

Reverzní inženýrství pro desktopové C/C++ aplikace

Bc. Jan Andrýsek



ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: Bc. Jan Andrášek
Osobní číslo: A23421
Studijní program: N0613A140022 Informační technologie
Specializace: Softwarové inženýrství
Forma studia: Kombinovaná
Téma práce: Reverzní inženýrství pro desktopové C/C++ aplikace
Téma práce anglicky: Reverse Engineering for Desktop C/C++ Applications

Zásady pro vypracování

- Provedte literární rešerši moderních metod reverzního inženýrství, které se používají pro dekompilaci aplikací napsaných v jazyce C/C++.
- Analyzujte dostupné nástroje pro reverzní inženýrství a jejich efektivitu při získávání zdrojového kódu.
- Vyberte a charakterizujte několik aplikací různého typu, které budou použity pro praktické ověření metod.
- Aplikujte vybrané metody reverzního inženýrství na tyto aplikace a vyhodnotte výsledky.
- Porovnejte efektivitu jednotlivých metod na základě získaných výsledků.
- Na základě zjištěných poznatků vypracujte obecná doporučení pro reverzní inženýrství u dalších aplikací.

Seznam doporučené literatury:

1. TUTORIALSPOINT. *Assembly Programming Tutorial*. Online. C2024. Dostupné z: https://www.tutorialspoint.com/assembly_programming/index.htm. [cit. 2024-10-22].
2. *Welcome to Hex-Rays docs*. Online. 2024-10-22. C2024, 2024-10-01. Dostupné z: <https://docs.hex-rays.com/git-book/pdf>. [cit. 2024-10-22].
3. ANDRIESSE, Dennis. *Practical binary analysis: build your own Linux tools for binary instrumentation, analysis, and disassembly*. San Francisco: No Starch Press, [2019]. ISBN 978-1-59327-912-7.
4. YURICHEV, Dennis. *Understanding Assembly Language*. Online. C2013-2015. Dostupné z: <https://bdigital.uvhm.edu.mx/wp-content/uploads/2020/07/Reverse-Engineering-for-Beginners.pdf>. [cit. 2024-10-22].
5. WONG, Reginald. *Mastering Reverse Engineering*. Online. Packt Publishing Limited, 2018. ISBN 978-1788838849. Dostupné také z: <https://download.packt.com/free-ebook/9781788838849>.
6. DANG, Bruce; GAZET, Alexandre; BACHAALANY, Elias a JOSSE, Sébastien. *Practical reverse engineering: x86, x64, ARM, Windows Kernel, reversing tools, and obfuscation*. Indianapolis, IN: John Wiley, 2014. ISBN 978-1-118-78725-0. Dostupné také z: <https://ebookcentral.proquest.com/lib/natl-ebooks/detail.action?docID=1629173>.
7. SIKORSKI, Michael a HONIG, Andrew. *Practical malware analysis: the hands-on guide to dissecting malicious software*. San Francisco: No Starch Press, c2012. ISBN 1593272901. Dostupné také z: <https://ebookcentral.proquest.com/lib/natl-ebooks/detail.action?docID=1137570>.
8. *Reverse Engineering For Everyone!*. Online. 2021. Dostupné z: <https://0xinfection.github.io/reversing/>. [cit. 2024-10-22].

Vedoucí diplomové práce:

Ing. Pavel Pokorný, Ph.D.

Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: 27. října 2024

Termín odevzdání diplomové práce: 2. června 2025



L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE

Beru na vědomí, že

- odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení diplomové práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – diplomovou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá; neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji, že

- jsem na diplomové práci pracoval(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
- odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.

Ve Zlíně, dne

.....
podpis autora

ABSTRAKT

Práce se zabývá reverzním inženýrstvím (Reverse Engineering (Reverzní inženýrství) (RE)) C/C++ aplikací, hodnotí jeho metody, nástroje a výzvy. Teoretický přehled doplňuje praktická analýza tří C/C++ úloh (CTF, shellcode, optimalizovaná C++ OOP aplikace) s využitím nástrojů Ghidra a experimentálně Artificial Intelligence (Umělá inteligence) (AI). Výsledky potvrzují ztížení RE vlivem optimalizací a absence symbolů, zejména u C++ OOP, a nezbytnost kombinace statické/dynamické analýzy. AI má asistentský potenciál, vyžaduje však validaci. Expertní úsudek zůstává pro náročné RE C/C++ klíčový, budoucnost je v synergii s AI.

Klíčová slova: Reverzní inženýrství, C++, Dekompilace, Statická analýza, Dynamická analýza, Optimalizace překladačem, Umělá inteligence v RE

ABSTRACT

This thesis addresses reverse engineering (RE) of C/C++ applications, evaluating its methods, tools, and challenges. A theoretical overview complements practical analysis of three C/C++ tasks (CTF, shellcode, optimized C++ OOP application) using tools like Ghidra and, experimentally, AI. Results confirm RE difficulty increases due to optimizations and lack of symbols, especially for C++ OOP, and the necessity of combined static/dynamic analysis. AI shows assistive potential but requires validation. Expert judgment remains crucial for demanding C++ RE; the future lies in synergy with AI.

Keywords: Reverse Engineering, C++, Decompilation, Static Analysis, Dynamic Analysis, Compiler Optimizations, Artificial Intelligence in RE

Rád bych poděkoval vedoucímu mé diplomové práce panu Pavlu Pokornému, Ph.D., za cenné rady, odborné vedení a trpělivost. Děkuji také rodině a přátelům za podporu během celého studia. Děkuji Vám.

OBSAH

ÚVOD	13
I TEORETICKÁ ČÁST	16
1 REVERZNÍ INŽENÝRSTVÍ	17
1.1 CÍLE A MOTIVACE REVERZNÍHO INŽENÝRSTVÍ	17
1.2 TERMINOLOGIE A SOUVISEJÍCÍ POJMY OBORU	19
1.3 METODOLOGIE REVERZNÍ INŽENÝRSKÉ PRAXE.....	20
1.3.1 Statická analýza	21
1.3.2 Analýza Dynamická	21
1.4 VÝZNAM REVERZNÍHO INŽENÝRSTVÍ V ŽIVOTNÍM CYKLU SOFTWARE	23
1.5 HISTORICKÝ VÝVOJ A KLÍČOVÉ MOMENTY	23
1.6 SOFTWAREOVÉ INŽENÝRSTVÍ JAKO ZÁKLAD REVERZNÍ ANALÝZY.....	24
1.7 SPECIFIKA A VÝZVY REVERZNÍHO INŽENÝRSTVÍ C/C++ APLIKACÍ.....	26
1.7.1 Optimalizace překladačem	26
1.7.2 Absence metadat a reflexe.....	26
1.7.3 Komplexita jazyka C++.....	27
1.7.4 Name mangling	28
1.7.5 Přímá práce s pamětí a ukazateli.....	28
1.7.6 Použití inline assembleru a nízkourovňových konstrukcí	28
1.7.7 Obfuskace a anti-reverzní techniky	28
2 MODERNÍ METODY A NÁSTROJE REVERZNÍ ANALÝZY C/C++ APLIKACÍ..	34
2.1 PŘEHLED MODERNÍCH METOD REVERZNÍHO INŽENÝRSTVÍ	34
2.1.1 Metody Založené na Umělé Inteligenci a Strojovém Učení (AI/ML)	34
2.1.2 Modelově Řízené Reverzní Inženýrství	35
2.1.3 Pokročilé Statické a Dynamické Analytické Techniky	36
2.2 ANALÝZA NÁSTROJŮ PRO REVERZNÍ INŽENÝRSTVÍ C/C++ A JEJICH	
EFEKTIVITA PŘI ZÍSKÁVÁNÍ ZDROJOVÉHO KÓDU	38
2.2.1 Komerční a Open-Source Dekompilátory	38
2.2.2 Nástroje s Podporou AI a Jejich Integrace.....	42

2.2.3	Specializované Nástroje a Frameworky	43
2.2.4	Efektivita a Omezení Současných Nástrojů při Získávání Kódu	44
2.3	VÝBĚR A CHARAKTERISTIKA APLIKACÍ RŮZNÉHO TYPU PRO PRAK- TICKÉ OVĚŘENÍ METOD	45
2.3.1	Analýza Malware (Škodlivého Softwaru)	45
2.3.2	Výzkum Zranitelností v Proprietárním Softwaru	47
2.3.3	Analýza Firmware Vestavěných Zařízení (<i>Embedded Systems</i>)	48
2.3.4	Reverzní Inženýrství Herních Enginů a Anti-Tamper Mechanizmů	49
2.3.5	Analýza Staršího (<i>Legacy</i>) Softwaru.....	50
2.3.6	Úlohy z Capture The Flag (Soutěž typu "Získej vlajku") (CTF) Soutěží (<i>Capture The Flag</i>)	51
2.4	VYHODNOCENÍ VÝSLEDKŮ APLIKOVANÝCH METOD REVERZNÍHO IN- ŽENÝRSTVÍ	52
2.4.1	Metriky pro Hodnocení Kvality Dekompilace.....	52
2.4.2	Výsledky a Přesnost AI/ML Metod	53
2.4.3	Výzvy při Interpretaci Výsledků.....	54
2.4.4	Vliv Optimalizací Kompilátoru na Výsledky Dekompilace	55
2.5	POROVNÁNÍ EFEKTIVITY RŮZNÝCH METOD REVERZNÍHO INŽENÝRSTVÍ	56
2.5.1	Srovnání Tradičních Dekompilačních Technik s Metodami Založenými na AI/ML	56
2.5.2	Výhody a Nevýhody Jednotlivých Přístupů v Kontextu C/C++	57
2.5.3	Analýza Nákladů a Přínosů	58
2.5.4	Dopad na Časovou Náročnost a Potřebné Expertní Znalosti	59
2.6	OBECNÁ DOPORUČENÍ A BUDOUCÍ SMĚRY V REVERZNÍM INŽENÝR- STVÍ C/C++	60
2.6.1	Doporučené Postupy pro Reverzní Analýzu Komplexních C/C++ Apli- kací	60
2.6.2	Strategie pro Překonávání Obfuskace a Anti-Reverzních Technik	61
2.6.3	Význam Porozumění Moderním C++ Rysům (C++20/23) pro Reverzní Inženýrství	63
2.6.4	Očekávaný Vývoj a Trendy v Oblasti	63

2.6.5	Doporučení pro Další Výzkum a Praktické Aplikace	64
II	PRAKTICKÁ ČÁST	68
3	METODIKA PRAKTICKÉ ČÁSTI	69
3.1	VÝBĚR APLIKACÍ PRO ANALÝZU	69
3.1.1	Aplikace A: bbbloat (CTF Úloha)	69
3.1.2	Aplikace B: Analyzovaný Shellcode	71
3.1.3	Aplikace C: GeometryMaster (Ukázková C++ Aplikace)	71
3.2	POUŽITÉ METODY A NÁSTROJE REVERZNÍHO INŽENÝRSTVÍ.....	72
3.2.1	Přehled Použitých Nástrojů	72
3.2.2	Aplikované Analytické Metody	73
3.2.3	Experimentální Využití Metod Umělé Inteligence	73
3.3	EXPERIMENTÁLNÍ USPOŘÁDÁNÍ A POSTUP ANALÝZY	74
3.3.1	Analytické Prostředí	74
3.3.2	Obecný Postup Analýzy pro Každou Aplikaci	75
3.4	KRITÉRIA HODNOCENÍ A METRIKY ÚSPĚŠNOSTI	75
3.5	ZAJIŠTĚNÍ REPRODUKOVATELNOSTI A ETICKÉ ASPEKTY	77
4	APLIKACE METOD, ANALÝZA VÝSLEDKŮ A VYHODNOCENÍ.....	78
4.1	ANALÝZA APLIKACE A: BBBLOAT	78
4.1.1	Prvotní Průzkum a Statická Analýza.....	78
4.1.2	Dynamická Analýza	79
4.1.3	Výsledky Experimentální AI Asistence	79
4.1.4	Vyhodnocení Analýzy Aplikace A	80
4.2	ANALÝZA APLIKACE B: ANALYZOVANÝ SHELLCODE	80
4.2.1	Prvotní Průzkum a Statická Analýza.....	80
4.2.2	Dynamická Analýza	81
4.2.3	Výsledky Experimentální AI Asistence	82
4.2.4	Vyhodnocení Analýzy Aplikace B	82
4.3	ANALÝZA APLIKACE C: GEOMETRYMASTER (UKÁZKOVÁ C++ APLIKACE) ..	82
4.3.1	Specifický Postup Analýzy, Použité Nástroje a Nastavení Prostředí	82
4.3.2	Výsledky Statické Analýzy a Vliv Optimalizací na Dekompilaci v Ghidře	83

4.3.3	Výsledky Dynamické Analýzy	84
4.3.4	Výsledky AI-Asistované Analýzy (Experimentální)	84
4.3.5	Vyhodnocení Analýzy Aplikace C	84
5	SROVNÁNÍ EFEKTIVITY METOD, NÁSTROJŮ A DISKUZE	86
5.1	SROVNÁVACÍ ANALÝZA POUŽITÝCH METOD	86
5.1.1	Statická vs. Dynamická Analýza a Jejich Synergie.....	86
5.1.2	Specifické Techniky pro C++.....	87
5.2	SROVNÁNÍ EFEKTIVITY A POUŽITELNOSTI NÁSTROJŮ	88
5.2.1	Hlavní Nástroje Statické Analýzy	88
5.2.2	Hlavní Nástroje Dynamické Analýzy.....	88
5.2.3	Hodnocení Experimentálního Využití AI/ML Metod.....	89
5.3	DISKUZE VÝSLEDKŮ A SPLNĚNÍ CÍLŮ	89
6	ZÁVĚRY, DOPORUČENÍ A NÁVRHY DALŠÍ PRÁCE	91
6.1	HLAVNÍ ZÁVĚRY VYPLÝVAJÍCÍ Z PRÁCE	91
6.2	VYPRACOVÁNÍ OBECNÝCH DOPORUČENÍ PRO REVERZNÍ INŽENÝRSTVÍ C/C++ APLIKACÍ	92
6.3	NÁVRHY PRO DALŠÍ PRÁCI A VÝZKUM.....	94
	ZÁVĚR.....	96
	SEZNAM POUŽITÉ LITERATURY	99
	SEZNAM TABULEK	104
	SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK.....	105
	SEZNAM PŘÍLOH.....	110

ÚVOD

Všudypřítomné softwarové systémy, klíčové pro infrastrukturu i osobní potřebu, vyžadují s rostoucí komplexitou pokročilé analytické schopnosti pro zajištění bezpečnosti a spolehlivosti. Mnohé z těchto systémů, zejména proprietární, starší či škodlivý software, fungují jako „černé skříňky“ bez dostupné dokumentace či zdrojového kódu. Reverzní inženýrství se tak stává kritickou nutností pro důvěru, bezpečnost a inovace, adresuje informační asymetrii a odhaluje skrytou realitu digitálních technologií.

Reverzní inženýrství (RE) softwaru, dle ISO/IEC/IEEE 24765 [1], je systematická, neinvazivní analýza systému pro identifikaci komponent, jejich interakcí a vytvoření abstraktní reprezentace. Cílem je porozumění, nikoli modifikace. Rekonstruuje sémantiku systému, odhaluje jeho účel, návrhová rozhodnutí a architekturu. Motivace dle Chikofského a Crosse [2] zahrnují porozumění systému, obnovu dokumentace, podporu údržby a evoluce, a kybernetickou bezpečnost (odhalování zranitelností, analýza malware). Dalšími jsou interoperabilita, analýza proprietárních formátů, ověřování shody a forenzní analýza. Problematika RE je však neodmyslitelně spjata i s komplexními etickými otázkami, které vyžadují pečlivé zvážení. Tyto otázky se týkají například hranice mezi legitimním zkoumáním za účelem interoperability, bezpečnosti či vzdělávání a neoprávněným zasahováním do duševního vlastnictví nebo obcházením ochranných mechanismů. Tato práce, ačkoliv se primárně soustředí na technické a analytické aspekty RE, si je těchto etických dilemat plně vědoma a klade důraz na zodpovědný a legální přístup k prováděným analýzám v celém svém rozsahu.

Analýza C/C++ aplikací přináší specifické výzvy kvůli výkonu a nízkoúrovňové kontrole, vedoucí ke komplexnímu binárnímu kódu. Problémy zahrnují agresivní optimalizace překladačů (GCC, Clang, MSVC) měnící strukturu kódu (inlinování, rozvinování smyček) a rozšiřující „sémantickou mezeru“. Absence metadat (bez *Program Database* (PDB), *Debugging With Attributed Record Formats* (DWARF)) vyžaduje heuristiky. Dále analýzu ztěžuje komplexita jazyka C++, zahrnující například šablony, komplexní mechanismy dědičnosti, přetěžování operátorů, virtuální funkce, podporu pro Run-Time Type Information (Informace o typu za běhu) (RTTI) či *Structured Exception Handling* (SEH), specifika „name mangling“, přímou práci s pamětí a možnost vkládání inline assembleru. Obfuskace a anti-reverzní techniky aktivně brání analýze. Dekompilace C/C++ je náročná; dekompilátor se snaží zvrátit proces kompilace s neúplnými daty, dosahujíc jen částečné sémantické ekvivalence, což

zdůrazňuje roli analytika.

Analytické techniky RE se vyvinuly. Tradiční statická (disasemblace, dekompilace) a dynamická analýza (ladění) zůstávají základem, jejich synergie je klíčová. Demokratizace nástrojů (např. Ghidra od *National Security Agency* (NSA) [3]) zpřístupnila pokročilé capacity. Moderní metody (viz Kapitola 2) zahrnují AI/Machine Learning (Strojové učení) (ML), zejména Large Language Model (Velký jazykový model) (LLM), pro zlepšení čitelnosti (LLM4Decompile [4]), obnovu sémantiky (VarBERT [5], ReSym [6]) a sumarizaci. Dalšími jsou *Model-Driven Reverse Engineering* (MDRE), fuzzing dekompilátorů (BIN2WRONG [7]) a symbolická exekuce. Směřuje se k „rozšířené inteligenci“, kde AI asistuje expertům. Budoucnost RE je v kombinaci znalostí analytika a pokročilých nástrojů.

Cílem práce je teoretický přehled RE softwaru (metodologie, historie) se zaměřením na C/C++. Kriticky hodnotí moderní metody a nástroje, včetně AI/ML. Stěžejní je praktické zkoumání C/C++ aplikací: analýza vlivu optimalizací, absence symbolů a C++ rysů (bez RTTI) na dekompilaci (GeometryMaster); techniky pro analýzu a deobfuskaci nízkoúrovňového kódu (shellcode); a standardní postupy RE (CTF úloha bbbbloat). Hodnotí se efektivita nástrojů (Ghidra, debuggery) a experimentálně AI asistence. Konečným cílem je syntéza poznatků do doporučení a identifikace směrů budoucího výzkumu.

Struktura práce: Kapitola 1 – teoretické základy RE, cíle, metodologie, výzvy C/C++. Kapitola 2 – moderní metody a nástroje RE, AI, pokročilé techniky. Kapitola 3 – metodika praktické části (výběr aplikací, nástroje, uspořádání, kritéria). Následující kapitoly prezentují výsledky, diskuzi a vedou k závěrům a doporučením.

Při zpracování této diplomové práce byly v roce 2025 jako podpůrné prostředky využity nástroje umělé inteligence (AI). Toto využití proběhlo po konzultaci a dohodě s vedoucím práce a v souladu s pravidly stanovenými garantem studijního programu. Konkrétně byly použity následující nástroje:

- ChatGPT vyvinutý společností OpenAI [8]
- Microsoft Copilot vyvinutý společností Microsoft [9]
- Gemini vyvinutý společností Google [10]

Uvedené nástroje byly využity například pro generování prvotních textových osnov, brainstorming klíčových konceptů, formulaci alternativních vysvětlení složitých témat a pomoc

při stylistických úpravách textu.

I TEORETICKÁ ČÁST

1 REVERZNÍ INŽENÝRSTVÍ

Tato kapitola se zabývá komplexní problematikou reverzního inženýrství softwaru. Nejprve budou vymezeny základní pojmy a definice dle relevantních standardů, následně budou detailněji rozebrány klíčové metodologie, cíle, historický kontext a význam RE v životním cyklu softwaru. Zvláštní pozornost bude věnována specifikům a výzvám spojeným s analýzou aplikací v jazycích C a C++.

Relevantní mezinárodní standardy (IEEE Std 1219-2021 [11], ISO/IEC/IEEE 24765 [1]) definují **reverzní inženýrství softwaru** (RE), označované rovněž termíny *Reverse Engineering*, případně méně frekventovaně *Backward engineering* či *Back engineering*, jako systematický proces. Jeho účelem je **identifikace komponent** existujícího softwarového systému, **deskripce jejich vzájemných interakcí** a následná **konstrukce abstraktní reprezentace** daného systému na vyšší úrovni abstrakce [1]. Norma ISO/IEC/IEEE 24765:2017[1] poskytuje následující formální definici:

“process of analyzing a subject system to identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction”[1]

Fundamentálním atributem RE je skutečnost, že analytický proces probíhá striktně **neinvazivním způsobem**, tj. bez jakékoli modifikace zkoumaného objektu. Primárním cílem je dosažení **komplexního porozumění logické struktury** a interním mechanismům fungování systému, nikoliv jeho úprava, kopírování či jiná forma zásahu. Například, pokud je nutné porozumět, jak starší, již nepodporovaná desktopová aplikace implementovaná v jazyce C++ ukládá data do proprietárního souborového formátu .dat bez dostupné dokumentace, lze aplikovat techniky RE. Prostřednictvím disassembleru a debuggeru by analytik zkoumal relevantní segmenty kódu zodpovědné za operace čtení a zápisu, s cílem odhalit vnitřní strukturu souboru. Získané poznatky by následně umožnily vývoj nástroje pro export dat či zajištění kompatibility s novějšími systémy.

1.1 Cíle a motivace Reverzního inženýrství

Úkolem RE je **rekonstrukce sémantiky** analyzovaného systému, tedy odhalení jeho podstaty, významu a původního zamýšleného účelu. Tento proces zahrnuje odkrývání původních

návrhových rozhodnutí, identifikaci implementovaných architektonických vzorů a celkové pochopení záměru tvůrců systému. Samotný kód, zejména v jeho binární formě (spustitelný soubor či knihovna), často vyjadřuje tyto sémantické aspekty pouze implicitně. Jejich odhalení a správná interpretace představují klíčovou a často nejnáročnější výzvu [12]. Analytik RE pracuje s různými úrovněmi abstrakce – od samotného **strojového kódu**, přes **jazyk symbolických adres** (*assembly*) a **dekompilovaný kód**, až po **vysokoúrovňové modely architektury a chování systému**. Snahou je typicky přechod od nízkoúrovňové reprezentace k vysokoúrovňovému porozumění. Proces disassemblování pak označuje konverzi strojového kódu do symbolické formy jazyka assembler.

Primární cíle a motivace pro aplikaci technik RE, jak je identifikují například Chikofsky a Cross [2], jsou rozmanité a zahrnují zejména následující aspekty:

Hluboké porozumění systému Detailní odhalení jeho architektury (např. jednotlivé vrstvy, moduly, komunikační rozhraní), identifikace klíčových komponent, použitých algoritmů (např. pro šifrování, kompresi, specifické výpočetní operace) a datových struktur (např. formáty souborů, databázová schémata, paměťové reprezentace). V kontextu C/C++ aplikací může jít například o detailní analýzu proprietárních binárních formátů souborů nebo síťových protokolů. Toto porozumění tvoří fundament pro veškeré následné informované aktivity.

Obnova dokumentace Generování nebo esenciální aktualizace technické dokumentace (návrhové diagramy, specifikace Application Programming Interface (Rozhraní pro programování aplikací) (API), popis datových modelů, apod) v případech, kdy původní dokumentace zcela absentuje, je nedostatečná, nekonzistentní, nebo ztratila aktuálnost v důsledku postupných modifikací systému.

Podpora údržby a evoluce Pochopení existujícího kódu je nezbytným předpokladem pro jeho efektivní a bezpečnou opravu chyb (*bug fixing*), modifikaci (implementace nových funkcí) a další rozvoj v souladu s měnícími se požadavky [13]. Bez RE by údržba často implikovala operace s vysokou mírou nejistoty a rizikem zavedení nových chyb.

Identifikace znovupoužitelných komponent Detekce a extrakce kvalitně navržených a implementovaných částí systému (např. knihovny, moduly, algoritmy), které mohou být

s minimálními úpravami integrovány do jiných softwarových projektů, čímž dochází k úspoře času a zdrojů a podpoře konzistence.

Analýza bezpečnosti Odhalování zranitelností (např. buffer overflows, integer overflows, use-after-free, injection flaws, logické chyby), bezpečnostních slabin v návrhu či implementaci a detailní analýza škodlivého softwaru (malware) za účelem porozumění jeho chování, mechanismů šíření, perzistence a vývoje efektivních protipatření. U C/C++ aplikací je toto typicky zaměřeno na hledání zranitelností v nativním kódu (např. *memory corruption*) [14].

Zajištění interoperability Umožnění komunikace a efektivní kooperace mezi různými, často heterogenními softwarovými systémy, které původně nebyly navrženy pro vzájemnou interakci. Tento proces může zahrnovat porozumění proprietárním API, datovým formátům nebo komunikačním protokolům, například pro zajištění interoperability starších C++ systémů s novými platformami nebo službami.

Analýza proprietárních formátů a protokolů Porozumění struktuře neveřejných datových formátů souborů nebo síťových komunikačních protokolů, například pro účely vývoje kompatibilních nástrojů, pro import/export dat, nebo pro analýzu chráněného obsahu.

Ověření shody a forenzní analýza Kontrola, zda systém splňuje stanovené průmyslové normy, legislativní požadavky, licenční podmínky (např. ověření použití *open-source* komponent a jejich licencí), nebo pro účely digitální forenzní analýzy při vyšetřování bezpečnostních incidentů, porušení duševního vlastnictví či jiných nelegálních aktivit [15].

1.2 Terminologie a související pojmy oboru

Pro účely jednoznačné terminologické orientace jsou vymezeny relevantní a související pojmy. Tabulka 1.1 poskytuje přehledné srovnání těchto termínů a jejich klíčové odlišnosti, či příbuznosti, od samotného reverzního inženýrství.

Tato práce se primárně zaměřuje na problematiku RE desktopových aplikací implementovaných v jazycích C a C++, což zahrnuje jak aspekty bezpečnostní analýzy (např. detekce zranitelností, analýza škodlivého softwaru), tak i technické dimenze provádění reverzní analýzy těchto nativních programů.

Tabulka 1.1 Srovnání klíčových pojmů souvisejících s Reverzním inženýrstvím

Pojem	Stručná definice	Klíčový rozdíl oproti RE
Backward Engineering	Starší, dnes méně frekventovaný, ale v podstatě synonymní termín pro RE [1].	Prakticky žádný, jedná se o synonymum.
Forward Engineering	Tradiční vývojový proces od specifikace požadavků, přes návrh, k implementaci a testování [2].	Opačný směr procesu; RE analyzuje již existující produkt.
Re-engineering (Reinženýrství)	Proces zkoumání a modifikace existujícího systému za účelem jeho rekonstrukce a reimplementace v nové formě, často s cílem zlepšení kvality, výkonu nebo údržby. RE je často první fází reinženýrství [12].	RE je primárně analytická fáze (porozumění), zatímco re-engineering zahrnuje i následnou modifikaci a implementaci změn.
Redokumentace (Redocumentation)	Tvorba nebo doplnění dokumentace k existujícímu systému na stejné úrovni abstrakce, bez jeho modifikace. Výstupem je nová reprezentace systému [2].	Specifický cíl RE; zaměřuje se na generování popisů a reprezentací na stejné úrovni abstrakce jako analyzovaný systém.
Obnova návrhu (Design Recovery)	Rekonstrukce vysokoúrovňových návrhových rozhodnutí, architektury a obchodní logiky systému. Využívá informace získané z kódu, dokumentace (pokud existuje) a obecných znalostí o dané doméně [2].	Specifický cíl RE; směřuje k vytvoření reprezentací na vyšších úrovních abstrakce než analyzovaný systém, s využitím i externích znalostí.
Reverse Code Engineering (RCE)	Často chápáno jako podmnožina RE specificky zaměřená na detailní analýzu strojového nebo bajtového kódu, typicky s cílem pochopit nízkoúrovňové detaily, identifikovat zranitelnosti, analyzovat malware nebo obejít ochrany.	Specifické zaměření RE na kódovou úroveň, často s bezpečnostním kontextem.

(zdroj: vlastní, inspirováno [2] a [12])

1.3 Metodologie Reverzní Inženýrské Praxe

Po vyjasnění terminologických východisek se nyní zaměříme na esenciální metodologie reverzní inženýrské praxe. Reverzní inženýrská praxe, jakožto disciplína zaměřená na dekonstrukci a pochopení existujících systémů, fundamentálně založena na systematické aplikaci dvou primárních a komplementárních analytických metodik: **statické analýzy** a **dynamické analýzy**. Volba a kombinace těchto přístupů je determinována povahou analyzovaného ob-

jektu a specifickými cíli reverzního inženýrství.

1.3.1 Statická analýza

Statická analýza představuje soubor technik, které zkoumají software a jeho komponenty bez nutnosti jejich faktického spuštění. Tento přístup se typicky uplatňuje při analýze binárních souborů, jako jsou spustitelné soubory, knihovny či bajtkód, zejména v situacích, kdy zdrojový kód není k dispozici. Mezi klíčové techniky statické analýzy patří:

Disasemblace: Proces převodu strojového kódu na jazyk symbolických adres (*assembler*), umožňující detailní vhled do nízkourovňové logiky programu a identifikaci základních stavebních bloků kódu.

Dekompilace: Pokus o rekonstrukci kódu ve vyšším programovacím jazyce z jeho binární nebo assemblerové reprezentace [16], což usnadňuje pochopení komplexnějších algoritmů a datových struktur, i když výsledek nemusí být vždy perfektní či plně ekvivalentní původnímu zdrojovému kódu.

Analýza řídicího toku: Identifikace a vizualizace možných cest provádění programu, odhalující jeho strukturu, smyčky, podmíněné větvení a funkce, což napomáhá pochopení celkové logiky a organizace kódu.

Analýza datového toku: Sledování, jak jsou data v programu definována, používána, modifikována a propagována mezi různými částmi kódu, což pomáhá porozumět manipulaci s informacemi a závislostem mezi daty.

Identifikace datových struktur: Rozpoznávání a interpretace způsobu, jakým jsou data organizována a uložena v paměti (např. objekty, pole, struktury), což je klíčové pro pochopení logiky zpracování dat a interakcí mezi komponentami.

1.3.2 Analýza Dynamická

Dynamická analýza se zaměřuje na sledování a interpretaci chování softwaru během jeho exekuce v kontrolovaném prostředí. Tento přístup umožňuje pozorovat reálné interakce programu s operačním systémem, hardwarem, síťovými službami a jinými procesy. Využívá široké spektrum technik, mezi něž patří:

Ladění (*Debugging*): Interaktivní proces spouštění programu pod kontrolou debuggeru, umožňující zastavování na určených místech (breakpoints), inspekci a modifikaci stavu paměti a registrů, což je esenciální pro identifikaci chyb, sledování konkrétních operací či porozumění specifickým vzorcům chování.

Krokování kódu: Detailní procházení programu instrukci po instrukci nebo řádek po řádku (pokud jsou dostupné ladicí symboly), poskytující granulární pohled na exekuční tok a změny stavu.

Sledování systémových volání: Monitorování interakcí programu s jádrem operačního systému (např. přístup k souborům, síťová komunikace, správa procesů a vláken), které odhaluje jeho externí závislosti a operace na úrovni systému.

Monitorování paměti a sítě: Sledování alokace a uvolňování paměti pro detekci úniků paměti, poškození paměti nebo neoprávněného přístupu, a analýza síťové komunikace pro odhalení přenášených dat, použitých protokolů a komunikačních partnerů.

Hooking (zachytávání): Technika umožňující interceptovat a potenciálně modifikovat volání funkcí, systémových volání nebo zpráv v rámci programu či mezi procesy, často využívaná pro monitorování specifických událostí nebo změnu chování programu za běhu.

Dynamická binární instrumentace (DBI): Vkládání analytického kódu (instrumentace) do běžícího programu za účelem sběru detailních informací o jeho chování (např. profilování výkonu, sledování provádění kódu, analýza pokrytí kódu) bez nutnosti modifikace původního binárního souboru na disku.

Společná aplikace statické a dynamické analýzy, kdy poznatky z jedné fáze informují a zpřesňují druhou, poskytuje nejkomplexnější a nejhlubší porozumění analyzovanému systému [14]. Například, statická analýza může identifikovat potenciálně zajímavé části kódu nebo datové struktury, které jsou následně podrobeny detailnímu zkoumání a ověření pomocí dynamických technik. Základními nástroji pro tyto metodologie jsou **disassemblery**, **dekompilátory** a **debuggery**, jejichž detailní popis a praktické využití bude předmětem diskuze v Kapitole 6.

1.4 Význam Reverzního inženýrství v životním cyklu softwaru

Význam RE v kontextu celého životního cyklu softwaru je nepopiratelný a mnohostranný. Představuje fundamentální prvek pro procesy **re-inženýrství** [17], které směřují k transformaci systému do nové, zdokonalené podoby (např. migrace na novou platformu, optimalizace výkonu, refaktoring architektury) [2]. Stejně tak je klíčové pro **modernizaci zastaralých systémů** (tzv. *legacy systems*) [17], kde poskytuje nezbytný vhled do jejich často komplexního a nedostatečně zdokumentovaného fungování [18]. V doméně kybernetické bezpečnosti je RE esenciálním nástrojem pro analytiku malware, výzkumníky zranitelností a etické hackery. S ohledem na skutečnost, že náklady na údržbu softwaru mohou mnohonásobně převýšit náklady na jeho původní vývoj, a samotné porozumění systému tvoří signifikantní část těchto nákladů na údržbu [13], stává se RE klíčovým instrumentem pro prodloužení životnosti, udržení hodnoty a snížení celkových nákladů na vlastnictví (Total Cost of Ownership (Celkové náklady na vlastnictví) (TCO)).

1.5 Historický vývoj a klíčové momenty

Historie RE softwaru je úzce spjata s evolucí výpočetní techniky a jejího právního rámce. Pro kontext této práce jsou relevantní zejména následující milníky:

- **Rané komerční využití a interoperabilita (80. léta):** Analýza Basic Input/Output System (BIOS)u IBM PC společností Phoenix Technologies pro tvorbu kompatibilních systémů ukázala potenciál RE pro podporu konkurence a inovací, což je relevantní i pro analýzu proprietárních formátů v C/C++ aplikacích [19].
- **Právní aspekty a duševní vlastnictví (90. léta):** Soudní případy jako *Sega v. Accolade* [20] v USA pomohly definovat legální hranice RE pro účely interoperability (tzv. *fair use*), což ovlivnilo přístup k analýze kódu chráněného autorským právem. Prolomení ochrany Content Scramble System (CSS) na Digital Versatile Disc (Digitální víceúčelový disk) nebo Digital Video Disc (Digitální videodisk) (DVD) programem DeCSS [21] vyvolalo rozsáhlé debaty o rovnováze mezi ochranou práv tvůrců a právem na RE pro legitimní účely.
- **Význam v kybernetické bezpečnosti (od 2000):** Analýza sofistikovaného malware (např. Stuxnet, Flame [22]) potvrdila nezastupitelnou roli RE při odhalování kom-

plexních kybernetických hrozeb, včetně těch cílících na C/C++ aplikace a kritickou infrastrukturu.

- **Demokratizace nástrojů (od 2010):** Uvolnění výkonných nástrojů jako Ghidra (vyvinutého NSA) jako *open-source* v roce 2019 [3] výrazně zpřístupnilo pokročilé analytické kapacity, včetně těch pro RE komplexních C/C++ aplikací, širší odborné veřejnosti, výzkumníkům i studentům.

Tento stručný přehled ilustruje, jak se RE, včetně analýzy C/C++ kódu, stalo klíčovou disciplínou pro interoperabilitu, bezpečnost, inovace a právní regulaci v dynamicky se vyvíjejícím softwarovém světě.

1.6 Softwarové inženýrství jako základ reverzní analýzy

Úspěšnost, efektivita a hloubka porozumění dosažitelná prostřednictvím RE jsou fundamentálně provázány s kvalitou a principy uplatněnými během původního vývojového procesu softwaru. Jak výstižně konstatuje Telea [23], efektivita a úspěšnost RE jsou nepřímo úměrné kvalitě původních softwarově inženýrských praktik. Jinými slovy, čím rigorózněji softwaroví inženýři navrhli a implementovali software, tím snazší a přesnější bude jeho následná reverzní analýza. Naopak, zanedbání softwarově inženýrských principů vede k systémům, jejichž analýza je extrémně časově náročná, nákladná a náchylná k chybné interpretaci [14]. Příkladem může být C++ aplikace s rozsáhlým použitím makr preprocesoru a komplexní, nedostatečně zdokumentovanou hierarchií tříd bez zjevného použití návrhových vzorů, kde identifikace skutečné logiky a datových toků představuje pro analytika RE značnou překážku. Klíčové principy softwarového inženýrství, definované například v Software Engineering Body of Knowledge (SWEBOK) [18], mají přímý dopad na analyzovatelnost kódu. Jejich absence či nedodržování proces RE významně komplikuje:

Modularita Systémy rozdělené na dobře definované, logicky oddělené moduly s jasnými rozhraními umožňují izolovanou analýzu jednotlivých částí, čímž snižují celkovou komplexitu RE. Absence modularity (monolitický kód, tzv. „*spaghetti code*“) naopak výrazně zvyšuje náročnost pochopení systému jako celku [24].

Abstrakce Kvalitní abstrakce ve zdrojovém kódu, která skrývá nepodstatné detaily implementace a zdůrazňuje klíčové koncepty, usnadňuje analytikovi RE identifikovat vy-

sokoúrovňové záměry a funkce. Nedostatečná abstrakce vede k nutnosti analyzovat nízkoúrovňové detaily bez jasného kontextu [25].

Zapouzdření Správné zapouzdření, které chrání vnitřní stav komponent a vystavuje pouze dobře definovaná rozhraní, pomáhá při RE soustředit se na interakce komponent, aniž by bylo nutné rozumět každému internímu detailu. Porušení zapouzdření vede k nepřehledným závislostem a ztěžuje izolovanou analýzu.

Moduly s vysokou soudržností (jejichž prvky logicky patří k sobě a spolupracují na dosažení jednoho cíle) a **nízkým propojením** (s minimálními a dobře definovanými závislostmi na jiných modulech) jsou pro RE snáze pochopitelné. Nízká soudržnost a vysoké propojení (tzv. „*big ball of mud*“) činí analýzu extrémně obtížnou.

Návrhové vzory Použití známých návrhových vzorů [26] v kódu poskytuje analytikovi RE vodítka pro rychlé pochopení záměru, struktury a interakcí mezi komponentami [12]. Absence vzorů nebo jejich nesprávné či nekonzistentní použití zvyšuje nejednoznačnost a časovou náročnost analýzy.

Celý **životní cyklus vývoje softwaru** (Software Development Lifecycle (Životní cyklus vývoje softwaru) (SDLC)) generuje rozličné artefakty – nejen zdrojový kód a spustitelné soubory, ale rovněž modely (např. *Unified Modeling Language* (UML) diagramy), projektovou a uživatelskou dokumentaci, testovací plány a sady testů, konfigurační skripty, provozní logy atd. [25]. Kvalita, úplnost a dostupnost těchto výstupů, stejně jako dodržování inženýrských principů během jejich tvorby (např. konzistentní konvence pojmenování, dodržování kódovacích standardů, minimalizace cyklomatické složitosti [27]), mohou buď významně usnadnit, nebo naopak extrémně zkomplikovat proces reverzní analýzy. RE tudíž nestojí mimo rámec softwarového inženýrství, nýbrž je jeho přirozenou a často nezbytnou součástí, zejména v kontextech údržby, modernizace, zajištění interoperability a bezpečnostní analýzy systémů [2]. Samotný proces **porozumění programu** (*program comprehension*), který tvoří jádro RE, se opírá o identické kognitivní strategie a modely (např. přístup *top-down*, *bottom-up*, oportunistický přístup), jaké využívají vývojáři při tvorbě a údržbě softwaru [28].

1.7 Specifika a výzvy Reverzního inženýrství C/C++ aplikací

Ačkoliv obecné principy a cíle RE platí napříč různými programovacími jazyky a typy softwaru, analýza aplikací implementovaných v jazycích C a C++ s sebou nese řadu specifických výzev. Tyto jazyky charakterizuje jejich výkon a flexibilita, které jsou však často vykoupeny vyšší komplexitou a nižší úrovní abstrakce ve výsledném binárním kódu.

1.7.1 Optimalizace překladačem

Moderní C/C++ překladače (např. GCC/G++, Clang, MSVC) aplikují **agresivní optimalizace** s cílem maximalizovat výkon a minimalizovat velikost kódu. Tyto techniky, jako je **inlining** funkcí, **rozvinování smyček**, **eliminace mrtvého kódu**, **alokace proměnných do procesorových registrů** či **změna pořadí instrukcí**, mohou fundamentálně alterovat strukturu kódu oproti původnímu zdrojovému textu a signifikantně znesnadnit identifikaci původních programových konstrukcí a logiky [29, 14]. Analytici se s tímto vyrovnávají pomocí pokročilých technik analýzy řídicího a datového toku, heuristik pro rozpoznávání optimalizačních vzorů, pečlivým manuálním trasováním kódu a v poslední době i s využitím velkých jazykových modelů pro zlepšení srozumitelnosti dekompilovaného kódu [16].

1.7.2 Absence metadat a reflexe

Kompilovaný C/C++ kód, na rozdíl od jazyků jako Java či C#, typicky **neobsahuje bohatá metadata** (informace o třídách, metodách, typech, jménech proměnných) ani **nepodporuje mechanismy reflexe**, pokud programátoři explicitně nezahrnou ladicí symboly (např. ve formátu PDB na Windows nebo DWARF na systémech používajících Executable and Linkable Format (Spustitelný a linkovatelný formát) (ELF)) [12]. Bez těchto symbolů je rekonstrukce vysokoúrovňových struktur, jako jsou názvy funkcí či typy proměnných, podstatně náročnější a často se musí opírat o heuristické přístupy a manuální analýzu [14]. Rekonstrukce těchto informací se často opírá o signatury knihovních funkcí, analýzu typového systému na základě použití dat, inferenci z kontextu a moderní přístupy využívající strojové učení pro obnovu sémantických informací, jako je například odvozování názvů funkcí [30].

1.7.3 Komplexita jazyka C++

Samotný jazyk C++ nabízí řadu **pokročilých rysů**, jejichž reprezentace v binárním kódu je komplikovaná a ztěžuje RE [14]. Pro zvládnutí těchto rysů je nutné kombinovat hluboké znalosti jazyka s detailní analýzou paměťových layoutů a vzorů generovaného kódu. Mezi tyto rysy patří:

Šablony Mohou vést k rozsáhlému objemu specializovaného kódu (*code bloat*) v důsledku **monomorfizace**, což znesnadňuje identifikaci původní generické šablony a její logiky [31]. Analytici musí často manuálně identifikovat vzory kódu vzniklé instanciací šablon.

Vícenásobná dědičnost a virtuální dědičnost Uspořádání bazových tříd v paměti a mechanismy volání virtuálních funkcí mohou být značně komplikované a závislé na konkrétním překladači a Aplikační binární rozhraní (ABI) [32]. Pochopení vyžaduje analýzu ukazatelů na bazové třídy a virtuálních tabulek.

Přetěžování operátorů V assembleru se jeví jako standardní volání funkcí (často s manglovanými jmény), což může zastírat jejich sémantiku a znesnadnit pochopení jejich účelu bez znalosti kontextu třídy [12]. Identifikace skutečné operace vyžaduje sledování typů argumentů a kontextu volání.

Virtuální funkce a polymorfismus Implementace prostřednictvím **tabulek virtuálních metod** (vtables) vyžaduje jejich identifikaci, rekonstrukci a analýzu ukazatelů na funkce, což je klíčové pro pochopení dynamického chování objektů [12]. Klíčová je zde pečlivá analýza volání přes ukazatele a rekonstrukce obsahu vtable.

Mechanismy zpracování výjimek RTTI, SEH (na Windows) nebo standardní C++ výjimky (často implementované pomocí Itanium ABI EH [33] nebo specifických mechanismů překladače) využívají komplexní datové struktury a tabulky, jejichž interpretace je náročná a vyžaduje detailní znalost ABI [12]. Analytici se zaměřují na identifikaci tabulek pro odvíjení zásobníku (*stack unwinding tables*) a handlerů výjimek.

Analýza těchto konstrukcí vyžaduje hluboké znalosti C++, aplikačního binárního rozhraní (ABI) a chování konkrétního překladače.

1.7.4 Name mangling

Překladače C++ používají různé konvence pro **name mangling** (dekoraci jmen) k zakódování typových informací (např. typy parametrů, jmenné prostory, třídy) do symbolických jmen funkcí a proměnných. Toto je nezbytné pro podporu přetěžování funkcí, šablon a jmenných prostorů. Pro analytika RE to znamená nutnost "**demanglovat**" (oddekorovat) symboly pomocí specializovaných nástrojů (např. `c++filt` pro Itanium ABI, `undname` pro MSVC) nebo znalosti konkrétního mangling schématu (např. Itanium C++ ABI mangling, Microsoft MSVC mangling [33]). Použití specializovaných nástrojů pro demangling je zde standardním a nezbytným postupem, přičemž moderní techniky se snaží automatizovat i obnovu původních jmen v případech, kdy jsou symboly zcela odstraněny [30].

1.7.5 Přímá práce s pamětí a ukazateli

Přímá manipulace s pamětí a manuální správa paměti (alokace a uvolňování) v C/C++ mohou vést k obtížně sledovatelným datovým tokům, komplexním závislostem a chybám jako jsou úniky paměti, použití paměti po uvolnění nebo přetečení bufferu. Analýza těchto aspektů v binárním kódu je velmi náročná a vyžaduje pečlivé sledování ukazatelů a alokací [34, 14]. Detailní sledování alokací, uvolňování a používání ukazatelů, často s podporou dynamické analýzy a specializovaných nástrojů pro detekci paměťových chyb, je zde nezbytné, přičemž porozumění strukturám v paměti je klíčové i pro forenzní analýzu [15].

1.7.6 Použití inline assembleru a nízkoúrovňových konstrukcí

Možnost vkládat **inline assembler** přímo do C/C++ kódu nebo používat jiné **nízkoúrovňové techniky** (např. přímý přístup k hardwarovým portům, specifické instrukce procesoru) dále komplikuje automatizovanou analýzu a dekompilaci, jelikož dekompilátory nemusí být schopny tyto konstrukce správně interpretovat nebo převést na ekvivalentní vysokoúrovňový kód [12]. Tyto sekce často vyžadují izolovanou manuální analýzu a pochopení nízkoúrovňových operací přímo v jazyce *assembler*.

1.7.7 Obfuskace a anti-reverzní techniky

Vývojáři, zejména v oblastech ochrany softwaru (*Digital Rights Management (DRM)*), licencování nebo v případě škodlivého softwaru (*malware*), cíleně aplikují techniky **obfuskace** a

anti-reverzní techniky, aby aktivně bránili analýze, znesnadnili porozumění kódu a zvýšili náklady na jeho prolomení [35, 36]. Cílem těchto technik není učinit software absolutně neprolomitelným, ale spíše analýzu natolik zkomplikovat a časově prodloužit, aby se stala neekonomickou nebo aby poskytla dostatek času na reakci (např. v případě malware). Tyto techniky zahrnují širokou škálu metod, které lze kategorizovat následovně:

- **Šifrování nebo komprese kódu či jeho částí (packery, protektory):** Jedná se o jednu z nejběžnějších forem ochrany. Většina původního kódu aplikace (nebo její klíčové části) je zašifrována nebo zkomprimována. Při spuštění se nejprve vykoná malá dešifrovací/dekompresní rutina, často označovaná jako **stub** nebo **loader**. Tento stub obnoví původní kód do paměti (často do nově alokované oblasti) a následně mu předá řízení.
 - *Příklady nástrojů:* UPX (populární open-source kompresor), ASPack, a komerční protektory jako Themida, VMProtect, Enigma Protector, které často kombinují packing s dalšími obfuskačními a anti-reverzními technikami.
 - *Výzvy pro RE:* Nutnost identifikovat originální vstupní bod (*Original Entry Point* (OEP) - *Original Entry Point*) po dekompresi/dešifrování, úspěšně dumpovat (uložit) dešifrovaný kód z paměti a případně opravit tabulku importů (Import Address Table (Importní tabulka adres) (IAT) - *Import Address Table*), kterou packer mohl modifikovat.
 - *Techniky proti:* Manuální *unpacking* (krokování stubu v debuggeru, hledání OEP), použití generických *unpackerů* (např. pomocí sledování paměťových operací a heuristik), nebo specializovaných skriptů pro konkrétní packery.
- **Vkládání zavádějících nebo nadbytečných instrukcí (*junk code*, *opaque predicates*):** Cílem je ztížit manuální čtení kódu a zmást automatické nástroje pro analýzu.
 - ***Junk code*** (balastní kód): Do původního kódu jsou vloženy sekvence instrukcí, které nemají žádný vliv na skutečnou logiku programu. Může jít o jednoduché instrukce jako NOP (*No Operation*), MOV EAX, EAX, nebo složitější, ale sémanticky irelevantní aritmetické operace, jejichž výsledek se dále nepoužívá.
 - ***Opaque predicates*** (neprůhledné predikáty): Jedná se o podmíněné skoky, jejichž podmínka je navržena tak, aby se vždy (nebo naopak nikdy) vyhodnotila

jako pravdivá, ale pro analytika (nebo statický analyzátor) to není na první pohled zřejmé. Například podmínka ‘if ($x * x < 0$)’ pro celočíselné ‘x’ je vždy nepravdivá. Sofistikovanější neprůhledné predikáty mohou záviset na komplexních výpočtech nebo zdánlivě na externím stavu, ale ve skutečnosti vedou vždy ke stejnému výsledku. Cílem je vytvořit falešné větve v grafu řídicího toku a zkomplikovat jeho analýzu.

- *Výzvy pro RE*: Nutnost identifikovat a mentálně (nebo pomocí nástrojů) odfiltrovat balastní kód, analyzovat a zjednodušit neprůhledné predikáty k odhalení skutečného toku řízení.
- *Techniky proti*: Symbolická exekuce, analýza datového toku (pro sledování, zda se výsledky operací používají), *pattern matching* pro známé vzory balastního kódu.
- **Zamlžování řídicího toku (*control flow obfuscation*)**: Tyto techniky mění strukturu programu tak, aby bylo obtížné sledovat jeho logický tok.
 - ***Control flow flattening*** (zploštění řídicího toku): Původní strukturovaný řídicí tok (smyčky, podmíněné větve) je transformován. Typicky je vytvořena hlavní smyčka s velkým příkazem ‘switch’ (nebo sérií vnořených ‘if-else if’ struktur). Každý původní základní blok kódu se stane jednou ‘case’ větví (nebo blokem v ‘if’). Speciální stavová proměnná určuje, která ‘case’ větev se provede jako další, čímž se simuluje původní tok.
 - **Nepřímé skoky a volání**: Použití instrukcí skoku nebo volání, jejichž cíl není explicitně uveden v kódu, ale je určen hodnotou v registru nebo paměťové lokaci, která se dynamicky vypočítává. To výrazně ztěžuje statickou analýzu grafu řídicího toku.
 - **Vkládání falešných podmíněných skoků** nebo přerušení sekvenčního provádění kódu pomocí bezpodmínečných skoků na následující instrukci.
 - *Výzvy pro RE*: Rekonstrukce původního, srozumitelného grafu řídicího toku je velmi obtížná a časově náročná.
 - *Techniky proti*: Pokročilá statická analýza (např. s využitím symbolické nebo konkrétní exekuce k určení cílů nepřímých skoků), dynamická analýza k traso-

vání skutečně provedených cest, specializované deobfuskační nástroje a skripty (např. pro de-flattening).

- **Anti-debugging a anti-disassembly triky:** Cílem je detekovat přítomnost analytických nástrojů (debuggerů, disassemblerů) nebo znesnadnit jejich správnou funkci.

– **Detekce debuggeru:**

- * Použití specifických API funkcí operačního systému (např. 'IsDebuggerPresent()', 'CheckRemoteDebuggerPresent()' na Windows, kontrola ptrace na Linuxu).
- * Časovací kontroly: Měření času provádění určitých úseků kódu; pod debuggerem běží program typicky pomaleji.
- * Detekce softwarových ('INT 3' - instrukce '0xCC') nebo hardwarových breakpointů.
- * Využití specifických výjimek, které se chovají jinak, pokud je proces laděn (např. EXCEPTION_BREAKPOINT).
- * Kontrola integrity kódu (zda nebyl modifikován debuggerem).

– **Anti-disassembly** (matení disassemblerů):

- * Vkládání dat (např. konstanty, řetězce) přímo do proudu instrukcí, které může disassembler chybně interpretovat jako kód.
- * Použití skoků doprostřed vícebajtových instrukcí, což způsobí, že disassembler začne analyzovat od chybného místa.
- * Použití instrukcí, které mají více možných interpretací délky v závislosti na prefixech.
- * Samo-modifikující se kód: Část kódu se za běhu mění (např. dešifruje další část kódu nebo modifikuje existující instrukce), což statický disassembler neodhalí.

– *Výzvy pro RE:* Nutnost obejít detekční mechanismy (např. patchováním binárky, použitím pluginů do debuggeru, které skrývají jeho přítomnost) a manuálně opravit chyby či nepřesnosti v disassembleru.

– *Techniky proti:* Použití pokročilých debuggerů s *anti-anti-debugging* funkcemi (např. ScyllaHide), manuální oprava kódu nebo dat v disassembleru, použití více

různých disassemblerů pro srovnání výstupů, dynamická analýza pro odhalení samo-modifikujícího se kódu.

- **Virtualizace kódu (*Code Virtualization*):** Jedna z nejsložitějších technik obfuskace. Původní instrukce programu (nebo jeho kritické části) jsou přeloženy do vlastního, často unikátního bajtkódu. Tento bajtkód je pak interpretován vestavěnou **virtuální mašinou** (Virtual Machine (Virtuální stroj) (VM)), která je součástí chráněné aplikace.
 - *Charakteristika VM:* Obvykle má vlastní, specifickou sadu instrukcí (tzv. *opcodes*), vlastní virtuální registry, vlastní zásobník a mechanismus pro zpracování těchto instrukcí (tzv. *handlers*). Může být *stack-based* (operace probíhají primárně na zásobníku) nebo *register-based*.
 - *Příklady nástrojů využívajících virtualizaci:* VMProtect, CodeVirtualizer, Themida (často obsahují virtualizační komponentu jako jednu z vrstev ochrany).
 - *Výzvy pro RE:* Extrémně náročné a časově velmi nákladné. Vyžaduje nejprve reverzní inženýrství samotné VM – pochopení její architektury, sady instrukcí (jejich formátu a sémantiky), způsobu, jakým VM přistupuje k paměti a registrům reálného procesoru, a jak jsou implementovány jednotlivé handlersy bajtkódových instrukcí. Teprve poté může analytik přistoupit k analýze samotného virtualizovaného bajtkódu.
 - *Techniky proti:* Detailní manuální analýza jednotlivých handlerů VM (často nejprve staticky, poté dynamicky), vývoj vlastních de-virtualizátorů nebo převodníků bajtkódu na nativní kód (tzv. *devirtualizers* nebo *lifters*), použití symbolické exekuce na úrovni VM pro pochopení logiky virtualizovaného kódu.

Překonání těchto ochran často vyžaduje aplikaci specifických deobfuskáčnických technik, vývoj vlastních skriptů či nástrojů pro automatizaci analýzy (např. s využitím IDA Python, Ghidra Scripting, Frida), a vždy hluboké specializované znalosti a značné časové investice. Kombinace statické a dynamické analýzy je zde naprosto klíčová, stejně jako schopnost rozpoznávat vzory (pattern matching) známých obfuskáčnických technik a využití heuristik. I přes pokročilé nástroje zůstává manuální analýza a "lidská intuice" nezbytnou součástí procesu.

Kombinace těchto faktorů činí reverzní inženýrství C/C++ aplikací vysoce specializovanou a intelektuálně náročnou disciplínou, vyžadující hluboké znalosti nejen samotných jazyků a jejich standardních knihoven, ale i operačních systémů, hardwarové architektury, formátů spustitelných souborů, aplikačních binárních rozhraní (ABI) a pokročilých analytických nástrojů a technik. Úspěšné zvládnutí těchto komplexních výzev je však klíčové pro dosažení cílů reverzního inženýrství popsanych v této kapitole, ať už se jedná o zajištění interoperability, analýzu bezpečnosti či obnovu ztracené dokumentace. Problematika RE tak zůstává dynamickou a esenciální oblastí softwarového inženýrství.

2 MODERNÍ METODY A NÁSTROJE REVERZNÍ ANALÝZY C/C++ APLIKACÍ

Zde se navazuje na předchozí výklad teoretických základů, metodologií a výzev reverzního inženýrství, zejména v kontextu C/C++ aplikací, jak byly podrobně rozebrány v kapitole první. Zaměříme na detailnější rozbor pokročilých a moderních metod RE, analýzu dostupných nástrojů ¹⁾ a jejich efektivitu při rekonstrukci kódu podobného zdrojovému, a charakteristiku typických aplikací vybraných pro praktické ověření těchto přístupů. Základní porozumění jazyku symbolických adres (*assembly*) je pro RE klíčové [37, 38].

2.1 Přehled Moderních Metod Reverzního Inženýrství

Zatímco tradiční přístupy k RE, jako je disasemblování, analýza řídicího a datového toku (podrobně diskutované v předchozí kapitole a v [39, 40]), tvoří základ, vývoj v oblasti AI a specializovaných analytických technik přináší nové možnosti.

2.1.1 Metody Založené na Umělé Inteligenci a Strojovém Učení (AI/ML)

AI a ML, zejména velké jazykové modely (LLM), nabízejí potenciál pro řešení některých problémů tradiční dekompilace, především v oblasti čitelnosti a obnovy sémantiky [41]. Následující přehled představuje klíčové směry výzkumu a aplikace AI/ML v kontextu reverzního inženýrství.

Modely Založené na LLM pro Dekompilaci: Tyto modely představují špičku výzkumu v automatizované dekompilaci a vylepšování jejich výsledků.

- **LLM4Decompile:** Série LLM modelů (s počtem parametrů od 1.3 miliardy do 33 miliard) specificky navržených pro zlepšení procesu dekompilace binárního kódu. Autoři tvrdí, že tyto modely mohou překonávat tradiční nástroje v čitelnosti a spustitelnosti generovaného kódu [4].
- **Refined-Decompile:** Tento přístup využívá LLM k vylepšení („zušlechtnění“) pseudokódu generovaného tradičními dekompilátory (např. z Ghidra nebo IDA Pro). Cílem je zlepšit čitelnost, obnovit idiomatické konstrukce jazyka C/C++ a případně opravit některé sémantické nepřesnosti [4].

¹⁾literatura, nástroje, kurzy a více je k nalezení na: <https://github.com/wtsxDev/reverse-engineering?tab=readme-ov-file>

- **End2end-Decompile:** Ambiciózní přístup, kde LLM provádí dekompilaci přímo z binárního kódu na vysokoúrovňový kód bez mezikroku tradičního dekompilátoru. Tento směr je stále předmětem aktivního výzkumu [4].

Techniky pro Obnovu Sémantických Informací: AI/ML se uplatňují při obnově klíčových sémantických informací, které jsou často ztraceny během procesu kompilace, čímž výrazně napomáhají pochopení dekompilovaného kódu.

- **Obnova Názvů Funkcí:** Metody jako NERO (využívající rozšířené Control Flow Graph (Graf řídicího toku) (CFG) a neuronové modely), NFRE (analyzující sekvenční a strukturální informace instrukcí), SymLM (zahrnující kontext volání) a NER (zaměřený na různé reprezentace binárního kódu) používají neuronové sítě k navrhování popisných názvů funkcí na základě jejich kódu a kontextu [41].
- **Sumarizace Binárního Kódu:** Cílem je automaticky generovat stručné popisy v přirozeném jazyce, které vysvětlují, co daná funkce nebo část kódu dělá. Příklady projektů zahrnují BinT5 (rozšiřující CodeT5), HexT5 (jednotný model pro více úloh), CP-BCS (založený na CFG a pseudokódu) a MiSum (multimodální přístup) [41].
- **Obnova Názvů a Typů Proměnných:** Neuronový model VarBERT využívá techniku *transfer learning* z velkého datasetu lidmi psaného zdrojového kódu k predikci názvů proměnných a jejich původu v dekompilovaném výstupu [5]. Hybridní technika ReSym kombinuje LLM s tradiční programovou analýzou pro obnovu názvů a typů lokálních proměnných a také uživatelsky definovaných datových struktur (např. `struct`) [6].

Budoucnost spočívá v synergii tradičních metod a AI. Tradiční metody poskytují robustní základní analýzu (např. konstrukci CFG), zatímco AI techniky excelují v zlepšování čitelnosti a obnově sémantických informací.

2.1.2 Modelově Řízené Reverzní Inženýrství

MDRE: Tento přístup využívá principy modelově řízeného inženýrství k analýze a pochopení existujících softwarových systémů. Hlavní myšlenkou je automatické nebo poloautomatické generování různých modelů (např. UML diagramy tříd, aktivit, sekvenční diagramy;

Knowledge Discovery Metamodel - Knowledge Discovery Metamodel (KDM); Abstract Syntax Tree - Abstract Syntax Tree (AST)) ze stávajícího softwaru [42]. Tyto modely reprezentují systém z různých perspektiv na vyšší úrovni abstrakce než samotný kód a mohou sloužit k dokumentaci, modernizaci nebo refaktoringu. Níže jsou shrnuty jeho hlavní charakteristiky.

Výhody: MDRE nabízí několik klíčových výhod pro analýzu a modernizaci softwarových systémů. Potenciál pro automatizaci (až 62 % přístupů je plně automatizovaných [42]), poskytování vysokoúrovňových abstrakcí pro snazší pochopení komplexních systémů, generování dokumentace a podpora evoluce systémů.

Výzvy: Přes svou užitečnost čelí MDRE několika významným výzvám, které je třeba adresovat. Zajištění přesnosti u velkých a komplexních systémů, podpora heterogenity programovacích jazyků a platform, škálovatelnost, adekvátní zachycení dynamického chování a správná interpretace sémantiky zdrojového kódu [42].

2.1.3 Pokročilé Statické a Dynamické Analytické Techniky

Kromě tradiční dekompile a nově se rozvíjejících metod založených na AI/ML, existuje řada dalších pokročilých statických a dynamických analytických technik, které hrají klíčovou roli v moderním reverzním inženýrství [43]. Tyto techniky často poskytují hlubší vhled do chování programu nebo pomáhají ověřovat a zlepšovat výsledky jiných metod.

Fuzzing Dekompilátorů: Jedná se o specifickou aplikaci fuzzingu, robustní metody testování softwaru, která v tomto případě cílí na samotné dekompilátory. Metoda zahrnuje poskytování velkého množství náhodných, polo-náhodných nebo strukturovaných dat jako vstupu do programu (dekompilátoru) s cílem vyvolat neočekávané chování, pády nebo odhalit sémantické chyby, tedy situace, kdy dekompilovaný kód funkčně neodpovídá původnímu binárnímu souboru.

- **BIN2WRONG:** Příklad pokročilého fuzzeru, který systematicky mutuje různé aspekty generování binárních souborů (zdrojový kód, kompilátor, optimalizační příznaky, formát souboru) k vytvoření široké škály testovacích případů pro dekompilátory [7].

Symbolická Exekuce: Tato pokročilá metoda analýzy programu zkoumá chování softwaru bez jeho skutečného spuštění s konkrétními daty; místo toho používá symbolické pro-

měnné pro reprezentaci vstupních hodnot. Program je „vykonáván“ tak, že se sledují podmínky (*path conditions*) vedoucí k různým cestám v kódu. Pomocí *Satisfiability Modulo Theories* (SMT) (*Satisfiability Modulo Theories*) solverů lze pak najít konkrétní vstupy, které splní dané podmínky a dosáhnou určité části kódu, což je neocenitelné pro hledání zranitelností nebo generování testovacích případů [44, 39].

Konkolická Exekuce: Představuje hybridní přístup, který efektivně kombinuje konkrétní (*CONCcrete*) exekuci programu s reálnými vstupy a symbolickou (*symBOLIC*) exekuci. Spouští program s konkrétními vstupy a zároveň sbírá symbolické podmínky podél vykonané cesty. Tyto podmínky jsou pak negovány a řešeny pomocí SMT solveru, aby se našly nové vstupy pro prozkoumání alternativních, dosud nepokrytých cest v kódu, čímž se zvyšuje pokrytí analýzy.

Formální Metody: Zahrnují aplikaci matematicky rigorózních technik pro specifikaci, vývoj a především verifikaci softwarových a hardwarových systémů. V kontextu RE mohou formální metody pomoci dokázat určité vlastnosti dekompilovaného kódu (např. absenci určitých typů chyb), ověřit sémantickou ekvivalenci mezi původním a dekompilovaným kódem, nebo formálně specifikovat chování kritické části systému pro další analýzu.

Techniky pro Zlepšení Struktury Řídicího Toků: Cílem těchto specializovaných technik je algoritmicky restrukturalizovat dekompilovaný kód tak, aby se co nejvíce podobal původnímu, lidsky psanému zdrojovému kódu, byl čitelnější a lépe strukturovaný, například redukcí počtu *goto* příkazů a obnovou standardních smyčkových a podmínkových konstrukcí.

- **SAILR:** Příklad projektu, který se zaměřuje na „vrácení zpět“ některých optimalizací provedených kompilátorem a inteligentní redukcí *goto* příkazů pro zlepšení čitelnosti CFG [5].

S nástupem LLM se mění i hodnocení kvality dekompilace – důraz se klade nejen na sémantickou ekvivalenci, ale stále více i na „čitelnost“ a „pochopitelnost“ kódu pro člověka-analytika [44].

2.2 Analýza Nástrojů pro Reverzní Inženýrství C/C++ a Jejich Efektivita při Získávání Zdrojového Kódu

Výběr správných nástrojů je zásadní pro úspěšné reverzní inženýrství [40, 45]. Efektivita nástrojů při „získávání zdrojového kódu“ se hodnotí podle kvality, čitelnosti, přesnosti a znovupoužitelnosti generovaného pseudokódu nebo dekompilovaného kódu podobného C/C++. Následující sekce detailněji rozebírají klíčové kategorie nástrojů.

2.2.1 Komerční a Open-Source Dekompilátory

Na trhu existuje řada dekompilečních nástrojů, od vysoce specializovaných komerčních produktů po flexibilní open-source alternativy. Každý z nich nabízí specifickou sadu funkcí a liší se v kvalitě generovaného pseudokódu, podpoře C++ konstrukcí a ceně.

IDA Pro (s dekompilátorem Hex-Rays): Tento nástroj je dlouhodobě považován za zlatý standard v oblasti reverzního inženýrství a dekompile, zejména pro C/C++ aplikace.

- **Přehled:** Dlouhodobě považován za průmyslový standard, komerční disassembler a debugger s výkonným dekompilátorem Hex-Rays [46]. Detailní dokumentace je dostupná [47].
- **Efektivita při získávání pseudokódu:** Hex-Rays generuje vysoce kvalitní a relativně čitelný C-like pseudokód. Má pokročilou podporu pro C++ konstrukty, včetně demanglingu jmen, analýzy tříd, virtuálních tabulek (vtables) a od verze 9.0 i vylepšenou dekompilaci C++ výjimek (pro MSVC++ x64) [48]. Rozpoznává mnoho idiomů generovaných různými C/C++ kompilátory.
- **Klíčové funkce pro C/C++:** Disassembler (>60 procesorů, *Portable Executable* (PE), ELF, Mach-O), technologie FLIRT, integrovaný debugger (Windows TTD, Linux, macOS).
- **Skriptování:** IDAPython, IDC, C++ *Software Development Kit* (SDK).
- **Licence:** Komerční, model předplatného. Ceny od ~ \$365/rok do ~ \$8,599/rok [49].
- **Silné stránky:** Nejpokročilejší dekompile pro C/C++, rozsáhlá podpora, výkonný debugger.

- **Slabé stránky:** Velmi vysoká cena. Sémantická přesnost se pohybuje kolem 55% [50] a klesá s optimalizacemi [44]; generovaný kód zřídka přímo spustitelný.

Ghidra (NSA): Vyvinutý Národní bezpečnostní agenturou (NSA) Spojených států a uvolněný jako open-source, Ghidra se rychle stal populární a výkonnou alternativou k komerčním nástrojům.

- **Přehled:** Bezplatný, open-source framework od NSA, rychle získal popularitu jako alternativa k IDA Pro [51].
- **Efektivita při získávání pseudokódu:** Vestavěný dekompilátor generuje C-like pseudokód pro širokou škálu architektur (díky jazyku Sleigh). Podporuje analýzu C++ tříd a struktur; Data Type Manager umožňuje definovat datové typy. Dokáže zpracovat STL objekty [52] a pokouší se určit signatury funkcí včetně thiscall.
- **Klíčové funkce pro C/C++:** Podpora procesorů (Sleigh), formáty (PE, ELF, Mach-O), debugger (od v10.0, WinDbg, GDB), emulátor.
- **Skriptování:** Java, Python (přes Jython).
- **Licence:** Apache License 2.0 (zdarma).
- **Silné stránky:** Zdarma a open-source, výkonný dekompilátor, multiplatformní, rozšiřitelná podpora architektur.
- **Slabé stránky:** Kvalita dekompilovaného C++ kódu může být v komplexních případech nižší než u Hex-Rays. Výkon u velmi velkých souborů.

Radare2 (r2) / Rizin (s GUI Cutter): Tyto open-source frameworky, primárně ovládané z příkazové řádky, nabízejí extrémní flexibilitu a širokou podporu architektur, přičemž Cutter poskytuje grafické rozhraní pro Rizin.

- **Přehled:** Bezplatné, open-source frameworky (Rizin je fork r2) primárně řízené z Command Line Interface (Rozhraní příkazového řádku) (CLI). Cutter poskytuje GUI pro Rizin [53].
- **Efektivita při získávání pseudokódu:** Nejlepší kvality dekompilace se dosahuje integrací dekompilátoru Ghidra přes pluginy r2ghidra [54] nebo rz-ghidra. Tyto poskytují C-like pseudokód srovnatelný s Ghidra.

- **Klíčové funkce pro C/C++:** Výkonná statická analýza, vestavěný debugger, obrovská podpora architektur a formátů.
- **Skriptování:** r2pipe (Python, Javascript, Go atd.).
- **Licence:** *GNU Lesser General Public License version 3* (LGPLv3) (zdarma).
- **Silné stránky:** Vysoce skriptovatelný, masivní podpora architektur/formátů, integrace Ghidra dekompilátoru.
- **Slabé stránky:** Strmá křivka učení (CLI). Kvalita nativní dekompilace (bez Ghidra pluginu) je omezená [55].

x64dbg / x32dbg: Jedná se o populární open-source debuggery pro operační systém Windows, které jsou neocenitelné pro dynamickou analýzu, avšak samy o sobě neposkytují dekompilaci do vysokoúrovňového kódu.

- **Přehled:** Bezplatné, open-source debuggery pro Windows, zaměřené na analýzu malware a RE [56]. Často používané při analýze malware [57].
- **Efektivita při získávání pseudokódu:** Neposkytují nativní dekompilaci do C-like pseudokódu. Jsou klíčové pro dynamickou analýzu.
- **Klíčové funkce pro C/C++ (Windows):** Ladění Dynamic Link Library (Dynamicky linkovaná knihovna) (DLL)/*Executable File* (EXE), breakpointy, sledování paměti/registrů, patchování, rekonstruktor importů (Scylla).
- **Licence:** *GNU General Public License version 3* (GPLv3) (zdarma).
- **Silné stránky:** Specializovaný Windows debugger, open-source, aktivní vývoj.
- **Slabé stránky:** Pouze pro Windows. Chybí funkce dekompilace.

Binary Ninja (Vector 35): Tento komerční nástroj se vyznačuje moderním uživatelským rozhraním a silným API, přičemž využívá vlastní víceúrovňové intermediární jazyky (BNIL) pro analýzu a dekompilaci.

- **Přehled:** Komerční, interaktivní dekompilátor, disassembler, debugger s moderním User Interface (Uživatelské rozhraní) (UI) a výkonným API [58].
- **Efektivita při získávání pseudokódu:** Vestavěný dekompilátor generuje výstup do C a různých úrovní BNIL (Binary Ninja Intermediate Languages).

HLIL (High-Level Intermediární Jazyk (*Intermediate Language*) (IL)) je nejbližší C-like pseudokódu a je dobře hodnocen pro čitelnost. Silné API (Python, C++, Rust) umožňuje programově interagovat s IL.

- **Klíčové funkce pro C/C++:** Dekompilátor (C, BNIL), architektury (x86, *Advanced RISC Machine* (ARM), *PowerPC* (PPC), *Microprocessor without Interlocked Pipeline Stages* (MIPS), *RISC Five* (RISC-V)), formáty (PE, Mach-O, ELF), integrovaný debugger.
- **Licence:** Komerční (Free Edition, Personal ~ \$299, Commercial ~ \$1499) [59]. Sidekick (AI) je samostatné předplatné.
- **Silné stránky:** Moderní UI, vynikající API, víceúrovňové IL, aktivní vývoj.
- **Slabé stránky:** Zralost dekompilátoru pro extrémně komplexní C++ kód nemusí dosahovat úrovně Hex-Rays. Cena komerční verze.

JEB Decompiler (PNF Software): Komerční platforma známá svou modularitou a silnou podporou pro analýzu Android aplikací, ale nabízí také robustní dekompilaci nativního C/C++ kódu pro různé architektury.

- **Přehled:** Komerční, modulární platforma, známá analýzou Androidu, ale se silnou podporou pro nativní kód C/C++ [60].
- **Efektivita při získávání pseudokódu:** Poskytuje nativní dekompilátory pro klíčové architektury (Intel x86/x64, ARM/*ARM 64-bit* (ARM64), MIPS, RISC-V) s výstupem do C. Zmiňuje „částečnou obnovu C++ tříd pro MS VCPP“ a pokročilé optimalizační průchody pro obfuskovaný C/C++ kód.
- **Licence:** Komerční, předplatné (JEB Pro ~ \$2000/rok). JEB Community Edition (zdarma, omezené x86/x64 dekompilátory).
- **Silné stránky:** Silné nativní dekompilátory, zaměření na obfuskaci.
- **Slabé stránky:** Cena. Hloubka dekompilace pokročilých C++ prvků nemusí být tak rozvinutá jako u IDA Pro.

Hopper Disassembler: Tento komerční nástroj, primárně zaměřený na ekosystémy macOS a Linux, poskytuje cenově dostupnou alternativu pro dekompilaci a analýzu, se zvláštním důrazem na Objective-C a Swift.

- **Přehled:** Komerční disassembler, dekompilátor, debugger primárně pro macOS a Linux, analyzuje i Windows a iOS [61].
- **Efektivita při získávání pseudokódu:** Poskytuje pseudokódovou reprezentaci podobnou C. Silná specializace na ekosystém Apple (Objective-C, Swift).
- **Licence:** Komerční (~ \$90–\$99), demo verze.
- **Silné stránky:** Cenově dostupný, vynikající podpora pro Objective-C a Swift.
- **Slabé stránky:** Neběží nativně na Windows. Zralost dekompilátoru pro generický C++ kód nemusí dosahovat úrovně Hex-Rays.

RetDec (Avast): Původně vyvinutý jako open-source projekt, RetDec je retargetovatelný dekompilátor založený na LLVM, který se zaměřuje na rekonstrukci C/C++ kódu, ačkoli je v současnosti v režimu omezené údržby.

- **Přehled:** Bezplatný, open-source, retargetovatelný dekompilátor založený na LLVM. V režimu omezené údržby [62].
- **Efektivita při získávání pseudokódu:** Dekompiluje do C a jazyka podobného Pythonu. Cílem je rekonstruovat C++ hierarchie tříd a demanglovat symboly.
- **Licence:** *Massachusetts Institute of Technology* (MIT) License (zdarma).
- **Silné stránky:** Open-source, široká podpora architektur, zaměření na C++ specifické funkce.
- **Slabé stránky:** Režim omezené údržby. Kvalita dekompilace nemusí odpovídat aktivně vyvíjeným nástrojům.

Následující Tabulka 2.1 poskytuje srovnání těchto nástrojů.

2.2.2 Nástroje s Podporou AI a Jejich Integrace

Integrace AI, zejména LLM, do RE nástrojů je silným trendem, který slibuje zefektivnění práce analytiků a zlepšení srozumitelnosti dekompilovaného kódu. Následující příklady ilustrují tento vývoj.

- **r2ai (pro Radare2):** Tento inovativní plugin propojující Radare2 s různými LLM umožňuje analytikům interaktivně klást dotazy ohledně analyzovaného kódu. Pomáhá

navrhovat smysluplné názvy pro funkce a proměnné, vysvětlovat funkcionalitu bloků kódu, asistovat při dekompilaci (např. „překladem“ assembleru do srozumitelnější podoby) a dokonce generovat skripty pro specifické úkoly, jako je deobfuskace [63].

- **SideKick (pro Binary Ninja):** Jedná se o oficiální AI rozšíření pro Binary Ninja, které má za cíl poskytovat podobné asistenční funkce jako r2ai, avšak plně integrované do prostředí Binary Ninja, čímž zvyšuje efektivitu při analýze a porozumění kódu [64].
- **Pluginy pro IDA Pro a Ghidra:** Pro tyto zavedené platformy existuje rostoucí počet komunitních i komerčních pluginů, jako jsou například IDA-Assistant, GptHidra nebo GhidrAssist. Tyto pluginy se snaží přinést sílu LLM do těchto robustních prostředí, pomáhají s anotací kódu, vysvětlováním jeho částí a dalšími úkoly vedoucími k lepšímu a rychlejšímu pochopení analyzovaného softwaru [63].

AI přímo negeneruje „zdrojový kód“ ve smyslu tradiční dekompilace s garantovanou sémantickou ekvivalencí, ale výrazně zvyšuje efektivitu analytika při interpretaci a vylepšování výstupů z tradičních dekompilátorů. Pomáhá překlenout sémantickou mezeru a zlepšit čitelnost, což je klíčové pro „získání“ co nejsrozumitelnější reprezentace původní logiky. Modely jako LLM4Decompile [4] však již přímo cílí na generování kódu z binární podoby a ukazují potenciál v produkci čitelnějšího a dokonce znovuspustitelného kódu.

2.2.3 Specializované Nástroje a Frameworky

Kromě samotných dekompilátorů existují specializované nástroje a frameworky, které sice přímo nepřispívají k „získávání zdrojového kódu“ ve smyslu jeho generování, ale jsou klíčové pro hodnocení, testování a zlepšování kvality a spolehlivosti dekompilačních nástrojů a metod.

- **Fuzzery dekompilátorů:** Tyto nástroje aplikují techniku fuzzingu specificky na dekompilátory s cílem odhalit jejich chyby a slabiny.
 - **BIN2WRONG:** Tento pokročilý fuzzer systematicky generuje rozmanité binární testovací případy mutováním zdrojového kódu, kompilátorů a jejich nastavení, aby odhalil sémantické chyby v dekompilátorech, tedy situace, kdy dekompilovaný

kód neodpovídá funkčnosti původního binárního souboru [7]. Jeho nálezy vedly ke zlepšení kvality komerčních dekompilátorů.

- **Benchmarky pro dekompilaci:** Standardizované benchmarky jsou nezbytné pro objektivní srovnání různých dekompilačních přístupů a měření skutečného pokroku v oboru.
 - **DecompileBench:** Jedná se o komplexní a moderní framework navržený pro hodnocení dekompilátorů (tradičních i LLM-založených) na souboru reálných programů. Zahrnuje nejen runtime validaci (ověření funkční ekvivalence dekompilovaného kódu), ale také inovativní přístup „LLM-as-a-Judge“ pro škálovatelné hodnocení srozumitelnosti, užitečnosti a dalších kvalitativních aspektů dekompilovaného kódu z lidské perspektivy [44].
 - **Decompile-Bench:** Tento rozsáhlý dataset, vytvořený autory LLM4Decompile, obsahuje páry binární kód – zdrojový kód a je určen specificky pro trénink a evaluaci velkých jazykových modelů pro úlohu dekompilace [4].

Tyto specializované nástroje a benchmarky jsou zásadní pro vědecký pokrok v oblasti dekompilace. Fuzzery pomáhají zlepšovat kvalitu a spolehlivost samotných dekompilačních nástrojů, zatímco standardizované benchmarky umožňují objektivní srovnání různých přístupů a měření skutečného pokroku.

2.2.4 Efektivita a Omezení Současných Nástrojů při Získávání Kódu

Přestože moderní nástroje pro reverzní inženýrství dosáhly značného pokroku, stále se potýkají s řadou omezení, která ovlivňují jejich efektivitu při „získávání“ kódu blížícího se původnímu zdrojovému kódu.

- **Sémantická konzistence/Přesnost:** Žádný dekompilátor negarantuje 100% sémantickou ekvivalenci mezi původním binárním kódem a generovaným pseudokódem. Studie ukazují, že i špičkové nástroje jako Hex-Rays mají přesnost kolem 55% [50], a tato přesnost dále klesá s použitím agresivních kompilátorových optimalizací [44]. LLM metody mohou v současnosti vykazovat ještě nižší funkční korektnost, i když mohou excelovat v jiných aspektech.

- **Čitelnost a Srozumitelnost:** Tradiční dekompilátory často produkují kód, který je sice funkčně (částečně) ekvivalentní, ale jeho struktura je nepřehledná, názvy proměnných a funkcí jsou generické a celkově je kód obtížně čitelný a srozumitelný pro člověka. LLM zde ukazují velký potenciál pro generování idiomatictějšího a strukturovanějšího kódu.
- **Podpora moderních C++ rysů:** Jazyk C++ se neustále vyvíjí (C++11 až C++23), přinášejíc nové a komplexní jazykové konstrukce (např. korutiny, moduly, koncepty, ranges). Současné dekompilátory mohou mít problémy s jejich správnou interpretací a rekonstrukcí, což vede k nepřesnému nebo nesrozumitelnému pseudokódu [65, 66].
- **Obfuskace a anti-reverzní techniky:** Záměrné techniky pro znesnadnění analýzy, jako jsou různé formy obfuskace kódu, anti-debugging a anti-VM triky, představují významnou překážku pro všechny typy nástrojů a mohou výrazně snížit jejich efektivitu nebo analýzu zcela znemožnit [65, 43, 40].
- **Znovuspustitelnost:** Dekompilovaný kód je jen zřídka přímo zkompilovatelný a spustitelný bez rozsáhlých manuálních úprav a doplnění chybějících informací (např. hlavičkové soubory, definice typů, linkovací informace). Nicméně, LLM-založené dekompilátory, jako je LLM4Decompile, vykazují v této oblasti slibný pokrok [4].

Nejefektivnější přístup často spočívá v kombinaci více nástrojů a technik, kde silné stránky jednoho nástroje kompenzují slabiny druhého, a kde klíčovou roli hraje expertíza a zkušenost lidského analytika.

2.3 Výběr a Charakteristika Aplikací Různého Typu pro Praktické Ověření Metod

Pro komplexní ověření efektivit různých metod a nástrojů reverzního inženýrství je nezbytné vybrat portfolio aplikací, které reprezentují široké spektrum výzev. Následující kategorie aplikací byly zvoleny s ohledem na jejich relevanci a specifické charakteristiky, které testují různé aspekty RE procesů.

2.3.1 Analýza Malware (Škodlivého Softwaru)

Malware představuje jednu z nejnáročnějších oblastí pro reverzní inženýrství kvůli aktivní snaze jeho autorů znesnadnit analýzu.

Charakteristika: Malware je software záměrně navržený tak, aby poškodil, zneužil nebo získal neoprávněný přístup k počítačovým systémům. Často je psán v C/C++ pro dosažení vysokého výkonu, nízkourovňové kontroly nad systémem a minimalizaci velikosti. Autoři malware aktivně používají širokou škálu anti-reverzních technik [65, 57].

Specifické výzvy pro ověření metod a nástrojů: Analýza malware klade na nástroje a metody RE mimořádné nároky.

- **Obfuskace kódu:** Techniky jako přejmenování proměnných a funkcí na nicneříkající názvy, vkládání irelevantního („mrtvého“) kódu, šifrování řetězců a citlivých dat, a především úpravy řídicího toku (např. zploštění CFG, použití neprůhledných predikátů, virtualizace kódu) [65, 43]. Toto intenzivně testuje schopnost dekompilátorů rekonstruovat původní logiku a srozumitelnost kódu, a také efektivitu AI metod při deobfuskaci a obnově sémantiky.
- **Anti-debugging a Anti-VM/Sandbox Evasion:** Malware se často snaží detekovat přítomnost debuggeru nebo virtualizovaného/sandboxovaného prostředí a v takovém případě mění své chování nebo se ukončí [67, 57]. To ověřuje robustnost nástrojů pro dynamickou analýzu a schopnost analytika tyto ochrany identifikovat a obejít.
- **Packers/Loaders:** Hlavní škodlivý kód (payload) je často komprimován nebo zašifrován a obalen do „spouštěče“ (loader) [67, 57]. To ztěžuje statickou analýzu a testuje schopnost nástrojů pro unpacking a analýzu kódu přímo v paměti.
- **Použití méně obvyklých jazyků/kompilátorů:** Někteří autoři malware sahají po jazycích jako Rust, Go, Nim, Delphi, nebo používají méně běžné C++ kompilátory (např. Borland C++, Pelles C), aby snížili míru detekce antivirovými programy a ztížili reverzní inženýrství [68]. To ověřuje adaptabilitu nástrojů (např. podporu různých ABI, knihoven) a znalosti analytika.

Relevance pro ověření: Analýza malware je klíčová pro testování odolnosti dekompilátorů vůči záměrnému zmatení, efektivitě deobfuskáčnických technik (manuálních i asistovaných AI) a schopnosti dynamických nástrojů sledovat a odhalit skutečné chování škodlivého kódu.

2.3.2 Výzkum Zranitelností v Proprietárním Softwaru

Identifikace bezpečnostních chyb v softwaru bez dostupného zdrojového kódu je další důležitou aplikací reverzního inženýrství.

Charakteristika: Cílem je identifikovat a analyzovat bezpečnostní chyby v softwaru, ke kterému není veřejně dostupný zdrojový kód (např. komponenty operačních systémů, webové prohlížeče, uzavřené softwarové knihovny, síťové služby) [40]. Tento software je často komplexní, optimalizovaný pro výkon a může obsahovat miliony řádků kódu.

Specifické výzvy pro ověření metod a nástrojů: Tento typ analýzy vyžaduje přesné a spolehlivé nástroje schopné analyzovat rozsáhlé a optimalizované kódové báze.

- **Identifikace zranitelných vzorů:** Hledání chyb jako přetečení bufferu, *use-after-free*, celočíselná přetečení, logické chyby v bezpečnostních mechanismech atd., v dekompilovaném kódu. To testuje přesnost a čitelnost dekompilovaného kódu a schopnost analytika interpretovat potenciálně nebezpečné konstrukce.
- **Analýza komplexní logiky:** Porozumění složitým algoritmům, stavovým strojům a interakcím mezi různými moduly nebo komponentami. Ověřuje schopnost nástrojů (včetně pokročilých technik jako symbolická exekuce [39]) mapovat cesty kódu, sledovat datové toky napříč funkcemi a identifikovat kritické body.
- **Práce s optimalizovaným kódem:** Kompilátorové optimalizace (inlining, odvíjení smyček, reordering instrukcí) mohou výrazně změnit strukturu kódu oproti původnímu zdrojovému kódu a systematicky mazat vysokoúrovňovou sémantiku [44]. Testuje schopnost dekompilátorů (např. Hex-Rays, Ghidra) a metod (např. SAILR) co nejvěrněji rekonstruovat původní logiku.

Relevance pro ověření: Tento typ aplikace ověřuje přesnost dekompilátorů při analýze rozsáhlých a optimalizovaných kódovýchází, užitečnost pokročilých technik statické a dynamické analýzy (včetně fuzzingu cíleného na konkrétní rozhraní) a schopnost nástrojů pomoci odhalit jemné bezpečnostní chyby.

2.3.3 Analýza Firmware Vestavěných Zařízení (*Embedded Systems*)

Firmware vestavěných zařízení představuje unikátní sadu výzev kvůli své těsné vazbě na hardware a často nestandardním prostředím.

Charakteristika: Firmware je software běžící na specializovaném hardwaru v rámci vestavěných zařízení (např. *Internet of Things* (IoT) zařízení, routery, průmyslové řídicí systémy (*Industrial Control Systems* (ICS)), automobilová elektronika). Je často psán v C nebo C++ kvůli požadavkům na výkon, přímou kontrolu nad hardwarem a omezené zdroje [69].

Specifické výzvy pro ověření metod a nástrojů: Analýza firmwaru vyžaduje nástroje schopné pracovat s různými architekturami a nízkoúrovňovým kódem.

- **Získání a příprava firmwaru:** Firmware nemusí být snadno dostupný a může vyžadovat hardwarové přístupy (např. JTAG, ISP, přímé čtení z paměťových čipů) [69]. Může být také šifrovaný nebo komprimovaný, což vyžaduje další kroky před samotnou analýzou.
- **Nestandardní formáty souborů a architektury:** Firmware může používat vlastní formáty souborů nebo méně běžné procesorové architektury (např. různé varianty ARM, MIPS, RISC-V, nebo proprietární mikrokontroléry). Testuje flexibilitu nástrojů (např. Ghidra s jazykem Sleigh, podpora architektur v IDA Pro, Binary Ninja) a jejich schopnost správně disasemblovat a dekompileovat kód pro tyto cíle.
- **Analýza interakce s hardwarem:** Porozumění přímé manipulaci s hardwarovými registry (*Memory-Mapped I/O* (MMIO)), obsluze přerušení, komunikaci s periferiemi. Vyžaduje hluboké znalosti cílového hardwaru a nástroje schopné modelovat nebo asistovat při analýze těchto nízkoúrovňových interakcí.
- **Omezené zdroje a specifické knihovny:** Firmware často běží na zařízeních s omezenou pamětí a výpočetním výkonem, což může ovlivnit styl kódu (např. absence dynamické alokace paměti, použití specifických **RTOS!** (**RTOS!**) nebo *bare-metal* programování).

Relevance pro ověření: Analýza firmwaru testuje schopnost nástrojů pracovat s různými (i

exotickými) architekturami, analyzovat nízkourovňový kód interagující přímo s hardwarem, a schopnost dekompilátorů produkovat smysluplný výstup pro specifické embedded platformy, kde mohou chybět standardní knihovny nebo operační systém.

2.3.4 Reverzní Inženýrství Herních Enginů a Anti-Tamper Mechanizmů

Moderní videohry s jejich komplexními C++ enginy a sofistikovanými ochranami představují jednu z největších výzev pro reverzní inženýrství.

Charakteristika: Moderní videohry často používají komplexní C++ herní enginy (např. Unreal Engine, Unity, CryEngine, nebo vlastní proprietární enginy) a implementují sofistikované ochranné mechanismy (*anti-tamper*, *anti-cheat*, DRM) k ochraně duševního vlastnictví a zabránění podvádění.

Specifické výzvy pro ověření metod a nástrojů: Analýza těchto systémů testuje absolutní limity současných RE nástrojů a technik.

- **Rozsáhlé a komplexní C++ kódové báze:** Herní enginy mohou obsahovat miliony řádků C++ kódu s intenzivním využitím objektově orientovaného programování, šablon, a dalších pokročilých C++ rysů. Testuje škálovatelnost, výkon a hloubku analýzy C++ v nástrojích.
- **Silná obfuskace a virtualizace kódu:** Ochrany jako VMProtect, Themida, Denuvo, nebo vlastní řešení mohou virtualizovat kritické části kódu (přeložit je do vlastního bytecode pro interní VM), šifrovat kód a data, a používat další pokročilé techniky znesnadňující analýzu [43]. Představuje to jednu z nejvyšších výzev pro dekompilátory a analytické techniky.
- **Anti-debug a anti-analysis techniky:** Podobně jako u malware, hry mohou obsahovat robustní mechanismy pro detekci debuggerů, monitorovacích nástrojů a modifikací kódu.
- **Dynamicky generovaný kód a polymorfismus:** Některé ochrany mohou za běhu generovat nebo modifikovat kód, což dále komplikuje statickou analýzu.

Relevance pro ověření: Hry s pokročilými ochranami jsou vynikajícím testem absolutních limitů dekompilačních nástrojů, deobfuskačních technik (manuálních i automatizova-

ných) a schopnosti AI asistovat při analýze extrémně složitého a aktivně chráněného kódu. Testuje také schopnost nástrojů pro dynamickou analýzu obejít anti-debug triky.

2.3.5 Analýza Staršího (*Legacy*) Softwaru

Reverzní inženýrství starších systémů, pro které již nemusí existovat zdrojové kódy nebo dokumentace, je často nezbytné pro jejich údržbu nebo migraci.

Charakteristika: Jedná se o starší softwarové systémy, které mohou být stále v provozu, ale pro které již nemusí existovat původní zdrojové kódy, dokumentace, nebo původní vývojáři [70]. Cílem reverzního inženýrství může být údržba (oprava chyb), migrace na nové platformy, porozumění nedokumentovaným funkcím nebo datovým formátům, nebo integrace s novými systémy.

Specifické výzvy pro ověření metod a nástrojů: Analýza staršího softwaru testuje schopnost nástrojů pracovat se zastaralými technologiemi a chybějícím kontextem.

- **Zastaralé technologie:** Kód mohl být zkompileován starými, dnes již nepoužívanými kompilátory, pro zastaralé operační systémy nebo hardwarové platformy. Testuje zpětnou kompatibilitu nástrojů a jejich schopnost analyzovat kód generovaný těmito staršími technologiemi (např. podpora starých volacích konvencí, formátů souborů).
- **Chybějící kontext a symboly:** Bez zdrojového kódu, dokumentace a ladicích symbolů je velmi obtížné pochopit původní záměr a logiku programu. Testuje schopnost nástrojů (zejména dekompilátorů a AI asistentů) obnovit co nejvíce sémantických informací a generovat srozumitelné popisy z „holého“ binárního kódu.
- **Velká komplexita nahromaděná léty vývoje:** Legacy systémy mohou být velmi rozsáhlé a obsahovat vrstvy kódu přidané v průběhu mnoha let, často bez konzistentního designu.

Relevance pro ověření: Ověřuje schopnost dekompilátorů a analytických metod extrahovat užitečné informace a rekonstruovat logiku z binárních souborů, které mohou být staré, postrádat jakékoli moderní ladicí informace nebo symboly, a byly vytvořeny technologiemi, které již nejsou běžně podporovány.

2.3.6 Úlohy z CTF Soutěží (*Capture The Flag*)

Soutěže typu CTF poskytují kontrolované a často velmi náročné prostředí pro testování specifických RE dovedností a nástrojů.

Charakteristika: CTF soutěže v oblasti kybernetické bezpečnosti často obsahují úlohy zaměřené na reverzní inženýrství. Tyto úlohy obvykle zahrnují analýzu malých, účelově vytvořených C/C++ binárních souborů s cílem najít skrytou „vlajku” (*flag*) – specifický řetězec nebo hodnotu – často vyžadující pochopení algoritmu, dešifrování dat nebo nalezení skryté logiky [65].

Specifické výzvy pro ověření metod a nástrojů: CTF úlohy jsou navrženy tak, aby testovaly konkrétní znalosti a schopnosti analytiků.

- **Zaměření na konkrétní triky, techniky a algoritmy:** Úlohy mohou být navrženy tak, aby testovaly znalost specifických typů obfuskace, kryptografických primitiv, matematických hádanek, nebo využití méně známých vlastností programovacího jazyka či chování kompilátoru.
- **Moderní C++ rysy:** Některé CTF úlohy, jako byl příklad „Decompetition: Vitamin C++” [66], se explicitně zaměřují na správnou interpretaci a rekonstrukci moderních C++ rysů (např. C++17 *structured bindings*, `const auto&`, *for-range* smyčky, lambda výrazy). Testuje hloubku podpory těchto rysů v dekompilátorech a schopnost analytika je rozpoznat v dekompilovaném kódu.
- **Časový tlak:** V CTF soutěžích je často omezený čas, což testuje rychlost a efektivitu analytika i nástrojů.

Relevance pro ověření: CTF úlohy poskytují kontrolované a často velmi náročné prostředí pro testování schopnosti nástrojů a analytiků řešit specifické, izolované problémy. Jsou užitečné pro ověření porozumění konkrétním aspektům C/C++ (včetně moderních standardů), chování kompilátoru, a pro nácvik technik deobfuskace a algoritmické analýzy v menším měřítku.

Výběr kombinace těchto typů aplikací umožňuje komplexní zhodnocení, jak si různé metody a nástroje vedou tváří v tvář rozmanitým výzvám.

2.4 Vyhodnocení Výsledků Aplikovaných Metod Reverzního Inženýrství

Hodnocení úspěšnosti a kvality metod RE, zejména dekompilace, je komplexní úkol. Nejde jen o to, zda se podařilo získat nějaký pseudokód, ale také o to, jak přesný, srozumitelný a použitelný tento kód je. S nástupem AI/ML metod se navíc objevují nové aspekty hodnocení.

2.4.1 Metriky pro Hodnocení Kvality Dekompilace

Pro objektivní posouzení výsledků dekompilace se používá několik klíčových metrik, které hodnotí různé aspekty kvality generovaného kódu. Přehled těchto metrik a jejich aplikací je uveden v Tabulce 2.2.

Sémantická konzistence/Přesnost (*Semantic Consistency/Accuracy*): Tato metrika hodnotí,

zda dekompilovaný kód funkčně odpovídá původnímu binárnímu kódu. Ideálně by měl dekompilovaný kód při spuštění produkovat stejné výsledky pro stejné vstupy jako originál [50]. Jak bylo zmíněno, studie ukazují, že i špičkové dekompilátory jako Hex-Rays mají sémantickou přesnost kolem 55%. LLM metody, ačkoliv mohou excelovat v jiných aspektech, mají podle benchmarku DecompileBench v průměru o 52.2% nižší funkční korektnost než tradiční nástroje [44]. Problémy s přesností jsou obzvláště kritické v bezpečnostní analýze.

Čitelnost/Srozumitelnost (*Readability/Understandability*): Tato metrika posuzuje, jak snadno

může člověk-analytik dekompilovaný kód přečíst, pochopit jeho logiku a identifikovat klíčové struktury a algoritmy. Tradiční dekompilátory často produkují kód, který strukturou připomíná spíše nízkoúrovňový assembler. LLM zde ukazují významný potenciál pro zlepšení, protože jsou trénovány na obrovském množství lidmi psaného kódu [44].

Znovuspustitelnost (*Re-executability*) / Znovukompilovatelnost (*Recompilability*): Hodnotí,

zda lze dekompilovaný kód znovu zkompilovat a spustit tak, aby se choval stejně jako původní binární soubor. Výstupy tradičních nástrojů často nejsou přímo spustitelné bez rozsáhlých manuálních úprav. LLM4Decompile dosáhl v benchmarkcích HumanEval a ExeBench (které testují schopnost generovat spustitelný kód) o více než 100% lepších výsledků než Ghidra nebo GPT-4o [4].

Blížkost k původnímu zdrojovému kódu: Pokud je k dispozici původní zdrojový kód (např. pro účely výzkumu), lze měřit, jak moc se dekompilovaný kód liší od originálu. Používá se například *Control Flow Graph Edit Distance* (CFGED) (*Control Flow Graph Edit Distance*), která měří minimální počet editačních operací potřebných k transformaci CFG dekompilovaného kódu na CFG původního zdrojového kódu [5].

Další metriky z DecompileBench: Framework DecompileBench navrhuje komplexní 12-dimenzionální hodnotící rámec, který zahrnuje aspekty jako jsou srozumitelnost řídicího toku (*Control Flow Clarity*), přesnost rozložení paměti (*Memory Layout Accuracy*), správnost typů, kvalita pojmenování proměnných a další, které jsou relevantní pro praktické reverzní inženýrství [44].

Hodnocení kvality dekompilace je tedy multidimenzionální. Různé metriky jsou relevantní pro různé cíle reverzního inženýrství.

2.4.2 Výsledky a Přesnost AI/ML Metod

Metody založené na AI a ML přinášejí do oblasti dekompilace nové možnosti, avšak jejich výsledky a přesnost se stále vyvíjejí a liší se v závislosti na konkrétní úloze, modelu a datech. Následující body shrnují některé publikované výsledky.

- **LLM4Decompile:** Tento specializovaný LLM dosáhl míry úspěšné dekompilace (generování kódu, který se zkompile a projde testy) 45.4% na benchmarku HumanEval a 18.0% na benchmarku ExeBench [4]. To jsou výrazně lepší výsledky než u tradičního dekompilátoru Ghidra nebo obecného LLM GPT-4o ve stejných úlohách.
- **ReSym:** Hybridní technika kombinující LLM a programovou analýzu pro obnovu symbolických informací. Dosahuje přesnosti 56.4% pro obnovu názvů lokálních proměnných a 64.6% pro obnovu jejich typů [6]. U rekonstrukce rozložení uživatelsky definovaných struktur (*struct layout*) dosahuje ReSym přesnosti (*precision*) 72.9%, ale nižší úplnosti (*recall*) 28.9%.
- **VarBERT:** Neuronový model zaměřený na predikci názvů proměnných. Na binárních souborech zkompilovaných bez optimalizací (O0) a dekompilovaných pomocí IDA dosáhl přesnosti predikce 50.70%, čímž překonal starší metody jako DIRE (35.94%) a DIRTY (38.00%) [5].

- **Obecné LLM (GPT-4, Gemini, LLaMa 3):** Studie ukázala, že tyto obecné LLM dosahují nižší výkonnosti při analýze dekompilovaného binárního kódu (úspěšnost 67% pro GPT-4, 54% pro Gemini, 33% pro LLaMa 3 v úlohách porozumění kódu) ve srovnání s jejich výkonností na původním zdrojovém kódu (75% pro GPT-4, 68% pro Gemini, 45% pro LLaMa 3) [72]. To naznačuje, že dekompilovaný kód představuje pro LLM větší výzvu.
- **AI asistence v r2ai:** Praktické zkušenosti s pluginem r2ai pro Radare2 ukazují, že kvalita analýzy s AI asistencí je často srovnatelná nebo dokonce lepší než analýza prováděná pouze člověkem. AI může výrazně zrychlit některé úkoly, jako je pochopení kódu nebo deobfuskace řetězců. Nicméně, AI vyžaduje pečlivé vedení zkušeným analytikem a je náchylná k „halucinacím“ nebo vynechávání důležitých detailů [63].

AI/ML metody tedy ukazují slibné výsledky, zejména v oblastech, kde tradiční metody často zaostávají, jako je obnova sémantických informací (názvy, typy, komentáře) a zlepšení celkové čitelnosti kódu. Jejich přesnost a spolehlivost se však stále vyvíjí a je třeba je hodnotit kriticky a v kontextu konkrétní úlohy.

2.4.3 Výzvy při Interpretaci Výsledků

Interpretace výsledků dekompilace, ať už z tradičních nástrojů nebo z AI/ML metod, není nikdy přímočará. Analytik musí brát v úvahu několik faktorů, které mohou ovlivnit kvalitu a spolehlivost dekompilovaného kódu.

- **Obfuskace:** Jak již bylo zmíněno, autoři malware a někteří vývojáři legitimního softwaru používají techniky obfuskace k znesnadnění analýzy. Obfuskovaný kód může vést k dekompilaci, která je neúplná, nesprávná, nebo extrémně obtížně srozumitelná [73, 43]. Dekompilátory mohou selhat při rozpoznávání skutečného řídicího toku nebo datových struktur.
- **Moderní C++ rysy (C++20/23):** Jazyk C++ se neustále vyvíjí a nové standardy (C++20, C++23) přinášejí pokročilé rysy jako korutiny, moduly, koncepty, ranges a další. Tyto moderní konstrukce mohou být pro současné dekompilátory obtížně analyzovatelné a jejich rekonstrukce do srozumitelné podoby v pseudokódu je velkou výzvou [65,

66, 74]. Příklad z CTF úlohy „Decompetition: Vitamin C++” ukázal, jak C++17 rysy ovlivňují dekompilovaný výstup.

- **Optimalizace kompilátoru:** Kompilátory provádějí řadu optimalizací (např. inlining funkcí, odvíjení smyček, odstraňování mrtvého kódu, reordering instrukcí) s cílem zlepšit výkon a zmenšit velikost výsledného binárního kódu. Tyto optimalizace však mohou dramaticky změnit strukturu kódu ve srovnání s původním zdrojovým kódem a systematicky mazat vysokoúrovňovou sémantiku, což výrazně ztěžuje dekompilaci. Například studie *DecompileBench* ukázala, že funkční korektnost dekompilátoru *Hex-Rays* klesla o 36.11% při použití agresivní optimalizační úrovně -O3 [44].
- **Halucinace LLM:** Velké jazykové modely, ačkoliv jsou schopny generovat velmi plausibilně vypadající kód nebo text, mohou také „halucinovat” – tj. generovat informace, které jsou nesprávné, irelevantní nebo si je zcela vymyslely [63]. Při použití LLM pro dekompilaci nebo asistenci při reverzním inženýrství je proto naprosto klíčové pečlivě validovat jejich výstupy.

Analytik tedy musí přistupovat k dekompilovanému kódu s kritickým myšlením a ověřovat svá zjištění.

2.4.4 Vliv Optimalizací Kompilátoru na Výsledky Dekompilace

Optimalizace prováděné kompilátorem mají zásadní a často negativní vliv na proces dekompilace a kvalitu jeho výsledků. Zatímco cílem optimalizací je vytvořit efektivnější a rychlejší kód, vedlejším efektem je ztráta informací a zvýšení komplexity pro reverzní inženýry.

- **Ztráta sémantiky a struktury:** Optimalizace jako *inlining* funkcí (nahrazení volání funkce jejím tělem), odvíjení smyček (rozepsání iterací smyčky), nebo odstraňování mrtvého kódu (kódu, který se nikdy nevykoná) systematicky mažou vysokoúrovňovou sémantiku a strukturu původního zdrojového kódu [44]. To ztěžuje dekompilátoru rekonstruovat původní záměry programátora a činí výsledný pseudokód méně čitelným.
- **Specifické chyby dekompilátorů indukované optimalizacemi:** Fuzzer *BIN2WRONG* odhalil, že některé chyby v dekompilátorech jsou specifické pro určité kombinace kompilátorů a použitých optimalizačních příznaků [7]. To znamená, že dekompilátor,

který funguje dobře pro kód zkompileovaný jedním způsobem, může selhat nebo produkovat nesprávné výsledky pro kód zkompileovaný jinak, zejména pokud byly použity agresivní optimalizace.

- **Snahy o „de-optimalizaci“ a restrukturalizaci:** Některé výzkumné projekty, jako je SAILR [5], se snaží aktivně „vrátit zpět“ některé kompilátorové optimalizace. Cílem je restrukturalizovat dekompileovaný kód tak, aby se více podobal původní, neoptimalizované verzi zdrojového kódu, což by mělo zlepšit jeho čitelnost a srozumitelnost (např. redukcí goto příkazů).
- **Problémy s mapováním na původní zdrojový kód:** Autoři benchmarku Decompile-Bench (Tan et al.) zdůrazňují, že agresivní kompilátorové optimalizace a všudypřítomné inlinování funkcí mohou zakrýt korespondenci mezi zkompileovaným binárním kódem a jeho původním zdrojem [4]. To činí přesné mapování (např. pomocí DWARF debug informací, pokud jsou k dispozici) neúplným a nespolehlivým, což komplikuje trénink a evaluaci dekompilačních modelů.

Porozumění běžným optimalizačním technikám a jejich dopadu na výsledný binární kód je proto pro reverzního inženýra velmi důležité. Umožňuje mu lépe interpretovat výstupy dekompilátorů a odhadnout, kde mohlo dojít ke ztrátě informací nebo ke zkreslení původní logiky.

2.5 Porovnání Efektivity Různých Metod Reverzního Inženýrství

Efektivita metod RE C/C++ aplikací závisí na mnoha faktorech, včetně typu analyzované aplikace, přítomnosti obfuskačních technik, cílů analýzy a dostupných zdrojů. Srovnání tradičních přístupů s moderními metodami založenými na AI/ML odhaluje zajímavé rozdíly v jejich silných a slabých stránkách.

2.5.1 Srovnání Tradičních Dekompilačních Technik s Metodami Založenými na AI/ML

Tradiční dekompilační techniky a metody založené na AI/ML představují dva odlišné přístupy k rekonstrukci kódu, každý s vlastními charakteristikami.

Tradiční dekompilační techniky: Tyto metody jsou založeny na explicitně definovaných pravidlech a algoritmech odvozených z teorie kompilátorů a programové analýzy.

- *Silné stránky:* Deterministické (výsledky jsou konzistentní a opakovatelné), založené na formální analýze (opírají se o dobře prostudované algoritmy pro analýzu řídicího a datového toku), dobré v rekonstrukci základní struktury (dokáží relativně spolehlivě rekonstruovat CFG a Data Flow Graph (Graf datového toku) (DFG)), relativně dobře prozkoumané (existuje rozsáhlé množství literatury a praktických zkušeností).
- *Slabé stránky:* Problémy s čitelností a sémantickou ztrátou (generovaný pseudokód je často obtížně srozumitelný, postrádá původní názvy, typové informace a abstrakce) [50], nízká přesnost u komplexního/optimalizovaného kódu (sémantická přesnost může být nízká, zejména u kódu po agresivních optimalizacích [44]), omezená schopnost deobfuskace (mohou být snadno zmateny sofistikovanými obfuskačními technikami).

Metody založené na AI/ML (zejména LLM): Tyto metody jsou založeny na učení se komplexních vzorů z rozsáhlých datových sad kódu.

- *Silné stránky:* Výrazně lepší čitelnost a srozumitelnost kódu (LLM jsou schopny generovat kód stylisticky bližší lidmi psanému kódu) [44], pokrok v obnově sémantických informací (názvy, typy, komentáře, sumarizace), schopnost učit se z obrovského množství kódu (mohou těžit z tréninku na miliardách řádků kódu), potenciál pro deobfuskaci (některé přístupy mohou pomoci s určitými typy obfuskace, např. dešifrování řetězců [63]).
- *Slabé stránky:* Nižší funkční korektnost/sémantická přesnost (aktuálně mají tendenci produkovat kód, který nemusí být vždy sémanticky ekvivalentní) [44], „halucinace“ (mohou generovat nesprávné informace) [63], citlivost na kvalitu trénovacích dat, výpočetní náročnost (trénink i inference mohou být náročné).

Budoucnost pravděpodobně spočívá v inteligentní kombinaci těchto přístupů, kde tradiční metody poskytnou robustní základ a AI pomůže se srozumitelností a obnovou sémantiky.

2.5.2 Výhody a Nevýhody Jednotlivých Přístupů v Kontextu C/C++

Specifika jazyků C a C++, jako jsou ukazatele, manuální správa paměti a agresivní kompilátorové optimalizace, představují výzvy pro všechny metody reverzního inženýrství. Násle-

dující body shrnují, jak se s těmito specifiky vyrovnávají různé přístupy.

- **Tradiční metody:** Mohou být přesnější při analýze nízkourovňových detailů, kde je důležitá bitová přesnost a porozumění interakcím s hardwarem nebo operačním systémem. Pokud jsou na to specificky naprogramovány, mohou lépe zvládat některé aspekty kompilátorově specifického chování nebo nízkourovňové triky běžné v C/C++.
- **AI/ML metody:** Mohou mít výhodu v učení se a rozpoznávání běžných C++ idiomů a programovacích vzorů z velkého množství kódu, což může vést k generování čitelnějšího a sémanticky bohatšího pseudokódu. Mohou také lépe zvládat variabilitu a „šum“ v reálném kódu. Jak ukazuje příklad `r2ai` [63], AI může pomoci i s některými typy deobfuskače.
- **MDRE:** Tento přístup je silný v abstrakci a vizualizaci celkové architektury softwarového systému. Může pomoci pochopit vztahy mezi komponentami a vysokoúrovňovou strukturu, ale nemusí poskytovat tak detailní vhled na úrovni jednotlivých funkcí nebo instrukcí jako specializované dekompilátory [42].
- **Fuzzing dekompilátorů (např. BIN2WRONG):** Ačkoliv se nejedná o metodu reverzního inženýrství samotnou, systematické testování a odhalování chyb v dekompilátorech vede ke zlepšení jejich kvality a spolehlivosti [7], což následně prospívá všem analytikům.

2.5.3 Analýza Nákladů a Přínosů

Ekonomické aspekty hrají důležitou roli při výběru přístupu k reverznímu inženýrství. Je třeba zvážit jak přímé náklady na nástroje, tak nepřímé náklady spojené s časem analytiků a potřebnou expertízou.

- **Tradiční nástroje:** Hlavním nákladem u komerčních nástrojů, jako je IDA Pro [75], jsou vysoké licenční poplatky. I u open-source nástrojů je třeba počítat s časem potřebným na jejich zvládnutí a konfiguraci. Klíčovým nákladem je však čas analytika strávený manuální analýzou a interpretací často neoptimálního dekompilovaného kódu. Přínosem jsou zavedené a relativně spolehlivé nástroje pro základní analýzu.

- **AI/ML nástroje/metody:** Vývoj a trénink vlastních velkých LLM modelů je extrémně nákladný. Používání komerčních LLM přes API může být také finančně náročné. Provoz lokálních LLM modelů vyžaduje výkonný hardware. Navíc je třeba počítat s časem na validaci výstupů AI. Přínosem je potenciál pro významnou úsporu času analytika díky lepší čitelnosti kódu a automatizaci obnovy sémantických informací [63].
- **Open-source vs. Komerční:** Použití open-source nástrojů jako Ghidra nebo Radare2 [46] výrazně snižuje přímé licenční náklady. Může však vyžadovat více času na úvodní nastavení, učení se jejich ovládání a případně na vývoj vlastních skriptů pro specifické potřeby.

2.5.4 Dopad na Časovou Náročnost a Potřebné Expertní Znalosti

Nástup AI/ML metod mění nejen nástroje, ale i požadavky na čas a dovednosti analytiků. Cílem není plná automatizace, ale spíše dosažení stavu „rozšířené inteligence”.

- **AI asistence a časová náročnost:** AI může potenciálně výrazně zkrátit čas potřebný pro některé fáze reverzního inženýrství, zejména pro úkoly jako je rychlé pochopení neznámého kódu, identifikace klíčových funkcí, nebo deobfuskace určitých typů dat [63].
- **Role lidské expertízy:** Navzdory pokrokům v AI zůstává lidská expertíza naprosto klíčová. AI nástroje, zejména LLM, vyžadují vedení a dohled zkušeného analytika, který dokáže formulovat správné dotazy (*prompts*), interpretovat výsledky a především kriticky hodnotit výstupy AI a detekovat potenciální chyby nebo „halucinace” [63].
- **Nové požadované dovednosti:** Efektivní využití AI v reverzním inženýrství vyžaduje od analytiků nové dovednosti. Patří sem základní porozumění principům fungování AI/ML modelů, schopnost efektivně interagovat s LLM (tzv. *prompt engineering*) a schopnost kriticky interpretovat jejich často probabilistické výstupy.
- **Tradiční dovednosti zůstávají nezbytné:** Znalost assembleru pro cílové architektury [38], principů fungování operačních systémů, volacích konvencí, formátů spustitelných souborů a technik používaných kompilátory zůstává absolutním základem pro každého reverzního inženýra.

Cílem je „rozšířená inteligence“ (*augmented intelligence*), kde AI slouží jako výkonný nástroj v rukou kvalifikovaného lidského experta.

2.6 Obecná Doporučení a Budoucí Směry v Reverzním Inženýrství C/C++

Reverzní inženýrství C/C++ aplikací je dynamicky se vyvíjející obor, který neustále čelí novým výzvám a příležitostem. Pro úspěšnou a efektivní analýzu je důležité dodržovat osvědčené postupy, být si vědom aktuálních výzev a sledovat budoucí trendy.

2.6.1 Doporučené Postupy pro Reverzní Analýzu Komplexních C/C++ Aplikací

Pro efektivní reverzní analýzu komplexních C/C++ aplikací je vhodné dodržovat několik osvědčených postupů, které kombinují systematický přístup s flexibilitou a využitím moderních nástrojů.

- Kombinovaný přístup (statická a dynamická analýza): Vždy je vhodné kombinovat statickou analýzu (zkoumání kódu bez spuštění) s dynamickou analýzou (sledování chování programu za běhu). Statická analýza poskytuje přehled o struktuře kódu, zatímco dynamická analýza může odhalit chování závislé na vstupech nebo runtime prostředí [65, 40].
- Iterativní a inkrementální proces: Analýza by měla probíhat iterativně. Začněte s vysokoúrovňovým přehledem (např. identifikace hlavních funkcí, importů, řetězců) a postupně se nořte do detailů konkrétních částí kódu, které jsou relevantní pro cíl analýzy [70]. Použijte dekompilátory pro generování počátečního pseudokódu a následně tento kód zpřesňujte.
- Využití skriptování pro automatizaci: Většina moderních RE nástrojů (IDA Pro, Ghidra, Binary Ninja, Radare2) nabízí výkonné skriptovací API (často v Pythonu). Využijte je k automatizaci repetitivních úkolů, jako je hromadné přejmenovávání, hledání specifických vzorů kódu, extrakce dat nebo deobfuskace [76, 77].
- Pečlivá dokumentace: Během analýzy si ved'te podrobné poznámky o svých zjištěních. Vytvářejte diagramy (např. CFG, DFG, diagramy volání funkcí), mapujte datové struktury a řídicí toky. Kvalitní dokumentace je klíčová pro pochopení komplexních systémů [65].

- **Bezpečné analytické prostředí:** Zejména při analýze potenciálně škodlivého softwaru (malware) je naprosto nezbytné používat izolované prostředí, jako jsou virtuální stroje (VM) nebo specializované sandboxy. Tím se zabrání infikování nebo poškození vašeho hlavního systému [65, 57].
- **Hluboké porozumění cílové platformě:** Znalost assembleru pro cílovou architekturu, principů fungování operačního systému, volacích konvencí, formátů spustitelných souborů a technik používaných kompilátory je naprosto zásadní.
- **Opatrné využití AI asistence s kritickým hodnocením:** Nástroje založené na LLM mohou být velmi užitečné pro generování hypotéz, vysvětlení neznámých částí kódu nebo obnovu názvů. Vždy však přistupujte k jejich výstupům kriticky a ověřujte je nezávislými metodami, protože AI modely mohou „halucinovat“ [63].

2.6.2 Strategie pro Překonávání Obfuskace a Anti-Reverzních Technik

Překonávání obfuskačních a anti-reverzních technik je často nejnáročnější a nejkreativnější částí RE [43, 40]. Jak bylo detailně rozebráno v předchozí kapitole, tyto techniky zahrnují šifrování kódu, vkládání zavádějících instrukcí, zamlžování řídicího toku, anti-debugging a virtualizaci kódu. Následující strategie, využívající metody a nástroje diskutované v této kapitole, mohou pomoci při jejich analýze.

- **Rozpoznání Typu Obfuskace:** Prvním krokem je identifikace použitých technik. Znalost běžných vzorů (např. typické struktury packerů, zploštělého CFG, použití neprůhledných predikátů) je klíčová. Nástroje jako PESTudio [78] nebo specializované detektory packerů mohou pomoci identifikovat některé typy ochran.
- **Deobfuskace:** Tento proces zahrnuje odstranění nebo zmírnění efektů obfuskačních transformací.
 - *Manuální deobfuskace:* Vyžaduje pečlivou analýzu obfuskovaného kódu a jeho postupné „rozplétání“ a přepisování do srozumitelnější podoby v nástrojích jako IDA Pro, Ghidra nebo x64dbg. To může být velmi časově náročné.
 - *Automatizovaná/Asistovaná deobfuskace:* Pro některé běžné typy obfuskace existují specializované nástroje nebo skripty.

- * Pro **unpacking**: Lze použít generické unpackery, nebo techniku krokování v debuggeru (např. x64dbg) k nalezení původního vstupního bodu (OEP) a následné dumpování paměti procesu [57].
 - * Pro **zamlžený řídicí tok**: Symbolická nebo konkolická exekuce (např. pomocí frameworku angr [5]) může pomoci odhalit skutečné cesty provádění kódu a zjednodušit komplexní predikáty. Nástroje jako SAILR [5] se snaží algoritmicky restrukturalizovat CFG.
 - * Pro **šifrované řetězce/data**: Dynamická analýza (sledování přístupů k paměti po dešifrování) nebo statická analýza (hledání dešifrovacích rutin) může odhalit původní data. AI nástroje jako r2ai [63] mohou asistovat při identifikaci šifrovacích algoritmů a klíčů, pokud jsou v kódu přítomny stopy.
- **Bypass Anti-Debugging/Anti-VM Technik**: To může zahrnovat modifikaci analytického prostředí tak, aby nebylo detekováno (např. úprava registrů, souborů nebo ovladačů ve VM), nebo patchování (úpravu) detekčních rutin přímo v analyzovaném binárním souboru tak, aby vždy vracely „bezpečný“ výsledek (např. pomocí x64dbg) [57, 43].
 - **Analýza Kódu v „Fancy” Jazycích nebo z Nestandardních Kompilátorů**: Pokud je software napsán v méně obvyklém jazyce (např. Rust, Go, Nim) nebo zkompilován nestandardním kompilátorem, je třeba se zaměřit na specifické artefakty, runtime chování a knihovny, které jsou pro daný jazyk/kompilátor typické [68]. Může být nutné se naučit základy daného jazyka nebo prostudovat dokumentaci kompilátoru.
 - **Analýza Virtualizovaného Kódu**: Překonání ochrany založených na virtualizaci kódu (např. VMProtect, Themida) je jednou z největších výzev. Vyžaduje nejprve reverzní inženýrství samotné virtuální mašiny (jejího interpretu a bytecode). Nástroje jako IDA Pro nebo Ghidra jsou nezbytné pro detailní analýzu handlerů VM a pochopení sémantiky jednotlivých bytecode instrukcí [40].

Překonávání těchto ochrany vyžaduje nejen hluboké technické znalosti, ale také kreativitu a trpělivost.

2.6.3 Význam Porozumění Moderním C++ Rysům (C++20/23) pro Reverzní Inženýrství

Jazyk C++ se neustále vyvíjí, přičemž nové standardy (C++11, C++14, C++17, C++20, C++23) [74] zavádějí pokročilé rysy, které mění způsob psaní kódu a následně i jeho binární reprezentaci. Porozumění těmto moderním rysům je pro efektivní reverzní inženýrství stále důležitější. Nové rysy jako korutiny, moduly, koncepty, ranges, *structured bindings* atd., přinášejí nové výzvy pro dekompilátory [65]. Jejich správná dekompilace a srozumitelná reprezentace jsou stále předmětem vývoje (viz např. CTF úloha zaměřená na C++17 rysy [66]). Analytik se znalostí moderního C++ bude lépe schopen rozpoznat tyto vzory v dekompilovaném kódu, pochopit původní záměry programátora a efektivněji analyzovat software napsaný s využitím těchto pokročilých rysů.

2.6.4 Očekávaný Vývoj a Trendy v Oblasti

Oblast reverzního inženýrství C/C++ je neustále v pohybu, poháněna jak pokroky v technologiích, tak snahou autorů softwaru chránit své výtvoř. Lze očekávat následující klíčové trendy.

- Pokročilejší AI/LLM modely: Očekává se další zlepšení sémantické přesnosti a schopnosti generovat spustitelný kód. Pravděpodobný je vývoj specializovaných LLM pro konkrétní architektury nebo typy softwaru, a také hlubší integrace LLM s formálními metodami pro ověřování správnosti.
- Vývoj a standardizace benchmarků: Pokračující vývoj komplexních a realistických benchmarků, jako je *DecompileBench* [44], bude klíčový pro měření pokroku a srovnávání různých přístupů.
- Vylepšené tradiční dekompilátory: I tradiční dekompilátory se budou nadále vyvíjet, s lepšími heuristikami pro rekonstrukci kódu, vylepšenou podporou pro nové rysy jazyka C++ a novější architektury, a pravděpodobně i s hlubší integrací AI pluginů pro specifické úkoly.
- Rostoucí role symbolické exekuce a formálních metod: Tyto pokročilé analytické techniky budou hrát stále větší roli v hloubkové analýze kódu, ověřování jeho vlastností a detekci zranitelností, potenciálně více integrované do standardních RE workflow [44].

- Zaměření na bezpečnost, spolehlivost a škálovatelnost AI v RE: Bude třeba řešit problémy spojené s „halucinacemi“ LLM, potenciálními bezpečnostními riziky a efektivním nasazením AI metod na analýzu rozsáhlých kódovýchází.
- Výzkum v dekompilaci dalších jazyků (např. Rust): Rostoucí popularita jazyků jako Rust, které sdílejí některé výzvy s C++ (nízkourovňová kontrola, optimalizace), vede k intenzivnímu výzkumu v oblasti jejich dekompilace [5]. Poznatky z této oblasti mohou být přenositelné a inspirovat nové přístupy i pro C++.

Budoucnost RE C/C++ bude pravděpodobně hybridní, kombinující sílu a flexibilitu umělé inteligence s robustností a přesností tradičních metod a pokročilých analytických technik. Klíčem k úspěchu bude efektivní interakce mezi člověkem-analytikem a stále chytřejšími nástroji.

2.6.5 Doporučení pro Další Výzkum a Praktické Aplikace

Pro další pokrok v oblasti reverzního inženýrství C/C++ je třeba se zaměřit na několik klíčových oblastí výzkumu a praktických aplikací, které pomohou překonat stávající omezení a plně využít potenciál nových technologií.

Doporučení pro další výzkum: Vědecká komunita by se měla soustředit na následující oblasti.

- Zlepšení sémantické přesnosti LLM dekompilátorů: Vývoj modelů a trénovacích strategií, které povedou k generování kódu s vyšší funkční korektností a spolehlivostí.
- Vývoj technik pro dekompilaci moderních C++20/23 rysů: Zaměření na spolehlivou rekonstrukci korutin, modulů, konceptů a dalších pokročilých jazykových konstrukcí do srozumitelné podoby.
- Robustní metody pro zvládání obfuskace s pomocí AI: Výzkum, jak mohou AI/ML techniky pomoci automatizovat nebo asistovat při deobfuskaci a překonávání sofistikovaných anti-reverzních ochran.
- Škálovatelné a spolehlivé benchmarky: Pokračování ve vývoji a standardizaci benchmarků, které umožní objektivní hodnocení a srovnání různých dekompilačních přístupů na reálném a moderním softwaru.

- Vysvětlitelná AI (*Explainable Artificial Intelligence* (XAI)) pro dekompilaci: Výzkum metod, které by umožnily lépe pochopit, jak LLM dospívají ke svým výsledkům při dekompilaci, což by zvýšilo důvěru v jejich výstupy a pomohlo identifikovat jejich slabiny.

Doporučení pro praktické aplikace: Pro efektivní využití současných i budoucích nástrojů v praxi je vhodné zvážit následující.

- Integrace nejnovějších AI nástrojů do stávajících RE workflow: Postupné začleňování AI asistentů a LLM-založených nástrojů do každodenní praxe reverzních inženýrů, s důrazem na oblasti, kde mohou přinést největší přidanou hodnotu (např. zlepšení čitelnosti, anotace).
- Školení analytiků pro efektivní používání AI nástrojů: Poskytování školení a vzdělávacích materiálů pro analytiku, aby se naučili efektivně používat nové AI nástroje, správně formulovat dotazy (*prompts*) a kriticky interpretovat jejich výstupy.
- Vývoj interních *best practices*: Organizace by měly vyvíjet vlastní osvědčené postupy pro kombinování tradičních a AI-založených metod reverzního inženýrství, přizpůsobené jejich specifickým potřebám a typům analyzovaného softwaru.
- Zaměření na specifické typy aplikací (firmware, IoT, malware): Aplikace RE technik na kritické oblasti, jako je bezpečnost firmwaru, IoT zařízení, nebo analýza sofistikovaného malware, kde je hluboké porozumění kódu zásadní.

Je důležité si uvědomit etický rozměr a potenciální „dual-use” povahu pokročilých technik RE. Rychlost změn v této oblasti vyžaduje neustálé vzdělávání [74]. Budoucnost RE C/C++ směřuje k hybridní inteligenci, kde člověk a AI spolupracují [63].

Tabulka 2.1 Srovnání Hlavních Dekompilačních Nástrojů pro C/C++ (se zaměřením na získávání pseudokódu)

Vlastnost	IDA Pro (Hex-Rays)	Ghidra	Binary Ninja	Radare2 / Rizin (s Cutter a Ghidra pluginem)
Licence	Komerční (velmi drahá)	Open-Source (Apache 2.0)	Komerční (různé úrovně)	Open-Source (LGPLv3/GNU General Public License (GPL))
Kvalita C-like pseudokódu	Velmi vysoká, často nejlepší čitelnost a přesnost pro C/C++	Dobrá až velmi dobrá, srovnatelná s komerčními v mnoha případech	Dobrá, HLIL je dobře čitelný, aktivně se zlepšuje	Dobrá až velmi dobrá (při použití Ghidra dekompilátoru)
Podpora C++ konstrukcí	Vynikající (třídy, vtables, demangling, výjimky pro MSVC)	Dobrá (třídy, demangling, typy), může vyžadovat manuální práci	Dobrá, aktivně se vyvíjí, BNIL umožňuje detailní analýzu	Dobrá (při použití Ghidra), závisí na analýze r2/Rizin
Podpora moderních C++ rysů	Omezená, závisí na verzi a konkrétním rysu	Omezená, závisí na verzi a konkrétním rysu	Aktivně se pracuje na zlepšení, ale stále výzva	Omezená, závisí na Ghidra dekompilátoru
Zvládání optimalizací	Dobré, ale přesnost klesá s agresivními optimalizacemi	Slušné, ale může mít problémy s některými optimalizacemi	Neustále se zlepšuje, BNIL pomáhá	Slušné (přes Ghidra), závisí na robustnosti Ghidra
Sémantická přesnost (odhad)	~55% (může klesat) [50]	Srovnatelná s IDA v mnoha případech, ale může kolísat	Aktivně se zlepšuje, cílem je vysoká přesnost	Srovnatelná s Ghidra (při použití pluginu)
Znovuspuštění kódu	Zřídka přímo	Zřídka přímo, vyžaduje úpravy	Zřídka přímo, cílem je zlepšení	Zřídka přímo (přes Ghidra)
AI Integrace (pro zlepšení kódu)	Pluginy (např. IDA-Assistant) pomáhají s porozuměním [63]	Pluginy (např. GptHidra) pomáhají s porozuměním [63]	Nativní (Sidekick [64]) a pluginy pro asistenci	Plugin r2ai velmi flexibilní pro vysvětlení, přejmenování [63]
Síla „získání kódu“	Nejpokročilejší dekompile, podpora C++	Výkonný bezplatný dekompilátor, rozšiřitelnost (Sleigh)	Moderní přístup, silné API, víceúrovňové IL	Flexib., skriptovatelnost, přístup k Ghidra dekompilaci zdarma

(zdroj: vlastní)

Tabulka 2.2 Přehled Metrik pro Hodnocení Efektivity Dekompilace a Jejich Aplikace

Metrika	Popis Metriky	Typické Nástroje/Metody pro Měření	Význam/Aplikace v RE
Sémantická Konzistence / Přesnost	Zda dekompilovaný kód funkčně odpovídá původnímu binárnímu kódu.	Diferenciální testování, symbolická exekuce, runtime validace (DecompileBench [71]).	Kritická pro bezpečnostní analýzu, analýzu malware.
Čitelnost / Srozumitelnost	Jak snadno může člověk-analytik pochopit dekompilovaný kód.	Subjektivní hodnocení, metriky složitosti, <i>LLM-as-a-Judge</i> (DecompileBench [71]).	Zásadní pro rychlost a efektivitu analýzy.
Znovuspustitelnost / Znovukompilovatelnost	Zda lze dekompilovaný kód znovu zkompileovat a spustit.	Pokus o kompilaci a spuštění, automatizované testovací frameworky (např. HumanEval).	Umožňuje dynamickou analýzu, modifikaci kódu.
Blízkost k Původnímu Zdrojovému Kódu	Měření rozdílu mezi dekompilovaným kódem a původním zdrojovým kódem (pokud je k dispozici).	Porovnání AST, CFGED (SAILR [5]), textové porovnání.	Užitečné pro výzkum dekompilátorů.
<i>Control Flow Clarity</i> (DecompileBench)	Jak jasně a přesně je reprezentován řídicí tok programu.	<i>LLM-as-a-Judge</i> , manuální inspekce CFG [44].	Důležité pro pochopení logiky programu.
<i>Memory Layout Accuracy</i> (DecompileBench)	Jak přesně dekompilovaný kód odráží rozložení datových struktur v paměti.	<i>LLM-as-a-Judge</i> , manuální analýza [44].	Klíčové pro analýzu datových struktur, paměťových zranitelností.

(zdroj: vlastní)

II PRAKTICKÁ ČÁST

3 METODIKA PRAKTICKÉ ČÁSTI

Tato kapitola definuje metodologický rámec uplatněný v praktické části diplomové práce. Jejím cílem je transparentně a systematicky popsat výběr konkrétních softwarových aplikací určených k analýze, specifikovat použité metody a nástroje reverzního inženýrství (RE), podrobně charakterizovat experimentální uspořádání včetně nastavení analytického prostředí a obecný postup analýzy. Dále jsou definována kritéria pro hodnocení získaných výsledků. Tento popis slouží jako základ pro prezentaci výsledků v následujících kapitolách a umožňuje formulaci obecných doporučení. Důraz je kladen na zajištění reprodukovatelnosti všech provedených experimentů.

3.1 Výběr Aplikací pro Analýzu

Pro účely praktické části byly pečlivě vybrány tři různorodé softwarové aplikace. Tento výběr byl záměrný, aby reprezentoval odlišné problémy a výzvy v oblasti reverzního inženýrství C/C++ aplikací a umožnil tak demonstraci širokého spektra analytických technik a nástrojů. Každá aplikace byla zvolena tak, aby přispěla k naplnění cílů práce specifickým způsobem. Přehled aplikací, jejich stručná charakteristika a hlavní metodologické cíle jsou uvedeny v Tabulce 3.1.

3.1.1 Aplikace A: bbbbloat (CTF Úloha)

Typ a charakteristika: Konzolová aplikace¹⁾ pro Linux (64bit ELF), typická CTF úloha. Program ověřuje zadané číslo proti interní hodnotě a v případě shody odhalí "flag". Nahradila původně zvažovanou analýzu `calc.exe` pro lépe ohraničenou výzvu. **Zdroj a licenční aspekty:** Veřejně dostupný archiv CTF soutěží, určený k volné analýze pro vzdělávací účely. **Metodologický cíl:** Demonstrovat základní postupy statické a dynamické analýzy na jednoduché linuxové aplikaci bez ladicích symbolů. Identifikovat ověřovací mechanismus, odhalit algoritmus pro získání "flagu" a oddělit relevantní kód od balastního. Slouží jako úvodní příklad a referenční bod pro srovnání náročnosti.

¹⁾<https://github.com/akshayrkapadia/CTF/tree/main/PicoCTF/ReverseEngineering/Bbbbloat>

Tabulka 3.1 Přehled Analyzovaných Aplikací a Jejich Metodologických Cílů

Označení (dále pouze)	Název	Typ, Charakteristika a Zdroj (stručně)	Metodologický Cíl
A	bbbbloat	Linux ELF (64bit), CTF úloha (ověření čísla pro získání "flagu"). Zdroj: Veřejné CTF archivy. Kompaktní, algoritmicky orientovaný problém.	Demonstrovat základní statickou a dynamickou analýzu na jednoduchém ELF souboru bez ladicích symbolů, identifikovat klíčový ověřovací algoritmus.
B	Shellcode	x86 shellcode (Windows), pozicně nezávislý kód (Position Independent Code (Pozicně nezávislý kód) (PIC)), XOR obfuskace payloadu, dynamické získávání adres API funkcí pomocí API hashingu. Zdroj: GitHub (<i>learning-malware-analysis</i>), XOR klíč 0x97.	Demonstrovat analýzu obfuskáčnických a deobfuskáčnických technik, porozumět dynamickému API resoluování bez IAT, odhalit funkcionální skrytého payloadu.
C	GeometryMaster	C++ konzolová aplikace demonstrující Object-Oriented Programming (Objektově orientované programování) (OOP) principy (třídy, virtuální metody, Standard Template Library (Standardní knihovna šablon) (STL)). Zdroj: Poskytnutý kód 0_GeometryMaster.cpp. Kompilována GNU Compiler Collection (Kolekce kompilátorů GNU) (GCC) v 6 variantách (V00-V05) s různými nastaveními (optimalizace, symboly, RTTI). Příklad dekompilátu V05 funkce main ilustruje dopad silné optimalizace.	Systematicky analyzovat a porovnat vliv různých kompilačních nastavení na čitelnost, strukturu a celkovou analyzovatelnost dekompilovaného C++ kódu, zejména schopnost rekonstruovat OOP struktury.

(zdroj: vlastní)

3.1.2 Aplikace B: Analyzovaný Shellcode

Typ a charakteristika: Ukázka²⁾ nízkourovňového kódu, x86 shellcode pro Windows (PIC). Využívá XOR šifrování payloadu a dynamické získávání adres API funkcí pomocí API hashingu (obchází IAT). **Zdroj a licenční aspekty:** Založen na ukázce z repozitáře *learning-malware-analysis* na GitHubu. Analyzovány soubory `shell_code_readme.txt` (obsahující popis a XOR klíč 0x97), `assembly_obfuscated.txt` a `assembly_deobfuscated.txt`. Pro dynamickou analýzu byl z hexadecimální reprezentace vytvořen binární soubor `shellcode.bin` a spuštěn pomocí jednoduchého C zavaděče. **Metodologický cíl:** Demonstrovat a porozumět obfuskačním/deobfuskačním technikám. Analyzovat mechanismus dynamického resolvování API funkcí, identifikovat cílové API funkce na základě hashů a odvodit funkcionalitu payloadu (např. spuštění `cmd.exe`).

3.1.3 Aplikace C: GeometryMaster (Ukázková C++ Aplikace)

Typ a charakteristika: Ukázková desktopová³⁾ (konzolová) C++ aplikace demonstrující OOP (třídy, virtuální metody, dynamická alokace, STL). **Zdroj a varianty:** Zdrojový kód `0_GeometryMaster.cpp` (cca 3kB) a `Readme.txt`. Zkompilován GCC (Linux) v 6 variantách (V00-V05) s různými nastaveními: úroveň optimalizace (-O0 až -O3, LTO), (ne)přítomnost ladicích symbolů (-s), cílová architektura (-m32/-m64), (ne)podpora C++ výjimek a RTTI. Příklady variant:

- V00 (-m32 -g -O0): 32bit, debug symboly, bez optimalizací (109kB).
- V03 (-m32 -O3 -s): 32bit, optimalizace na velikost, stripovaná (18kB).
- V05 (-m64 -O3 -s -flto): 64bit, optimalizace -O3, stripovaná, LTO (19kB).

Metodologický cíl: Systematicky analyzovat a porovnat vliv různých kompilačních nastavení na čitelnost, strukturu a analyzovatelnost dekompilovaného C++ kódu. Důraz na rekonstrukci OOP struktur (třídy, datové členy, vtables) v Ghidře.

²⁾<https://github.com/jstrosch/learning-malware-analysis/tree/master/Shellcode>

³⁾vlastní tvorba

3.2 Použité Metody a Nástroje Reverzního Inženýrství

Pro analýzu byla použita kombinace statických a dynamických metod RE, jejichž výběr byl informován teoretickou částí práce.

3.2.1 Přehled Použitých Nástrojů

Tabulka 3.2 uvádí konkrétní nástroje, jejich verze a primární účel v této práci. Verze odpovídaly stabilním verzím dostupným v době provádění analýzy.

Tabulka 3.2 Přehled nástrojů RE použitých v praktické části.

Nástroj	Verze	Platforma	Primární Účel v Těto Práci
Ghidra	11.0.3	Multiplatformní	Statická analýza, dekompilace (všechny aplikace)
Radare2	5.9.0	Multiplatformní	Doplňková statická analýza (Shellcode, GeometryMaster)
GNU Binutils (objdump)	2.41	Linux/Multi.	Inspekce ELF (bbbbloat, GeometryMaster Linux)
x64dbg / x32dbg	Stabilní (IV/2024)	Windows	Dynamická analýza Windows (Shellcode loader)
GDB (GNU Debugger)	14.1	Linux/Multi.	Dynamická analýza Linux (všechny aplikace na Linuxu)
GEF (GDB Enhanced Feat.)	Stabilní (I/2024)	Linux	Rozšíření GDB
ltrace / strace	Stabilní (dle distribuce)	Linux	Trasování knihoven/syst. volání (bbbbloat, experimentálně)
Detect It Easy (DIE)	3.08	Win/Linux	Identifikace souborů (bbbbloat, Shellcode)
HxD (Hex Editor)	2.5.0.0	Windows	Inspekce/modifikace binárních dat
Python	3.10.12	Multiplatformní	Pomocné skripty (např. deobfuskace)
VMware® Workstation 17 Pro	17.x	Multiplatformní	Izolovaná analytická prostředí
GCC (g++)	11.4.0	Linux	Kompilace variant GeometryMaster

(zdroj: vlastní)

Zdůvodnění výběru hlavních nástrojů: Ghidra byla zvolena jako primární nástroj pro

statickou analýzu a dekompilaci díky své komplexnosti, kvalitní podpoře C/C++ kódu a dostupnosti zdarma. **Radare2** posloužil pro doplňkovou nízkoúrovňovou analýzu a rychlý průzkum. Pro dynamickou analýzu byly vybrány debuggery **x64dbg/x32dbg** pro prostředí Windows (specificky pro analýzu C zavaděče shellcode) a **GDB s rozšířením GEF** pro prostředí Linux. Tato sada nástrojů komplexně pokrývá potřeby analýzy definované pro praktickou část práce.

3.2.2 Aplikované Analytické Metody

V praktické části byly aplikovány standardní metody RE, jejichž detailní teoretický popis je obsažen v předchozích kapitolách.

- **Statická analýza:** Představovala primární metodu a zahrnovala disasemblaci, dekompilaci (převážně pomocí Ghidry), analýzu grafu řídicího toku (CFG) a grafu datového toku (DFG), identifikaci funkcí a datových struktur, demangling symbolů (pokud byly přítomny), analýzu importovaných a exportovaných funkcí a extrakci řetězcových literálů.
- **Dynamická analýza:** Sloužila především k ověření hypotéz formulovaných během statické analýzy a ke sledování skutečného chování programu za běhu. Zahrnovala použití breakpointů, krokování kódu (instrukci po instrukci nebo na úrovni zdrojového kódu, pokud byly dostupné symboly), inspekci stavu procesorových registrů a obsahu paměti a sledování systémových či knihovních volání (např. pomocí nástrojů strace, ltrace).
- **Specifické techniky pro C++ (aplikované na Aplikaci C):** Vzhledem k povaze aplikace GeometryMaster byly použity techniky zaměřené na analýzu C++ konstrukcí. Tyto zahrnovaly analýzu tabulek virtuálních metod (vtables), identifikaci volání virtuálních metod (zejména v situacích bez dostupných RTTI), pokusy o rekonstrukci tříd a jejich datových členů, analýzu manglovaných (dekorovaných) jmen funkcí a metod a sledování alokace a uvolňování dynamické paměti.

3.2.3 Experimentální Využití Metod Umělé Inteligence

Metody umělé inteligence (AI) a strojového učení (ML) byly v rámci praktické části této práce aplikovány pouze experimentálně a ve velmi omezené míře. Sloužily jako doplňková

technika a hlavní důraz zůstal na zavedených manuálních a poloautomatických technikách RE. Konkrétně šlo o orientační průzkum možností některých dostupných AI nástrojů (například pluginů pro RE platformy jako Radare2⁴⁾ s r2ai⁵⁾ nebo Ghidra s GhidrAssist⁶⁾, případně cloudových API jako Claude 3 Sonnet) pro dílčí úkoly, jako je automatická sumarizace funkcionality segmentů kódu nebo návrh potenciálně smysluplnějších názvů pro proměnné a funkce. Tyto experimenty byly provedeny zejména u Aplikace B (Analyzovaný Shellcode) a Aplikace C (GeometryMaster). Stručné zmínky o výsledcích těchto omezených experimentů jsou uvedeny v příslušných sekcích Kapitoly 4.

3.3 Experimentální Uspořádání a Postup Analýzy

Tato sekce popisuje detaily analytického prostředí a obecný metodologický postup, který byl uplatněn při analýze každé z vybraných aplikací.

3.3.1 Analytické Prostředí

Pro zajištění kontrolovaného, bezpečného a reprodukovatelného prostředí pro RE bylo využito virtualizační prostředí. Tento přístup je klíčový zejména při analýze potenciálně škodlivého kódu (Aplikace B).

- **Virtualizační software:** VMware® Workstation 17 Pro.
- **Hostitelský operační systém:** Windows 10
- **Hostující operační systémy (Virtuální stroje):**
 - Windows 10 Professional x64 aktuální (česká verze) – pro analýzu artefaktů specifických pro Windows (C zavaděč pro Aplikaci B).
 - Kali linux ⁷⁾. 2025.1c – pro analýzu linuxových cílů (Aplikace A, všechny varianty Aplikace C).
- **Konfigurace VM:** Každý VM s minimálně 4 GB RAM, 2 vCPU a virtuálním diskem alespoň 60 GB. Nainstalovány VMware Tools.

⁴⁾<https://github.com/radareorg/radare2>

⁵⁾<https://github.com/radareorg/r2ai>

⁶⁾<https://github.com/jtang613/GhidrAssist>

⁷⁾<https://www.kali.org/get-kali/#kali-platforms>

- **Sít'ová izolace a bezpečnostní opatření:** Při analýze Aplikace B byl sít'ový adaptér VM nastaven do režimu "Host-only Adapter" nebo deaktivován. Pravidelně byly vytvářeny snapshoty. Analytické nástroje instalovány přímo do VM.

3.3.2 Obecný Postup Analýzy pro Každou Aplikaci

Pro analýzu každé aplikace byl uplatněn systematický a iterativní postup, flexibilně přizpůsobený specifikům.

1. **Přípravná Fáze:** Shromáždění souborů, ověření integrity, základní identifikace (typ souboru, architektura, OS, detekce packerů), základní behaviorální analýza v izolovaném prostředí.
2. **Fáze Statické Analýzy:** Import do primárního nástroje (Ghidra), automatická analýza. Manuální průzkum disasemblovaného kódu a dekompilovaného pseudokódu. Identifikace klíčových funkcí a datových struktur. Systematická anotace kódu (přejmenovávání, komentáře). Analýza CFG. U C++ aplikací (Apl. C) pokus o rekonstrukci OOP konstrukcí. Doplnková analýza (Radare2, objdump).
3. **Fáze Dynamické Analýzy:** Spuštění v debuggeru (x32dbg/x64dbg, GDB+GEF). Nastavení breakpointů na relevantních místech. Krokování kódu, sledování hodnot v registrech a paměti. Ověřování hypotéz. Monitorování interakcí s OS (strace/ltrace).
4. **Iterativní Zpřesňování a Validace:** Plynulé přepínání mezi statickou a dynamickou analýzou, křížová validace zjištění.
5. **Experimentální Využití AI/ML Asistence (Apl. B, C):** Stručné experimentální použití vybraných AI nástrojů pro dílčí úkoly.
6. **Dokumentace, Vyhodnocení a Reporting:** Průběžná dokumentace, ukládání artefaktů, vyhodnocení výsledků dle kritérií (sekce 3.4).

3.4 Kritéria Hodnocení a Metriky Úspěšnosti

Pro systematické vyhodnocení výsledků RE (Kapitola 4) a srovnání metod/nástrojů (Kapitola 5) byla definována sada kritérií.

- **Kvalita dekompilovaného/rekonstruovaného kódu:**

- *Čitelnost a srozumitelnost (škála 1-5)*: Snadnost porozumění logice a struktuře pseudokódu.
 - *Přesnost a sémantická ekvivalence*: Funkční shoda dekompilátu s původní funkcionalitou.
 - *Blížkost k původním C/C++ konstrukcím*: Věrnost rekonstrukce C/C++ idiomů, datových struktur.
 - *Úspěšnost obnovy sémantických informací*: Schopnost obnovit/navrhnout smysluplné názvy.
- **Efektivita analytického procesu:**
 - *Časová náročnost (kvalitativně: nízká, střední, vysoká)*: Celkový čas potřebný k dosažení cílů.
 - *Úroveň potřebné manuální intervence*: Množství a náročnost manuální práce.
 - *Přínos kombinace metod*: Efektivita synergie statické a dynamické analýzy, přínos AI.
 - *Jednoduchost použití nástroje (User Experience (Uživatelský prožitek / Uživatelská přívětivost) (UX), škála 1-5)*: Intuitivnost, křivka učení, stabilita.
 - **Schopnost nástrojů a metod zvládat specifické výzvy RE:**
 - *Zvládání optimalizací překladačem (relevantní pro Apl. C)*: Jak si poradily dekompilátory.
 - *Rekonstrukce C++ rysů bez RTTI (relevantní pro Apl. C)*: Schopnost identifikovat OOP struktury.
 - *Odolnost vůči obfuskaci (relevantní pro Apl. B)*: Efektivita metod/nástrojů.
 - *Podpora pro cílovou platformu a formát souboru*: Adekvátnost podpory.
 - *Možnosti skriptování a automatizace*: Přínos pro efektivitu.

Výsledky hodnocení budou systematicky zaznamenány a použity pro srovnávací analýzu a formulaci doporučení.

3.5 Zajištění Reprodukovatelnosti a Etické Aspekty

Důraz je kladen na transparentnost pro možnost ověření a reprodukce výsledků a na důsledné dodržování etických principů. **Kroky pro podporu reprodukovatelnosti:**

- Detailní specifikace nástrojů a prostředí (sekce 3.2.1 a 3.3.1).
- Jednoznačná identifikace analyzovaných souborů (názvy, kompilační příznaky pro Apl. C, původ pro Apl. A, B).
- Systematický popis analytických postupů (sekce 3.3.2, detaily v Kapitole 4).
- Dostupnost materiálů (citace veřejných zdrojů, zdrojový kód pro Apl. C).

Etické aspekty provádění analýzy:

- Výběr aplikací: Výhradně open-source, studijní (CTF), veřejně dostupné příklady (shellcode) nebo vlastní ukázkové kódy (GeometryMaster). Žádný neoprávněný RE komerčního softwaru.
- Analýza potenciálně škodlivého kódu (Shellcode): Výhradně v přísně izolovaném virtuálním prostředí s deaktivovaným síťovým připojením.
- Účel analýzy: Výhradně pro akademické a výzkumné účely, s cílem studia a demonstrace metod a technik RE.

Dodržování těchto principů je nezbytným předpokladem pro zodpovědný výzkum v oblasti RE.

4 APLIKACE METOD, ANALÝZA VÝSLEDKŮ A VYHODNOCENÍ

Tato kapitola detailně prezentuje proces reverzního inženýrství aplikovaného na každou z vybraných softwarových aplikací. Pro každou aplikaci je popsán konkrétní postup analýzy, použité nástroje, klíčová zjištění a vyhodnocení výsledků na základě kritérií definovaných v Kapitole 3.

4.1 Analýza Aplikace A: bbbloat

Analýza programu bbbloat (CTF úloha) se zaměřila na pochopení jeho základní architektury, zpracování uživatelských vstupů a implementaci ověřovací logiky vedoucí k odhalení "flagu".

4.1.1 Prvotní Průzkum a Statická Analýza

Po získání binárního souboru bbbloat byla provedena základní identifikace: nástroj `file` potvrdil 64bitový ELF, `Detect It Easy` (DIE) nenalezl packery. Analýza řetězců (např. v Ghidře) odhalila texty jako "What's my favorite number?" a "Sorry, that's not it!", naznačující interakci a ověření.

Hlavní statická analýza proběhla v **Ghidře**. Po importu a automatické analýze byla identifikována funkce `main`, která volala další klíčovou funkci, Ghidrou pojmenovanou např. `FUN_00101307`. Detailní analýza této funkce (viz poskytnutý dekompilovaný kód) odhalila:

- Výpis výzvy "What's my favorite number?" pomocí `printf`.
- Načtení celočíselného vstupu od uživatele pomocí `__isoc99_scanf` do lokální proměnné `local_48`.
- Klíčové porovnání: `if (local_48 == 0x86187)`. Hodnota `0x86187` (dek. 549255) je hledané číslo.
- Pokud je podmínka splněna, volá se funkce `FUN_00101249`, která z interně uložených částí (v proměnných `local_38 = 0x4c75257240343a41;`, `local_30 = 0x3062396630664634;` atd.) sestaví a pomocí `fputs` a `putchar` vytiskne flag: `picoCTF{cu7_7h3_b1047_44f74a60}`.

Funkce FUN_00101249 provádí jednoduchou transformaci (posun znaků) na těchto částech před jejich spojením.

- Jinak se pomocí puts vytiskne "Sorry, that's not it!".

Nástroj **objdump** byl použit doplňkově pro rychlou inspekci disassemblovaného kódu.

4.1.2 Dynamická Analýza

Dynamická analýza pomocí **GDB** s rozšířením **GEF** potvrdila zjištění ze statické analýzy.

- Byly nastaveny breakpointy na začátek FUN_00101307 a na instrukci porovnání local_48 == 0x86187.
- Při zadání vstupu 549255 byla podmínka vyhodnocena jako pravdivá a program vytiskl flag, což odpovídalo výstupu z přímého spuštění.
- Při jiném vstupu byla vytištěna zamítavá zpráva.
- Hodnota proměnné local_48 byla po volání __isoc99_scanf úspěšně inspekceována v GDB.

Nástroje **strace** a **ltrace** (jejichž výstupy byly poskytnuty) ukázaly očekávaná systémová volání jako execve, brk, mmap, access, openat, fstat, read, write, close, arch_prctl, set_tid_address, set_robust_list, rseq, mprotect, prlimit64, munmap, getrandom, exit_group a knihovní volání jako printf, __isoc99_scanf, puts, bez neočekávané aktivity.

4.1.3 Výsledky Experimentální AI Asistence

Pro funkci FUN_00101307 byl dekompilovaný výstup z Ghidry stručně experimentálně analyzován pomocí AI nástroje (plugin GhidrAssist). Model poskytl obecný popis funkce ("ověřuje vstup a na základě toho tiskne výsledek") a částečně relevantní návrhy názvů proměnných (např. userInput pro local_48), avšak klíčové detaily jako konkrétní hodnota pro porovnání nebo mechanismus sestavení flagu musel analytik identifikovat manuálně.

4.1.4 Vyhodnocení Analýzy Aplikace A

- **Kvalita rekonstruovaného kódu (Ghidra):** Velmi dobrá (4/5). Dekompilovaný kód klíčové logiky byl snadno čitelný a srozumitelný.
- **Efektivita analytického procesu:** Celková časová náročnost byla nízká. Manuální intervence byla minimální, spočívala hlavně v interpretaci dekompilátu.
- **Splnění cílů analýzy:** Cíle byly plně splněny. Byl identifikován ověřovací mechanismus, odhalena hodnota pro úspěšné ověření a získán flag.
- **Přínos nástrojů:** Ghidra pro statickou analýzu a GDB+GEF pro dynamickou analýzu byly plně dostačující. strace a ltrace poskytly doplňující kontext.

4.2 Analýza Aplikace B: Analyzovaný Shellcode

Analýza binárního souboru `shellcode.bin` se zaměřila na identifikaci obfuskačních technik, deobfuskaci payloadu a pochopení jeho finální funkcionality. Pro dynamickou analýzu byl použit C zavaděč.

4.2.1 Prvotní Průzkum a Statická Analýza

Soubor `shellcode.bin` byl nástrojem DIE identifikován jako surová binární data. Analýza entropie naznačovala šifrování. V nástroji **Ghidra** (x86 architektura), na základě poskytnutého disasemblovaného kódu:

- **XOR Dešifrovací Smyčka:** Funkce `FUN_00000002` implementuje XOR dešifrovací smyčku. Po `POP EAX` (získání adresy následujícího bloku) a inicializaci čítače `ECX` na `0x186`, smyčka (`LAB_00000008`) dekrementuje `ECX` a provádí `XOR byte ptr DS:[EAX + ECX*1], 0x97`. Tento proces obnovuje původní instrukce payloadu. Klíč `0x97` byl potvrzen z `shell_code_readme.txt`.
- **API Hashing Rutina:** Po dešifrování (volání `CALL EAX` na konci `FUN_00000002` předá řízení na dešifrovaný kód) následuje komplexnější rutina pro dynamické získávání adres API funkcí. Funkce jako `FUