength`, resp. `key-bytes`, kdy je v základu zvolena velikost konstanty 32 bajtů. Náhodná data jsou vygenerována z výchozího zdroje náhodnosti `/dev/random` a nebo volitelného zdroje `/dev/urandom`. To samé je poté uplatněno pro proměnnou `mk-digest-salt`, kdy je délka vygenerovaného řetězce ovlivněna konstantou `LUKS_SALTSIZE`.

Dříve vygenerovaný *master* klíč je poté společně se solí, počtem iterací a proměnnou určující délku kontrolního součtu vypočítané hodnoty z této funkce vložen do PBKDF2 funkce, jako by to bylo normální heslo od uživatele. Proměnná `mk-digest-iter` je plně závislá na uživatelském vstupu. Dle dokumentace bylo dříve `mk-digest-iter` nastaveno na 10 iterací. V současnosti je minimální hodnota nastavena na 1 000. Průměr hodnot používaných na notebooku s procesorem `Intel i5-7200U` se pohybuje okolo 180 tisíc iterací klíče. Vypočítaná hodnota je následně několikanásobně zkontrolována přepočítáním a poté uložena do `mk-digest`.

Při vytváření nového kontejneru je bezpředmětné, aby se vytvořil pouze *master* klíč, je třeba zadat alespoň jeden klíč pro přístup. Ostatní *key sloty* jsou proměnnou `active` nastaveny na „disabled“, čili konstantu `0x0000DEAD`. Současně s tím probíhá vypočítání jednotlivých odsazení. Celkové odsazení je nakonec zapsáno do `payload-offset`.

V neposlední řadě dochází k zapsání počtu stripů. Jak již bylo zmíněno výše, v současnosti nelze použít jiný počet stripů než 4 000. Tato konstanta je napevno vložena do kódu. Při změně konstanty na cokoliv jiného je vyhodnoceno makro assert, vyvolána výjimka, a my dostaneme odpověď `LUKS keyslot <number> is invalid`.

Jako poslední dochází k vygenerování identifikátoru svazku (UUID), jeho následnému převodu z binární podoby do ASCII a konečně k zapsání do `uuid`.

Pokud vše proběhlo korektně, je nová hlavička atomicky zapsána do sektoru na disku. Inicializace je zakončena návratovou hodnotou funkce. Můžeme

však použít přepínač `-v`, kdy je status úspěchu vrácen i textově, oznamující: `Command (un)successful`.

## 2.4 Získání master klíče

Pro přístup k datům je musíme nejdříve rozšifrovat *master* klíčem. Toto popisuje algoritmus 2.

Po načtení phdr z disku je ověřena existence magické konstanty na začátku kontejneru a zkontrolována verze LUKS. Následně zadá uživatel heslo, které je podrobeno průchodu smyčkou přes všechny aktivní sloty v phdr. V této smyčce prochází heslo funkcí PBKDF2 a výsledek je, jakožto kandidát na *master* klíč, porovnán se skutečným kontrolním součtem uloženým na disku.

Pro správné ověření kandidáta je třeba jej po dešifrování nechat spojit antiforenzními stripy. To za nás dokáže funkce AFmerge.

Pokud se podařilo ověření kandidáta s uloženým klíčem, je správný klíč navrácen z funkce. V opačném případě je vrácena chyba, žádné heslo totiž neodpovídá dosud uloženým heslům.

```
načti phdr z disku
zkontroluj LUKS_MAGIC a verzi
masterKeyLength ← phdr.key-bytes
pwd ← načti heslo od uživatele
foreach aktivní ks v phdr do
    pwd-sec ← PBKDF2(pwd, ks.salt, ks.iteration-count,
                     masterKeyLength)
    načti_z_disku(encryptedKey,                    // cíl
                  ks.key-material-offset,          // číslo sektoru
                  masterKeyLength · ks.stripes) // počet bajtů
    splitKey ← decrypt(phdr.cipherSpec, pwd-sec,
                       encryptedKey, encryptedLength)
    MKkandidát ← AFmerge(splitKey, masterkeyLength, ks.stripes)
    MKkandidát-res ← PBKDF2(MKkandidát, phdr.mk-digest-salt,
                            phdr.mk-digest-iter,
                            LUKS_DIGEST_SIZE)
    if MKkandidát-res == phdr.mk-digest then
        return MKkandidát jako správný master klíč
    end
end
vrať chybu, heslo neodpovídá uložených heslům
```

Algoritmus 2: Získání master klíče [15]

```
masterKey // je v paměti díky předchozímu kroku
          // – inicializaci, nebo rekonstruováno ze správného hesla

masterKeyLength ← phdr.key−bytes
emptyKeySlotIndex ← najdi neaktivní slot podle konstanty
                    LUKS_KEY_DISABLED v keyslot.active
keyslot ks ← phdr.keyslots[emptyKeySlotIndex]
PBKDF2-IterationsPerSecond ← benchmark systému
ks.iteration−count ← PBKDF2-IterationsPerSecond ·
                     intendedPasswordCheckingTime (v sekundách)

ks.salt ← vygeneruj náhodný vektor, délka: LUKS_SALTSIZE
splitKey ← AFsplit(masterKey, masterKeyLength, ks.stripes)
splitKeyLength ← masterKeyLength · ks.stripes
pwd ← načti heslo od uživatele
pwd-sec ← PBKDF2(pwd, ks.salt, ks.iteration-count, masterKeyLen)
encryptedKey ← encrypt(phdr.cipher−name, phdr.cipher-mode,
                 pwd-sec, splitKey, splitKeyLength)

zapis_na_disk(encryptedKey,
              ks.key-material-offset, // číslo sektoru
              splitKeyLength)         // délka v bajtech

ks.active ← LUKS_KEY_ACTIVE
aktualizuj key slot na phdr
```

Algoritmus 3: Vytvoření klíče [15]

## 2.5 Přidání hesla

Přidávání nového hesla je popsáno v algoritmu 3 a začíná získáním nezašifrované kopie *master* klíče. Kopii můžeme získat dvěma způsoby, klíč už je v paměti (právě totiž probíhá proces inicializace nového kontejneru), nebo přidáváme nový klíč nezávisle a tudíž musíme znát alespoň jedno z dříve zadaných hesel.

Za předpokladu, že se nám podařilo získat kopii *master* klíče, který tímto máme uložený v paměti, je zvolen první volný *key slot*, do kterého přidáme nové heslo. Nebo si můžeme slot explicitně zvolit.

V případě, že uživatel chce přidat další heslo, i když máme zaplněné všechny sloty, je i tak nejdříve požádán o zadání existujícího hesla. Varování o zaplnění všech volných slotů je zachyceno na obrázku 2.5. To samé se děje v případě zvolení zaplněného slotu, v tomto případě chybová hláška vypadá následovně: `Key slot <slot> is full, please select another one`.

Dalším krokem pro přidání nového hesla je získání soli z (pseudo) náhodného zdroje (máme na výběr mezi výchozí hodnotou `/dev/random` a nebo

```
Enter any existing passphrase:
Enter new passphrase for key slot:
Verify passphrase:
All key slots full.
```

Obrázek 2.5: Varování o zaplnění všech volných slotů

`/dev/urandom`), a poté musíme zvolit počet iterací klíče. Počet iterací musí být striktně vyšší než 1 000. V případě zadání menšího čísla je nastaveno právě na 1 000. Veškeré informace jsou zapsány do volného slotu na klíč[8].

Minimální délka hesla není nijak omezena – heslo může být dlouhé 0 znaků, maximální počet je teoreticky omezen až na 512 znaků. Cokoliv nad bude oříznuto právě na 512 prvních znaků. Při volbě hesla dlouhého 0 znaků lze bez problému přidávat další klíče, nicméně pokud tento kontejner využíváme pro zašifrování systému, je pro heslo vyžadován neprázdný řetězec.

Uživatel zadá heslo (pro zajistění maximální kompatibility napříč systémy a platformami je striktně doporučeno vytvořit ho výhradně tisknutelnými znaky z ASCII), které je ihned zpracováno funkcí PBKDF2. *Master* klíč je rozdělen pomocí AFsplitteru do určeného počtu stripů. Toto číslo je stanoveno podle zadané konstanty v *key slotu*. Výsledek je zapsán do odpovídající *key material* části hlavičky, a to v zašifrované podobě. Šifrování používá ten samý postup jako šifrovaná data, ale zde je namísto *master* klíče použit výstup z PBKDF2.

## 2.6 Odebrání hesla

Příkazem `cryptsetup luksRemoveKey <zařízení>` je umožněno smazání určitého hesla z LUKS kontejneru. Odebráním se myslí smazání daného oddílu s *key materialem* a jeho několikanásobným přepsáním. Současně s tím probíhá to samé v *key slotu*. Konečnou fází je přepsání obou oblastí binárními nulami.

Pokud se snažíme smazat neexistující heslo, jsme na to upozorněni chybovou hláškou zobrazenou na obrázku 2.6.

```
Enter passphrase to be deleted:
No key available with this passphrase.
```

Obrázek 2.6: Varování o mazání neplatného hesla

Oblast pro výmaz konkrétního hesla začíná na místě definovaném proměnnou `ks.key-material-offset` a pokračuje až po počet bajtů získaný z `phdr.key-bytes` · `ks.stripes`.

[8]Tedy takového, který je označen jako neaktivní.

```
WARNING
=======
This is the last keyslot. Device will become unusable
after purging the this key.
Are you sure? (type uppercase yes):
```

Obrázek 2.7: Varování o smazání posledního klíče

Pro výmaz konkrétního slotu s klíčem můžeme využít vestavěný příkaz `luksKillSlot`. Pro úspěšně provedenou operaci je nezbytné znát heslo od jiného slotu, nelze použít heslo ke slotu, který právě mažeme.

Výjimkou je smazání posledního klíče, ale provedení této operace je podmíněno zadáním souhlasu. Toto je ilustrováno na obrázku 2.7. Díky této akci se stává kontejner nadále nepoužitelným[9].

## 2.7 Změna hesla

Změna hesla se skládá z operací Získání master klíče, Přidání hesla a Odebrání hesla, které jsou popsané v sekcích výše.

## 2.8 Otevření zašifrovaného disku

Poté, co vytvoříme zašifrovaný svazek, ho chceme používat jako jakýkoliv nezašifrovaný. Prvním krokem je ale vytvoření souborového systému, jeho následné připojení a až poté na něj můžeme nahrávat nejrůznější data.

Pro úspěšné otevření zařízení jsou vyžadována vyšší privilegia, než která má běžný uživatel, nesmíme zapomenout na příkaz `sudo`. Následně zařízení otevřeme pomocí `cryptsetup open <svazek> [--type luks] <jméno>`, kdy je prvním argumentem připojované zařízení, druhým, volitelným, je typ kontejneru a třetím je název, pod kterým bude dešifrované zařízení, umístěné v `/dev/mapper/<jméno>`, namapováno do systému.

Popřípadě můžeme využít starší syntaxi `cryptsetup luksOpen <svazek> <jméno>`, kdy opět prvním argumentem je připojované zařízení a druhým je zamýšlený název zařízení. Ihned nás nástroj požádá o zadání hesla k zašifrovanému svazku.

Pokud jsme zadali správné heslo, máme k dispozici dešifrované zařízení `/dev/mapper/<jméno>`, na kterém pomocí nástroje `mkfs` vytvoříme souborový systém a zařízení připojíme. Pro naše účely využijeme souborový systém ext4[10] příkazem `mkfs.ext4 /dev/mapper/<jméno>`. Následně vytvoříme

[9] Neboť pro zadání nového klíče je třeba zadat existující klíč. My jsme ale poslední smazali!

[10] Extended File System je třída souborových systémů nativní pro operační systém Linux.

tzv. *mountpoint*, přípojný bod, odkud se bude do našeho zašifrovaného zařízení přistupovat příkazem `mount /dev/mapper/<jméno> <mountpoint>`. To vše zakončíme přidáním práv uživateli pomocí `chown -R <jméno uživatele> <mountpoint>`, aby měl nad diskem úplnou kontrolu.

## 2.9 Uzavření zašifrovaného disku

Pro úspěšné uzavření disku probíhají opačné operace než v případě otevření disku. Tyto operace začínají odpojením již připojeného zařízení přes `umount <zařízení>`, kdy `<zařízení>` můžeme identifikovat jak klasickým označením, tak přípojným bodem. Jedná se opět o operaci vyžadující vyšší privilegia, než která má běžný uživatel. Opět tedy využijeme příkaz `sudo`.

Proces uzavření disku dokončíme odebráním namapovaného disku pomocí `cryptsetup luksClose <jméno>` a nebo opět využijeme starší syntaxi `cryptsetup luksClose <jméno>`. Současně s tím je z paměti kernelu odebrán i klíč ke svazku.

Kapitola 3

# Kryptografický backend

V následující kapitole jsou popsány veškeré šifry, jejich operační módy a hashovací funkce, které využívá nástroj LUKS. Zároveň je zde vysvětlen princip antiforenzních stripů, unikátní ochrany klíče jeho rozprostřením po *key materialu*, který je specifický pro tento nástroj.

## 3.1 AF-stripy

„*Pevné disky mají opravdu velmi dlouhou paměť. Když si myslíte, že data jsou pryč, i pokud přepíšete celý disk nulami, dokonce i když zajistíte bezpečné smazání pomocí nástrojů pevného disku, mohou být snadno obnovena, pokud není věnována zvláštní péče jejich řádnému zničení.*“ [3]

LUKS je unikátní zejména díky svým antiforenzním stripům (dále též jako „AF-stripy“), které umožňují rozprostření zašifrovaných informací o klíči napříč celou oblastí *key materialu*. Jelikož *key material* je poměrně krátký a vejde se do jednoho sektoru disku, je teoreticky možné ho podrobit forenzní analýze. Myšlenka AF-stripů je založena na konceptu, kdy je *key material* rovnoměrně rozložen napříč více sektory a úspěšné zničení jednoho datového sektoru vede ke znemožnění obnovení celé oblasti. Toto řešení bylo v dokumentu [17] představeno autorem nástroje LUKS – Clemensem Fruhwirthem. Ten jej následně implementoval do svého nově vzniklého nástroje.

Písmenem $D$ označíme zašifrovaný master klíč nebo data, která chceme rozprostřít na disku. Difuzní funkci označíme jako $H$ a počet stripů jako $n$. Výstup dat z funkce AF-split označíme jako $S$, kdy $S = s_1, s_2, \ldots, s_n$. Prvky $s_1, \ldots, s_{n-1}$ jsou náhodně zvolená čísla, kdežto $s_n$ je získáno dle předpisu 3.1

$$s_n = s_1 \oplus s_2 \oplus \cdots \oplus s_{n-1} \oplus D. \tag{3.1}$$

Rekonstrukce klíče $D$ dosáhneme provedením inverzní operace, v našem případě se jedná o XOR všech prvků $s_i$.

$$D = \bigoplus_{i=1}^{n} s_i \tag{3.2}$$

Pro správný výsledek je nezbytně nutné, abychom měli všechny prvky $s_i$ v jejich původní podobě, změna byť jediného bajtu vede k úplně jinému výsledku. Což, pokud máme v plánu zničit uložený klíč, je účel.

Za současného řešení není schéma citlivé na změny v jednotlivých bitech. Pokud nastane změna k-tého bitu v $s_i$, je poté ovlivněn pouze k-tý bit výsledných dat $D$. Proto schéma rozšíříme o přidání hashovací funkce do řetězce XORů. Tento přístup nám zajistí právě onu citlivost a změna jednoho bitu $s_i$ vede ke změně celého $D$.

$$d_0 = 0 \tag{3.3}$$

$$d_k = H(d_{k-1} \oplus s_k) \tag{3.4}$$

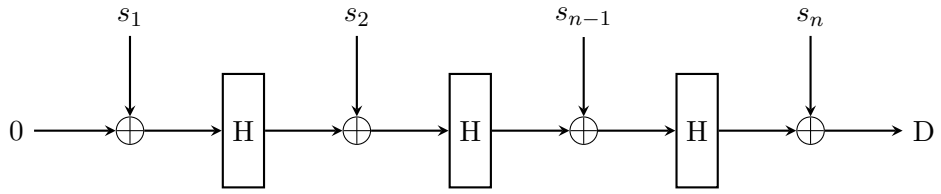
Jelikož hashovací funkce má být pouze jednosměrná a nemůžeme si vybrat výstup, je poslední prvek $s_n$ spočítán bez průchodu touto funkcí, a tedy podobně jak již bylo zmíněno výše.

$$s_n = d_{n-1} \oplus D \tag{3.5}$$

Zatímco funkce AF-split se stará o rozprostření *master* klíče napříč *key slotem*, funkce AF-merge se stará o jeho rekonstrukci. *Master* klíč $D$ získáme úpravou rovnice 3.5.

$$D = d_{n-1} \oplus s_n \tag{3.6}$$

Princip fungování AF-split je znázorněn na obrázku 3.1.




Obrázek 3.1: Schéma funkce AF-split

### 3.1.1 $H_1$

$H_1$ je hashovací funkce, znázorněná na obrázku 3.1 jako H, která může operovat nad proměnlivým počtem dat a jako její základ je v implementaci použita

hashovací funkce SHA1. Výstupní počet bajtů z této funkce je označen jako $P$ a délka výstupu je vyjádřena jako $|P|$. Funkce SHA1 má $|P|$ rovno 160 bitům.

Vstupem do funkce $H_1$ je parametr $d$, který je rozdělen do jednotlivých bloků $d_i$ o délce $|P|$. Toto se opakuje až do posledního bloku $d_n$, který může být kratší a jeho délka je označena jako $|d_n|$. Číslo $i$ vstupuje do funkce jako `uint32_t` v kódování big endian. Transformace je následující

$$p_i = P(i \parallel d_i) \tag{3.7}$$

„*$H_1$'s function definition stems from an implementation error that I'm responsible for. Do not try to analyse it, the structure given here is specified according to this implementation error and hence is a mistake itself*".

Toto je přepis upozornění z dokumentace [15], kde bylo zjištěno, že hashovací funkce $H_1$ trpí nezamýšlenou chybou v implementaci. Všechny implementace by se měly ale striktně držet této chyby, zamýšlené fungování je následně opravené funkcí $H_2$ popsanou níže.

### 3.1.2 $H_2$

$H_2$ funguje totožně jako $H_1$, s výjimkou

$$p_i = P(i \parallel d) \tag{3.8}$$

kdy oproti předcházející rovnici 3.7 chybí u $d$ dolní index. Podle poznámky v dokumentaci [15] je tato chyba držena napříč celým vývojovým stromem LUKS1. Opravená implementace je ale až v druhé verzi nástroje LUKS.

### 3.1.3 PBKDF2

PBKDF2[11] je tzv. *key derivation function*, která pomocí hesla, soli, zvoleného hashovacího algoritmu a počtu iterací vytvoří klíč o délce definované uživatelem. Při správném použití této funkce (a vyššímu počtu iterací) nám vznikne klíč, který je obtížněji napadnutelný jak metodou *brute-force* (kdy se zkouší vygenerovat všechny možné kombinace, až nalezneme správné heslo), tak metodou slovníkového útoku a předpočítaných tabulek. Je to z důvodu delšího času potřebného pro průběh.

V případě nástroje LUKS je počet iterací vypočítán interním benchmarkem, kdy je zkoumáno, kolik iterací lze provést za jednotku času. V základu je zvolena 1 sekunda. Příklad několika procesorů a k němu odpovídajícímu počtu iterací *master* klíče a počtu iterací *key slotů* je viditelný v tabulce 3.1.

S pomocí přepínače `-i` nebo `--iter-time` lze konstantu zadanou v milisekundách změnit. Novější a výkonnější procesory zvládají za jednotku času více operací a počet iterací u nich tedy může jít až k jednotkám milionů.

[11] Password Based Key Derivation Function 2

| Procesor | Počet iterací MK | Počet iterací slotu |
|---|---|---|
| Intel i7-7500U | 122 750 | 476 721 |
| Intel i5-7200U | 186 712 | 2 987 396 |
| Intel i5-4200U | 165 704 | 2 681 780 |
| Intel i7-7700HQ | 210 726 | 3 371 626 |

Tabulka 3.1: Počet iterací klíčů za jednotku času

Lze využít přepínač `--pbkdf-force-iterations`, kdy je pevně stanoven počet iterací bez ohledu na čas. Přesto však nelze počet iterací klíče snížit pod hranici 1 000 iterací. Použitím tohoto přepínače se ale odstraňuje podstatná bezpečnostní výhoda, která je později napadána v kapitole 5.

LUKS používá hashovací funkci SHA1 jako PRF[12], tato funkce je uvedena jako výchozí v proměnné `hash-spec`, ale lze sem zvolit prakticky jakoukoliv. $INT(i)$ je číslo bloku $i$ převedené na `uint32_t`.

Průběh algoritmu popisuje algoritmus 4. Referenční implementace je dostupná v dokumentaci [18].

**Funkce** *F(PRF, Password, Salt, count, i)*
$U_1 \leftarrow PRF(Password, Salt \parallel INT(i))$
**for** $a \leftarrow 2$ **to** *count* **do**
$U_a \leftarrow PRF(Password, U_{a-1})$
**end**
$Res \leftarrow U_1 \oplus U_2 \oplus \cdots \oplus U_{count}$
**return** *Res*

**Funkce** *PBKDF2(PRF, Password, Salt, count, keyLen)*
blockCnt $\leftarrow \left\lceil \frac{keyLen}{blockLen} \right\rceil$
extraBytes $\leftarrow$ keyLen $-$ (blockCnt - 1) $\cdot$ blockLen
**for** $i \leftarrow 1$ **to** *blockCnt* **do**
$T_i \leftarrow F(PRF, Password, Salt, count, i)$
**end**
KEY $\leftarrow T_1 \parallel T_2 \parallel \cdots \parallel T_{blockCnt<0...extraBytes-1>}$
**return** KEY

Algoritmus 4: Průběh PBKDF2

[12]Pseudo Random Function

## 3.2 Šifrovací algoritmy

V současnosti je podporováno pouze pět šifer, kterými je umožněno šifrovat data tímto nástrojem. Každá ze šifer byla navrhnuta známými kryptology a poté podrobena prozkoumání. V žádné z nich nebyla nalezena bezpečnostní zranitelnost. Šifra CAST6 byla kandidát a šifry Rijndael, Twofish a Serpent byly finalisty v celosvětové soutěži o nový standard při šifrování. Organizací NIST[13] byla nakonec v říjnu roku 2000 jakožto AES (Advanced Encryption Standard) zvolena šifra Rijndael [19].

Výběr konkrétní šifry lze zvolit přepínačem `-c` nebo `--cipher` a specifikováním názvu malými písmeny. Jedná se o šifry:

- AES,
- Twofish,
- Serpent,
- CAST5,
- CAST6.

Na rozdíl od nástroje VeraCrypt námi zkoumaný nástroj LUKS neumožňuje použití tzv. kaskádových šifer, kombinaci více šifer použitých za sebou. Toto přináší VeraCryptu určitou výhodu – každá z jednotlivých šifer má vlastní *master* klíč a tudíž je kontejner tak silný, jak jeho nejsilnější šifra. Pokud je jedna z šifer prolomena, je tu stále další šifra v pořadí, která musí být prolomena.

Toto řetězení ovšem zpomaluje proces šifrování a dešifrování. Ačkoliv můžeme použít dedikované instrukce procesoru, např. AES-NI[14] a tento proces zrychlit, stále je při použitém řetězení znatelný propad v rychlosti. (Za povšimnutí stojí rozdíl v rychlosti procesu šifrování/dešifrování mezi použitím samotné šifry a jejím použitím v libovolné kaskádě v tabulce 3.2.)

### 3.2.1 AES

AES je symetrická bloková šifra s délkou bloku 128 bitů a délkou klíče 128, 192, popřípadě 256 bitů. AES je speciálním případem šifry Rijndael (odvozeno od jmen autorů V. Rijmena a J. Daemena). Názvem AES se implicitně myslí délka klíče 128 bitů.

Blok délky 128 bitů je rozdělen do matice $4 \times 4$ bajtů. V závislosti na délce klíče je poté následně proveden určitý počet rund – 10 pro délku 128 bitů, 12 pro délku 192 bitů a pro délku klíče 256 bitů je použito rund 14.

[13] Americký úřad pro standardizaci, National Institute of Standards and Technology
[14] Advanced Encryption Standard New Instructions

| Šifra | Šifrování | Dešifrování |
|---|---|---|
| AES | 1 483,4 MiB/s | 1 556,5 MiB/s |
| Serpent | 559 MiB/s | 561,2 MiB/s |
| Twofish | 310,5 MiB/s | 309,6 MiB/s |
| AES(Twofish) | 392 MiB/s | 406 MiB/s |
| Twofish(Serpent) | 225 MiB/s | 230 MiB/s |
| AES(Twofish(Serpent)) | 211 MiB/s | 211 MiB/s |

Tabulka 3.2: Benchmark šifer

Pro zašifrování jednoho bloku jsou použity tzv. rundovní klíče, které jsou odvozené z šifrovacího klíče. Počet těchto klíčů je roven počtu rund plus jedna, načež jsou xorovány do právě šifrovaného bloku dat před první rundou, mezi nimi a po poslední rundě. Detailní popisy všech algoritmů jsou v dokumentaci [20], zde nebudou rozebírány.

Na začátku šifrování je provedena operace `Key Expansion`, kdy jsou vytvořeny rundovní klíče používané během šifrování bloků.

Každý 128bitový vstupní blok textu, rozdělený do matice $4 \times 4$ bajtů, prochází následujícími operacemi:

1. `AddRoundKey` – před zahájením rundovních operací je proveden jeden průchod `AddRoundKey`, čímž dochází k naxorování všech bitů šifrovaného bloku jedním z vytvořených rundovních klíčů

2. Jednotlivé rundy

   a) `SubBytes` – substituce každého bajtu podle pevně zvolené tabulky

   b) `ShiftRows` – posouvá bajty v řádcích matice doleva

   c) `MixColumns` – bajty ve sloupcích matice jsou zkombinovány

   d) `AddRoundKey`

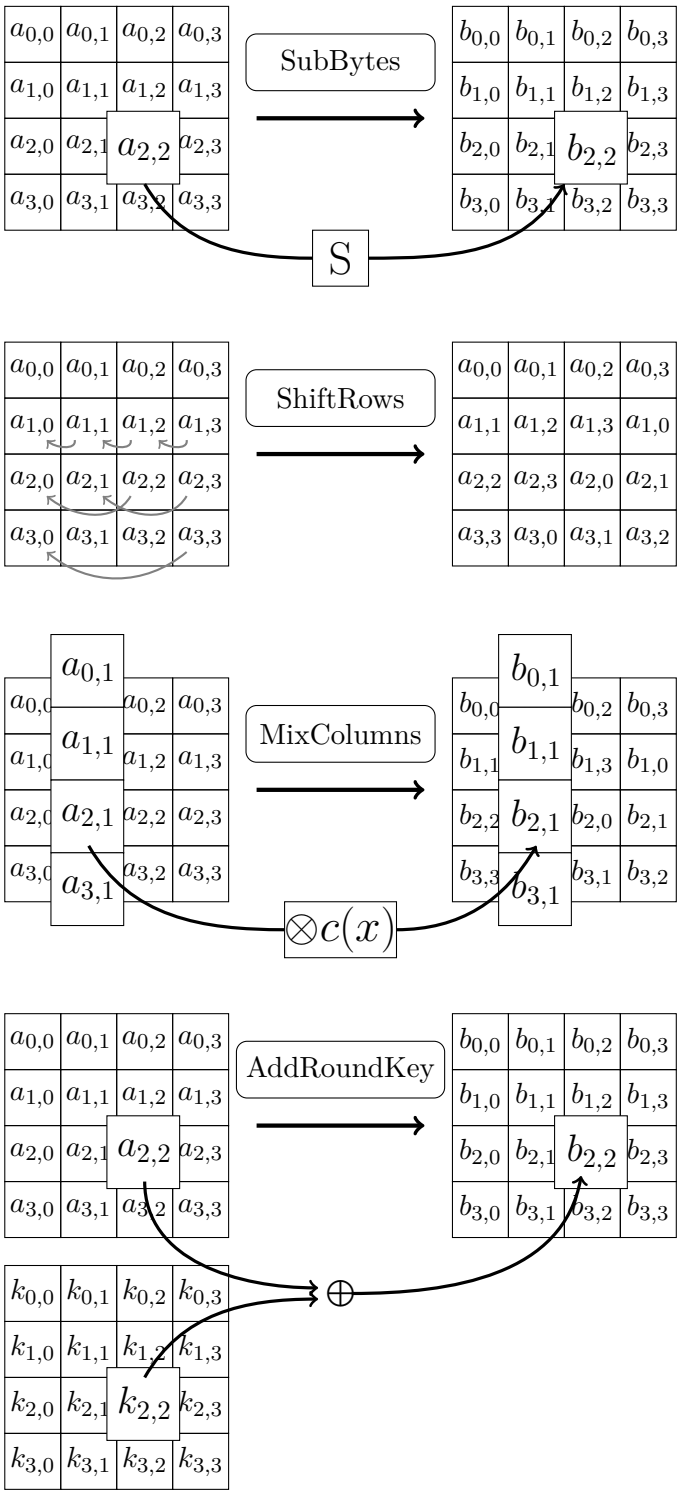
3. Poslední runda (neobsahuje operaci `MixColumns`)

   a) `SubBytes`

   b) `ShiftRows`

   c) `AddRoundKey`

Operace pro šifrování jsou znázorněny na obrázku 3.2. Pro dešifrování jsou použity inverzní operace, které probíhají v opačném pořadí. Ty jsou opět vysvětleny v dokumentaci [20].




Obrázek 3.2: Jednotlivé operace rundy AES [21, Překresleno autorem]

### 3.2.2 Twofish

Twofish je symetrická bloková šifra s délkou bloku 128 bitů a délkou klíče 128, 192, popřípadě 256 bitů. Jedná se o finalistu standardu AES. Částečně vychází z šifry Blowfish a navazuje na ni šifra Threefish, které jsou obě od stejného autora.

Jde o šifru Feistelova typu[15], využívá S-boxy (používané například v šifře AES) a počet rund je stanoven na 16. Úplný popis šifry je dohledatelný v dokumentaci [22].

### 3.2.3 Serpent

I tato šifra se dostala mezi posledních pět finalistů o standard AES a skončila jako druhá.

Velikost bloku je 128 bitů a délka klíče je libovolná – až do délky 256 bitů, kdy je zbytek do 256 bitů doplněn paddingem (jednička doplněná nulami). Počet rund je stanoven na 32. Jedná se o šifru navrženou pro paralelní zpracování a využívající substitučně-permutační síť. Detailní popis je dostupný na stránce šifry [23].

### 3.2.4 CAST5

Šifra CAST5, známá též jako CAST-128, je šifrou Feistelova typu. Jsou zde opět použity S-boxy a počet rund je stanoven na 16. Délka klíče je volitelná od 40 do 128 bitů a délka bloku je 64 bitů. Toto je v porovnání s ostatními algoritmy, které mají dvojnásobek, málo. To je jeden z důvodů, proč by tato šifra neměla být pro šifrování disku vůbec používána. Detailní specifikace šifry jsou rozebrány v dokumentaci [24].

### 3.2.5 CAST6

CAST6 je nástupcem CAST5, alternativním názvem CAST-256. Oproti svému předchůdci má vyšší počet rund, těch je zde 48, velikost bloku je 128 bitů a velikost klíče je 128 až 256 bitů. Všechny operace probíhají takřka identicky jako v případě CAST5. Šifra je popsána v [25].

## 3.3 Operační módy šifer

Operační módy blokových šifer přinášejí způsoby, jak zajistit nakládání s dalšími bloky dat, neboť blokové šifry pracují v základu jen s jedním blokem dat a klíčem. Tyto módy tedy zajišťují využití vstupního otevřeného textu

[15] Blok otevřeného textu je rozdělený na levou a pravou část, kde pravá část prochází určitou funkcí a následně je na výstup a levou část aplikována operace XOR. Načež je tento kus považován za vstup jakožto pravá část v následující rundě. Levá část je kopií původní pravé části.

(plaitext) pro vygenerování bloků šifrového textu (ciphertext), kdy je velikost bloku dat závislá na použité šifře.

U většiny operačních módů je jejich součástí inicializační vektor, který zaručí, že budeme-li šifrovat ten samý (i velmi dlouhý) otevřený text pokaždé s jiným inicializačním vektorem, dostaneme úplně jiný šifrový text.

Různé operační módy se odlišují v nakládání s otevřeným textem, aplikací inicializačního vektoru, šifrovací a dešifrovací funkcí, dokonce v nakládání s nekompletními bloky a propagací chyby při změně dat.

Seznam podporovaných operačních módů šifer:

- ECB,
- CBC-plain,
- CBC-ESSIV:*hash*,
- XTS-plain64.

### 3.3.1 ECB

Electronic Code Book je základním operačním módem a zároveň tím nejjednodušším. To s sebou přináší velké nevýhody, zejména fakt, že stejné bloky otevřeného textu mají vždy stejný šifrový obraz. Pokud nalezneme několik shodných bloků, můžeme tím za určitých okolností rozkrýt hodnotu zašifrovaného bloku, nebo alespoň získáme velmi přesnou podobu o struktuře dat.

Tento operační mód by neměl být za žádných okolností použit pro šifrování disku a jeho podpora nástrojem LUKS by měla být ihned ukončena.

### 3.3.2 CBC-plain

V módu Cipher Block Chaining je každý následující blok zkombinován s předchozím pomocí operace XOR a výstup je následně zašifrován. Toto zřetězení je vždy mezi sektory přerušeno a jako inicializační vektor je použito číslo sektoru převedené na `uint32_t` a kódované jako little endian pro zajištění kompatibility napříč architekturami. Pokud ale disk obsahuje více než $2^{32}$ sektorů, jsou stejná data zašifrována stejnou hodnotou. I toto je také důvod, proč operační mód CBC nepoužívat.

V tomto módu byla Clemensem Fruhwirthem objevena velmi závažná chyba, která byla následně popsána ve čtvrté kapitole [17]. Jedná se o tzv. „watermark attack“. Kvůli těmto slabinám je krajně nevhodné používat operační mód CBC-plain.

### 3.3.3 CBC-ESSIV:*hash*

Jako odpověď na dříve zmíněný útok na CBC byl následně Clemensem vytvořen mód CBC-ESSIV (cipher-block chaining – encrypted salt-sector initi-

alization vector), kdy je za dvojtečkou specifikována hashovací funkce. Tato funkce je použita pro vytvoření druhého klíče jako zahashované hodnoty prvního klíče. Číslo disku převedené do `uint64_t` a použitým kódováním little endian je následně zašifrováno a poté použito jako IV. Použití tohoto módu vyžaduje, aby byla délka zahashované hodnoty stejně dlouhá jako délka klíče šifry. Díky inicializaci sektorů zašifrovanou, pro útočníka neznámou, hodnotou se poté „watermark attack“ stává nemožným [26].

Nicméně další slabiny CBC popsané v [17] zůstávají, princip fungování se od CBC-plain neliší.

### 3.3.4 XTS-plain64

Operační mód XTS je složen z operačního módu XEX (XOR – Encrypt – XOR), doplněného o operaci CTS (Cipher Text Stealing). CTS umožňuje zašifrovat text, který je kratší než velikost bloku, bez změny délky výsledného zašifrovaného textu. Při šifrování posledního (kratšího) bloku je doplněn do velikosti celého bloku bity z bloku předchozího šifrového textu. Následně je celý blok zašifrován a prohozen s předposledním blokem. Poslední blok je následně oříznut na velikost vstupního textu.

Jelikož je v současnosti obvyklé používat disky převyšující velikost 2 TiB, nebo $2^{32}$ počet sektorů, je zde použita 64bitová verze módu.

Použití 128bitových bloků vylučuje použití operačního módu XTS současně se šifrou CAST5, která používá blok o délce 64 bitů.

## 3.4 Hashovací funkce

Smyslem hashovacích funkcí je převod libovolně dlouhého vstupu do výstupu určité délky. Hlavní vlastnosti těchto funkcí jsou:

- libovolná velikost vstupu,
- fixní délka výstupu,
- jednosměrnost – z výstupu nelze získat vstup,
- bezkoliznost prvního řádu – nemožnost nalezení libovolného páru vstupů, které produkují stejný výstup,
- bezkoliznost druhého řádu – nemožnost nalezení druhého konkrétního vstupu k dříve danému vstupu, kdy oba produkují stejný výstup.

Výběr funkce je proveden příkazem `--hash` nebo `-h`. U operačního módu CBC-ESSIV:*hash* musí být hashovací funkce vybrána jednou z následujících.

- SHA1,
- SHA256,
- SHA512,
- RIPEMD160.

### 3.4.1 SHA

Algoritmus SHA (Secure Hash Algorithm) je rodina hashovacích funkcí zavedená NIST a původně publikovaná v normě FIPS 180 v roce 1993 [27]. V současné době je norma nahrazena už čtvrtou revidovanou verzí [28].

SHA umožňuje zpracovávat zprávy dlouhé až $2^{64} - 1$ bitů. Velikost výstupního bloku ve skupině SHA1 je 160 bitů, ve skupině SHA-2 je to stanoveno číslem v názvu. V případě SHA256 se tedy jedná o délku 256 bitů, v případě SHA512 o délku 512 bitů. Vstupní zpráva je zpracovávána po blocích s velikostí 512 bitů s výjimkou SHA512, kdy je velikost bloku 1 024 bitů.

Nejvýznamnějším rozdílem mezi těmito funkcemi je zejména odolnost výsledného hashe vůči nalezení kolizí 1. a 2. řádu. Struktura těchto funkcí je však téměř stejná.

Pro SHA256 jsou vstupní data rozdělena na 16 bloků o slovech délky 64 bitů. Na úvodní inicializační vektor o délce 512 bitů jsou postupně aplikována vstupní data. Pro jeden vstupní blok probíhá v součtu 80 rund bitových operací AND, OR, XOR, bitová rotace, bitový posun doleva a modulární sčítání. Na výstupu je poté nový hash o délce 512 bitů. Ten je následně použit jako vstup do dalšího bloku dat. Výsledný hash je poslední takto vzniklý hash v pořadí.

### 3.4.2 RIPEMD160

RIPEMD160 patří do rodiny hashovacích funkcí RIPEMD (RIPE[16] Message Digest), která byla vyvinuta na půdě katolické univerzity v Lovani v roce 1992 [29]. Tato skupina je o poznání méně známá jak rodina SHA, přesto je využívána například v Bitcoinu a od něj odvozených kryptoměn.

I RIPEMD160 umožňuje zpracovávat zprávy dlouhé až $2^{64}-1$ bitů. Velikost výstupního bloku této funkce je 160 bitů, neboli pět 32bitových slov. Pro výpočet výsledného hashe musí projít každý blok o délce 512 bitů stejným množstvím rund jako SHA, tedy 80.

[16] Race Integrity Primitives Evaluation

Kapitola 4

# Realizace

Následující kapitola je zaměřena na implementaci stěžejních operací nástroje LUKS. Jsou zde porovnány vybrané kryptografické knihovny a poté je jednou z vybraných knihoven provedeno ověření shody implementace s dokumentací nástroje.

Implementace je realizována v jazyce C a může být zkompilována pomocí programu *gcc*, kdy je zapotřebí zahrnout všechny soubory s koncovkou „.c“. Popřípadě lze využít přiložený Makefile, kdy příkazem *make compile* dojde k přeložení nástroje.

Nástroj interně počítá s kódováním little endian, výjimkou je ukládání vícebajtových čísel. Tehdy nástroj vyžaduje kódování big endian, pro vyhovění struktury hlavičky LUKS.

Mimo standardní knihovny byly použity ještě následující knihovny:

- UUID – pro práci s UUID zařízení,
- OpenSSL – pro práci s kryptografickými primitivy,
- popt – pro zpracování příkazové řádky.

## 4.1 Výběr krypto knihovny

Pro implementaci nejdůležitějších částí nástroje LUKS bylo třeba vybrat knihovnu třetích stran, která bude podporovat výše zmíněné šifry, jejich operační módy a hashovací funkce. Z několika kandidátů byla poté vybrána knihovna OpenSSL.

Jak je vidět v tabulce 4.1, tato knihovna sice oproti ostatním nepodporuje šifru Twofish, ale její výhodou je poměrně snadné používání a tedy snadnější implementace. Podle srovnávacích tabulek 4.2 a 4.3 jsou poté rozdíly v knihovnách zanedbatelné.

Pro implementaci bychom se zároveň měli držet zásady: „Nikdy neimplementovat vlastní krypto.“ Z toho důvodu nebyly implementovány zbývající

| Knihovna | AES | Twofish | Serpent | CAST5 | CAST6 |
|---|---|---|---|---|---|
| cryptlib | ANO | ANO | NE | ANO | NE |
| Libgcrypt | ANO | ANO | NE | ANO | NE |
| OpenSSL | ANO | NE | NE | ANO | NE |
| wolfCrypt | ANO | NE | NE | NE | NE |

Tabulka 4.1: Knihovnami podporované blokové šifry

| Knihovna | ECB | CBC-plain | CBC-ESSIV | XTS-plain64 |
|---|---|---|---|---|
| cryptlib | ANO | ANO | NE | ANO |
| Libgcrypt | ANO | ANO | NE | ANO |
| OpenSSL | ANO | ANO | NE | ANO |
| wolfCrypt | ANO | ANO | NE | NE |

Tabulka 4.2: Knihovnami podporované operační módy šifer

| Knihovna | SHA-1 | SHA-2 | RIPEMD-160 |
|---|---|---|---|
| cryptlib | ANO | ANO | ANO |
| Libgcrypt | ANO | ANO | ANO |
| OpenSSL | ANO | ANO | ANO |
| wolfCrypt | ANO | ANO | ANO |

Tabulka 4.3: Knihovnami podporované hashovací funkce

kryptografické operace. Výjimkou jsou antiforenzní stripy, věc, díky které je LUKS jedinečný, jediná existující a prověřená implementace se nachází v samotném nástroji. Přesto byly implementovány dle [3, 15].

## 4.2 Hlavička

Struktura phdr realizovaná v jazyce C je takřka identická s částí popisovanou dříve v tabulce 2.1. Musí to tak být, neboť obě implementují totéž, jen každá s jinými knihovnami. Zdrojový kód hlavičky je dostupný v příloze B ve výpisu kódu 4.

## 4.3 Ověření shody

Během reimplementace byly úspěšně implementovány nejen podpůrné operace jako luksDump, ale i stěžejní operace luksFormat, která vytváří zašifrovaný LUKS kontejner.

Na základě výsledků této operace bylo zjištěno, že nástroj LUKS dodržuje svojí dokumentaci. Zároveň bylo po vložení konkrétních bajtů do proměnné

`mkDigestSalt` a na místo *master* klíče ověřeno, že obsah proměnné `mkDigest` je shodný v obou implementacích. I tyto dílčí kroky vedou k závěru, že nástroj LUKS implementuje vše přesně tak, jak tvrdí.

Ačkoliv nástroj LUKS původně využívat pouze dm-crypt, během několika posledních let byla vytvořena podpora pro další krypto knihovny, jmenovitě Libgcrypt [30], Nettle [31], Network Security Services [32] a OpenSSL [33]. Vzhledem k existenci několika alternativních knihoven, které implementují jednotlivá kryptografická primitiva, bylo rozhodnuto, že implementace dalších částí nástroje by byla neúměrně časově náročná a nepřinesla by mnoho nových poznatků.

## 4.4 Analýza kódu

Nástroj LUKS během několika posledních let narostl do obřích rozměrů, kdy v půlce května 2019 čítá na 28 tisíc řádků kódu, které se věnují pouze verzi LUKS1. Důkladné prostudování kódu je proto práce pro celý tým, jak je například vidět v provedené bezpečnostní analýze [26] z roku 2012.

Přesto bylo při vytváření implementace občas přihlédnuto k existujícím kódům. V žádné z prozkoumávaných oblastí nebyly nalezeny žádné zranitelnosti. Pokud ovšem nepočítáme jako chybu nepřetypování návratové hodnoty u dynamické alokace, např. `char * tmp = (char *)malloc(sizeof(*tmp))`.

Kapitola 5

# Útok na svazek

Tato kapitola popisuje princip útoku na LUKS kontejner a poté navrhuje dvě metody útoku. Následně je proveden útok na heslo ke svazku, změřena jeho účinnost a čtenář je seznámen s pozorovanými výsledky.

## 5.1 Útok hrubou silou

*Brute force* neboli útok hrubou silou spočívá ve zkoušení všech možných kombinací daného intervalu a testování, zda se jedná o hledané řešení problému. Prolomení krátkých hesel touto metodou je poměrně snadné a rychlé, ale heslo obsahující více než 8 znaků je pro běžného útočníka vybaveného jedním strojem časově velmi náročné.

V tabulce 5.1 je zaznamenán teoretický celkový čas pro vyzkoušení všech možností pro počet znaků pokrývající pouze čísla, pouze malá písmena a kombinaci malých písmen a čísel. V této tabulce je použit předpoklad, že jsme schopni vyzkoušet 100 hesel za sekundu.

| Délka hesla | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|
| Čas pro [0–9] | 2m | 16m | 3h | 28h | 12d |
| Čas pro [a–z] | 1h | 33h | 35d | 29měs. | 66r |
| Čas pro [a–z; 0–9] | 5h | 6d | 8měs. | 25r | 900r |

Tabulka 5.1: Porovnání síly hesla

### 5.1.1 Bash skript

Pravděpodobně nejjednodušší způsob pro zjištění správného hesla ke svazku je vytvoření jednoduchého skriptu, kdy je využita funkce `open`, která vrací návratovou hodnotu `0` pro signalizaci úspěchu a hodnotu `2` pro značení špatného

hesla. Použitím přepínače `--test-passphrase` není svazek aktivován, dochází pouze k ověření hesla. Útočníkovi tedy postačí testovat návratovou hodnotu na úspěch operace. Základní myšlenka skriptu může vypadat podobně jako ve výpisu kódu 1. Pokud navíc chceme skript rozšířit z testování pouze čísel na test čísel a písmen (popřípadě dalších skupin), provedeme jednoduchou úpravu v cyklu for z původního rozsahu `{0..9}` na rozsah `{{a..z},{0..9}}`.

```
for i in {0..9}{0..9}{0..9}{0..9}; do
    echo "$i" | cryptsetup open --test-passphrase <device>
    if [ $? -eq 0 ]; then
        echo "Password: $i"
        exit 0
    fi
done
echo "Password not found"
```

Výpis kódu 1: Skript pro útok hrubou silou

### 5.1.2 C program

Můžeme také využít znalosti zdrojového kódu nástroje cryptsetup a použít jeho knihovní funkci pro ověření hesla v daném svazku. V našem případě se jedná o funkci pro aktivaci heslem, kdy `crypt_activate_by_passphrase` vrací záporné číslo při zadaném nesprávném heslu a kladné číslo v rozmezí `0` až `7` pro úspěšně ověřené heslo. Toto číslo nám zároveň označuje i slot patřící onomu heslo.

Funkce `crypt_activate_by_passphrase` umožňuje vytvořit nový cryptsetup kontejner, popřípadě ověřit jeho heslo. Jako vstupní parametry vyžaduje aktivní LUKS zařízení, jméno nového zařízení (nebo hodnotu `NULL` pouze pro kontrolu hesla). Dalším parametrem je konstanta `CRYPT_ANY_SLOT` nebo číslo v rozmezí `0` až `7` pro kontrolu zadaného slotu. Následuje samotné heslo, jeho velikost a příznaky. V našem případě použijeme `CRYPT_ACTIVATE_READONLY` pro označení zařízení pouze pro čtení.

```
int crypt_activate_by_passphrase(struct crypt_device * cd,
                                 const char * name,
                                 int keyslot,
                                 const char * passphrase,
                                 size_t passphrase_size,
                                 uint32_t flags);
```

Výpis kódu 2: Rozhraní funkce pro aktivaci LUKS kontejneru

Program, který je součástí této práce, je napsán jako jednovláknová aplikace, kdy při mírné modifikaci kódu pro podporu více vláken – popřípadě paralelním běhu více instancí programu – lze výsledky podstatně zlepšit a získat heslo mnohem efektivněji. Nevýhoda tohoto řešení je závislost na knihovně `libcprytsetup-dev`, která umožňuje používat funkce nástroje cryptsetup.

### 5.1.3 Výsledky útoku hrubou silou

Tabulka 5.2 ukazuje naměřené časy bash skriptu, C programu a porovnává je s teoretickým výpočtem času potřebného pro provedení útoku hrubou silou. Na základě výsledků tabulky 5.2 lze pozorovat, že bash skript je o poznání pomalejší než C program, proto je v tabulkách 5.3 a 5.4 porovnáván pouze C program.

Vyšší čas nutný pro průběh skriptu je zřejmě zaviněn vyšší režií funkce `open`, kdy dochází k inicializaci a ověření zařízení během každého průchodu cyklem. V C programu je tento krok proveden pouze při začátku útoku.

Zároveň můžeme považovat informace v tabulce 5.1 jako poměrně přesný odhad trvání útoku při 1 000 iterací *master* klíče a jednotlivých slotů. Tyto časy však zaznamenávají potřebný čas pro vyzkoušení všech možností v daném intervalu, v průměru bude ovšem heslo prolomeno za polovinu této doby.

| Délka hesla [0-9] | 4 | 5 | 6 | 7 |
|---|---|---|---|---|
| Naměřený čas – C | 1m 31s | 14m 58s | 2h 42m | 27h 2m |
| Naměřený čas – bash | 2m | 20m 46s | 3h 49m | 33h 19m |
| Teoretický čas | 1m 40s | 16m 40s | 2h 47m | 27h 47m |

Tabulka 5.2: Naměřené časy pro získání hesla

V tabulce 5.3 je vidět závislost rychlosti útoku hrubou silou na počtu aktivních *key slotů*. Pro každý slot musí být vypočítáno unikátní heslo, protože každý ze slotů má totiž unikátní sůl. Pro toto porovnání byl zvolen počet iterací *master* klíče i jednotlivých slotů na 1 000 iterací.

| Počet slotů | Čas | Počet hesel za sekundu |
|---|---|---|
| 1 | 12s | 90 |
| 2 | 24s | 51 |
| 4 | 48s | 25 |
| 8 | 1m 48s | 11 |

Tabulka 5.3: Porovnání útoku pro čtyřznakové heslo

Tabulka 5.4 naopak ukazuje nárůst času potřebného pro vyzkoušení dostatečného počtu hesel při zvyšujícím se počtu iterací slotu. Pro tuto ukázku byla jako heslo zvolena čtveřice čísel `1234`.

| Počet iterací MK | Počet iterací slotu | Čas | Počet hesel za sekundu |
|---|---|---|---|
| 1 000 | 10 000 | 23s | 53 |
| 6 250 | 100 000 | 1m 45s | 11 |
| 62 500 | 1 000 000 | 15m 50s | 1 |
| 186 712 | 2 987 396 | 44m 8s | 0,5 |

Tabulka 5.4: Nárůst času potřebného pro získání hesla

## 5.2 Slovníkový útok

Slovníkový útok spočívá ve zkoušení hesel z předem předpřipraveného seznamu, neboli slovníku. Ve většině případů se jedná o efektivnější metodu než útok hrubou silou. Ve slovníkovém útoku ovšem není zaručeno, že se nám podaří heslo získat. Pro zvýšení účinnosti se proto používá slovníkový útok s pravidly, kdy jsou vytvořena pravidla aplikující se na hesla ve slovníku.

Dnes je navíc uživatel ze všech stran upozorňován na zásady při tvorbě hesla (např. [34]) zahrnující pravidla:

- alespoň 8 znaků,
- velké písmeno,
- malé písmeno,
- číslo,
- speciální znak.

Aplikací těchto pravidel můžeme vytvořit heslo `Password1!`, které na pohled vypadá silně. Pokud bychom ho zkoušeli prolomit metodou *brute force*, ke správnému heslu bychom se dostali v průběhu několika milionů let.

### 5.2.1 Bash skript

Základní myšlenka skriptu pro slovníkový útok je prakticky totožná s myšlenkou popsanou ve výpisu kódu 1. Opět je využita funkce `open` a testována návratová hodnota. Tentokrát je však na vstupu heslo ze slovníku.

Hesla jsou do slovníku přidávána z dřívějších úniků hesel, pro naše účely použijeme seznam nejčastějších hesel, například [35] a získáme tím slovník. Následně vytvoříme seznam pravidel, podle výše uvedeného seznamu pro tvorbu bezpečného hesla. Pravidla mohou být podobná následujícím: `$[0-9]` (přidání číslice na konec hesla), `$[!-/]` (přidání speciálního znaku na konec hesla) a `c` (zvětšení prvního písmena v heslu). Ve skutečnosti jsou ovšem používány mnohem obsáhlejší slovníky s miliony slov a sady pravidel obsahující statisíce pravidel.

Pravidla a slovník předáme programu John the Ripper [36] a necháme vygenerovat seznam nových hesel s aplikovanými pravidly. Výsledný skript poté může vypadat jako skript ve výpisu kódu 3.

```
john --wordlist=<wordlist> --stdout --rules > <dictionary>
for i in $(cat <dictionary>); do
    echo "$i" | cryptsetup open --test-passphrase <device>
    if [ $? -eq 0 ]; then
        echo "Password: $i"
        rm <dictionary>
        exit 0
    fi
done
echo "Password not found"
rm <dictionary>
```

Výpis kódu 3: Skript pro slovníkový útok

### 5.2.2 Výsledky slovníkového útoku

Při nejnižším počtu iterací klíče a *key slotu* jsme dokázali heslo prolomit téměř ihned. Jak je vidět v tabulce 5.5, i při zvyšujícím se počtu iterací bylo heslo prolomeno během chvíle.

| Počet iterací MK | Počet iterací slotu | Čas | Počet hesel za sekundu |
|---|---|---|---|
| 1 000 | 1 000 | 2s | 82 |
| 1 000 | 10 000 | 3s | 54 |
| 6 250 | 100 000 | 13s | 12 |
| 62 500 | 1 000 000 | 2m | 1 |
| 187 245 | 2 995 930 | 5m 56s | 0,5 |

Tabulka 5.5: Nárůst času potřebného pro slovníkový útok

## 5.3 Diskuze výsledků

Veškeré výpočty byly provedeny jako jednovláknová aplikace na průměrném notebooku s procesorem `Intel i5-7200U`. Pokud bychom použili výkonnější počítač nebo server s více procesory, zcela jistě bychom se dostali k jiným výsledkům a heslo bychom prolomili rychleji.

O výsledné délce prováděného útoku a jeho úspěšnosti rozhoduje kombinace několika faktorů

- počet iterací *master* klíče,
- počet iterací *key* slotu,
- délka hesla a jeho rozmanitost,
- počet aktivních slotů.

Počet iterací klíče a slotu je spolu velmi úzce provázán a rozhodně by neměl být snižován pod hranici doporučovanou benchmarkem. Naopak zvýšení počtu iterací také není nejlepší volba. Dochází k omezování uživatele a velmi snadno lze přesáhnout hranice trpělivosti uživatelů, kteří by se mohli pokusit počet iterací snížit pod bezpečnou hranici.

Základem pro bezpečnost je dostatečně dlouhé a silné heslo, které by podle nového doporučení NIST [37] nemělo být příliš komplexní. Místo toho upřednostňuje tzv. *passphrase* – heslovitou frázi o několika náhodně zvolených slovech.

Ačkoliv tabulka 5.3 ukazuje vyšší čas potřebný pro průběh útoku při vyšším počtu slotů, neměli bychom zapomínat na možnost zaútočit pouze na konkrétní slot.

Přes všechny teoretické výpočty bychom měli pamatovat na pravidelné přešifrování disku, abychom se vyhnuli prolomení hesla a udrželi informace zabezpečené.

# Závěr

Práce měla za cíl seznámit čtenáře s problematikou šifrování diskových oddílů a nástrojem Linux Unified Key Setup. Zároveň bylo provedeno srovnání nástroje LUKS s ostatními konkurenčními nástroji.

Tohoto cíle bylo dosaženo v kapitole 1, která se věnuje historii nástroje LUKS a porovnává ho s konkurencí, kapitole 2, která se zabývá seznámením s jednotlivými operacemi nástroje LUKS, a kapitole 3, která popisuje jednotlivé šifry, jejich operační módy a hashovací funkce, které poskytují důkladné teoretické pozadí pro pochopení práce.

Dalším úkolem této bakalářské práce bylo prozkoumat zdrojový kód nástroje LUKS a nalézt místa s možným bezpečnostním dopadem. V programu LUKS byly nalezeny zranitelnosti, týkající se podpory a používání operačních módů ECB a CBC. Tyto operační módy by nikdy neměly být použity pro šifrování pevných disků. Nástroj LUKS by měl od jejich používání upustit. Zároveň by nadále neměla být podporována šifra CAST5, která neumožňuje použití operačního módu XTS.

Kapitola 4 je zaměřena na reimplementaci nejdůležitějších částí nástroje LUKS. Díky reimplementaci funkce luksFormat, která se zabývá vytvořením nového svazku, byla ověřena shodnost implementace nástroje LUKS a jeho popisu v dokumentaci.

Reimplementace nástroje pro mne byla velmi přínosná. Při práci jsem se dozvěděl nejen jak detailně fungují samotné šifrovací aplikace, ale rozšířil jsem si své znalosti na poli šifrovacích algoritmů a operačních módů.

Silnou stránkou nástroje LUKS je jeho odolnost vůči slovníkovým útokům. Toho je dosaženo použitím velkého množství iterací klíče ke svazku. Ačkoliv má uživatel nad množstvím iterací kontrolu, neměl by jí využívat. Měl by naopak ponechat hodnotu stanovenou interním měřením. Jak je ukázáno v kapitole 5, při nižším počtu iterací je klíč ke svazku prolomitelný v řádu minut až hodin. Přesto je potřeba používat silná hesla a po určité době přešifrovat celý disk.

Pro účely útoku hrubou silou a slovníkového útoku byly vytvořeny bash skripty a program v jazyce C, které umožňují prolomení hesla ke svazku nástroje LUKS.

Tato bakalářská práce je zaměřena na LUKS první verze, druhá verze však zůstává upozaděna. V budoucnu by bylo určitě možné provést analýzu druhého standardu. Ten využívá pokročilejší znalosti z oblasti kryptografie, proto by se toto téma dalo použít pro diplomovou práci.

# Bibliografie


1. About clefru. *Clefru's blog* [online] [cit. 2019-04-07]. Dostupné z: `http://clemens.endorphin.org/p/about-me.html`.
2. Global digital population as of January 2019. *Statista* [online] [cit. 2019-04-17]. Dostupné z: `https://www.statista.com/statistics/617136/digital-population-worldwide/`.
3. FRUHWIRTH, Clemens. *TKS1: An anti-forensic, two level, and iterated key setup scheme* [online]. 2004 [cit. 2019-04-17]. Dostupné z: `https://www.kernel.org/pub/linux/utils/cryptsetup/LUKS_docs/TKS1-draft.pdf` Citovaný text přeložen autorem.
4. BROŽ, Milan. *Authenticated and Resilient Disk Encryption*. Brno, 2018. Dostupné také z: `https://is.muni.cz/th/vesfr/final.pdf`. Doktorandská práce. Masarykova Univerzita, Fakulta informatiky. Vedoucí práce Václav Matyáš.
5. BROŽ, Milan; KOZINA, Ondřej. The future of disk encryption with LUKS2 - Milan Brož a Ondřej Kozina. In: *Youtube* [online]. 2016 [cit. 2019-04-07]. Dostupné z: `https://youtu.be/XmFUb6qP9M0`. Kanál uživatele DevConf.
6. AORIMN. *Dislocker Version 0.7* [software]. 2012 [cit. 2019-04-16]. Dostupné z: `https://github.com/Aorimn/dislocker`. Gitový repozitář projektu, hash commitu 3939a18.
7. BROŽ, Milan. *Šifrování disků…: (nejen) v Linuxu* [online]. 2011 [cit. 2019-04-18]. Dostupné z: `https://mbroz.fedorapeople.org/talks/Europen2011/` Slajdy z přednášky 39. konference EurOpen.
8. BROŽ, Milan. Šifrování disků (nejen) v Linuxu. In: *Sborník příspěvků z 39. konference EurOpen.CZ*. Plzeň: EurOpen.CZ, 2011, s. 57–68. ISBN 978-80-86583-22-8. Dostupné také z: `https://www.europen.cz/Anot/39/eo-2-11.pdf`.

9. *VeraCrypt. Version 1.23* [software]. 2018 [cit. 2019-05-14]. Dostupné z: https://www.veracrypt.fr/en/Home.html.

10. *TrueCrypt. Version 7.2* [software]. 2014 [cit. 2019-05-14]. Dostupné z: http://truecrypt.sourceforge.net/.

11. *Bitlockerj* [software]. 2018 [cit. 2019-05-14]. Dostupné z: https://docs.microsoft.com/cs-cz/windows/security/information-protection/bitlocker/bitlocker-overview.

12. PHIPPS, Simon. TrueCrypt or false? Would be open source project must clean up its act. *InfoWorld* [online]. 2013 [cit. 2019-04-16]. Dostupné z: https://www.infoworld.com/article/2609745/truecrypt-or-false--would-be-open-source-project-must-clean-up-its-act.html.

13. BROŽ, Milan. *Frequently Asked Questions* [online]. 2018 [cit. 2019-04-16]. Dostupné z: https://gitlab.com/cryptsetup/cryptsetup/wikis/FrequentlyAskedQuestions Sekce 2.14.

14. CRYPTSETUP. *Cryptsetup. Version 2.1.0* [software]. 2004 [cit. 2019-04-07]. Dostupné z: https://gitlab.com/cryptsetup/cryptsetup/. Gitlabový repozitář projektu, hash commitu 428e6125.

15. FRUHWIRTH, Clemens. *LUKS1 On-Disk Format Specification* [online]. 2018. Verze 1.2.3 [cit. 2019-05-03]. Dostupné z: https://gitlab.com/cryptsetup/cryptsetup/blob/master/docs/on-disk-format.pdf.

16. dm-crypt/Device encryption. In: *Archlinux* [online] [cit. 2019-04-13]. Dostupné z: https://wiki.archlinux.org/index.php?title=Dm-crypt/Device_encryption&oldid=570730.

17. FRUHWIRTH, Clemens. *New methods in hard disk encryption* [online]. 2005 [cit. 2019-04-29]. Dostupné z: http://clemens.endorphin.org/nmihde/nmihde-A4-ds.pdf.

18. KALISKI, Burt. *PKCS# 5: Password-based cryptography specification version 2.0* [online]. 2000 [cit. 2019-04-14]. Dostupné z: https://www.rfc-editor.org/rfc/pdfrfc/rfc2898.txt.pdf.

19. CHEN, Lily. Cryptographic Standards and Guidelines: AES Development. *NIST* [online] [cit. 2019-05-02]. Dostupné z: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development.

20. PUB, NIST FIPS. 197: Advanced encryption standard (AES). *Federal information processing standards publication* [online]. 2001, roč. 197, č. 441, s. 0311 [cit. 2019-04-07]. Dostupné z: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

21. WIKIMEDIA, Commons. *AES SubBytes, ShiftRows, MixColumns, AddRoundKey* [online]. 2005 [cit. 2019-05-02]. Dostupné z: https://commons.wikimedia.org/wiki/Category:AES.

22. SCHNEIER, Bruce. *Twofish: A 128-Bit Block Cipher* [online]. 1998 [cit. 2019-05-02]. Dostupné z: https://www.schneier.com/academic/paperfiles/paper-twofish-paper.pdf.

23. ANDERSON, Ross; BIHAM, Eli; KNUDSEN, Lars. *Serpent: A Proposal for the Advanced Encryption Standard* [online]. 2006 [cit. 2019-05-02]. Dostupné z: https://www.cl.cam.ac.uk/~rja14/serpent.html.

24. ADAMS, Carlisle. *The CAST-128 encryption algorithm* [online]. 1997 [cit. 2019-05-02]. Dostupné z: https://tools.ietf.org/html/rfc2144.

25. ADAMS, Carlisle. *The CAST-256 Encryption Algorithm* [online]. 1999 [cit. 2019-05-02]. Dostupné z: https://tools.ietf.org/html/rfc2612.

26. TEAM, Ubuntu Privacy Remix. *Security Analysis of Cryptsetup/LUKS* [online]. 2012 [cit. 2019-05-02]. Dostupné z: https://www.privacy-cd.org/analysis/cryptsetup_1.4.1-luks-analysis-en.pdf.

27. FIPS 180: Secure Hash Standard (SHS). *NIST* [online] [cit. 2019-05-02]. Dostupné z: https://csrc.nist.gov/publications/detail/fips/180/archive/1993-05-11.

28. FIPS 180-4: Secure Hash Standard (SHS). *NIST* [online] [cit. 2019-05-02]. Dostupné z: https://csrc.nist.gov/publications/detail/fips/180/4/final.

29. DOBBERTIN, Hans; BOSSELAERS, Antoon; PRENEEL, Bart. *RIPEMD160: A Strengthened Version of RIPEMD* [online]. 1992 [cit. 2019-05-02]. Dostupné z: https://homes.esat.kuleuven.be/~bosselae/ripemd160/pdf/AB-9601/AB-9601.pdf.

30. *Libgcrypt. Version 1.8.4* [software]. 2018 [cit. 2019-05-14]. Dostupné z: https://www.gnupg.org/software/libgcrypt/index.html.

31. *Nettle. Version 3.4* [software]. 2017 [cit. 2019-05-14]. Dostupné z: http://www.lysator.liu.se/~nisse/nettle/.

32. *Network Security Services. Version 3.43* [software]. 2019 [cit. 2019-05-14]. Dostupné z: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS.

33. *OpenSSL. Version 1.1.0j* [software]. 2018 [cit. 2019-05-14]. Dostupné z: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS.

34. HOFFMAN, Chris. *How hackable is your password?* [online]. 2018 [cit. 2019-05-14]. Dostupné z: https://www.howtogeek.com/195430/how-to-create-a-strong-password-and-remember-it/.

35. PICHETA, Rob. *How hackable is your password?* [online]. 2019 [cit. 2019-05-14]. Dostupné z: `https://edition.cnn.com/2019/04/22/uk/most-common-passwords-scli-gbr-intl/index.html`.

36. *John the Ripper. Version 1.8.0* [software]. 2013 [cit. 2019-05-14]. Dostupné z: `https://www.openwall.com/john/`.

37. NIST. *Digital Identity Guidelines: Now Available* [online]. 2017 [cit. 2019-05-02]. Dostupné z: `https://pages.nist.gov/800-63-3/`.

Příloha A

# Seznam použitých zkratek

**AES** Advanced Encryption Standard

**AES-NI** Advanced Encryption Standard New Instructions

**ASCII** American Standard Code for Information Interchange

**CBC** Cipher Block Chaining

**CBC-ESSIV** Cipher Block Chaining – Encrypted Salt-Sector Initialization Vector

**CTS** Cipher Text Stealing

**ECB** Electronic Code Book

**JSON** JavaScript Object Notation

**LUKS** Linux Unified Key Setup

**MBR** Master Boot Record

**NIST** National Institute of Standards and Technology

**NTFS** New Technology File System

**PBKDF2** Password-Based Key Derivation Function 2

**TPM** Trusted Platform Module

**USB** Universal Serial Bus

**XEX** Xor–Encrypt–Xor

**XTS** XEX-based tweaked-codebook mode with ciphertext stealing

Příloha B

# Hlavička LUKS v jazyce C

```
struct phdr {
    char     magic[LUKS_MAGIC_L];
    uint16_t version;
    char     cipherName[LUKS_CIPHERNAME_L];
    char     cipherMode[LUKS_CIPHERMODE_N];
    char     hashSpec[LUKS_HASHSPEC_L];
    uint32_t payloadOffset;
    uint32_t keyBytes;
    char     mkDigest[LUKS_DIGESTSIZE];
    char     mkDigestSalt[LUKS_SALTSIZE];
    uint32_t mkDigestIterations;
    char     uuid[UUID_STRING_L];
    key_slot keyblock[LUKS_NUMKEYS];
    char     _padding[432];
};

struct key_slot {
    uint32_t active;
    uint32_t passwordIterations;
    char     passwordSalt[LUKS_SALTSIZE];
    uint32_t keyMaterialOffset;
    uint32_t stripes;
};
```

Výpis kódu 4: Hlavička LUKS1 v jazyce C

```
struct luks2_hdr_disk {
    char     magic[MAGIC_L];
    uint16_t version;
    uint64_t hdr_size;
    uint64_t seqid;
    char     label[LABEL_L];
    char     csum_alg[CHECKSUM_ALG_L];
    uint8_t  salt[SALT_L];
    char     uuid[UUID_L];
    char     subsystem[LABEL_L];
    uint64_t hdr_offset;
    char     _padding[184];
    uint8_t  csum[CHECKSUM_L];
    char     _padding4096[7*512];
} __attribute__ ((packed));
```

Výpis kódu 5: Hlavička LUKS2 v jazyce C

Příloha C

# Konstanty

| Konstanta | Hodnota | Popis |
|---|---|---|
| LUKS_MAGIC | {'L','U','K','S', 0xBA,0xBE} | magická konstanta LUKS kontejneru |
| LUKS_DIGESTSIZE | 20 | kontrolní součet pro master klíč |
| LUKS_SALTSIZE | 32 | délka PBKDF2 soli |
| LUKS_NUMKEYS | 8 | počet key slotů |
| LUKS_KEY_DISABLED | 0x0000DEAD | neaktivní key slot |
| LUKS_KEY_ENABLED | 0x00AC71F3 | aktivní key slot |
| LUKS_STRIPES | 4 000 | počet antiforenzních stripů pro *AFsplit* |
| LUKS_ALIGN_KEYSLOTS | 4 096 | zarování pro key sloty v bajtech |
| LUKS_SECTOR_SIZE | 512 | LUKS1 používá velikost sektoru 512 bajtů |
| LUKS_MKD_ITER_MIN | 1 000 | minimum iterací master klíče pro PBKDF2 |

Tabulka C.1: Konstanty v nástroji LUKS

Příloha D

# Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `attack`
    - `brute` ..... zdrojové soubory pro útok hrubou silou
    - `dict` ..... zdrojové soubory pro slovníkový útok
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF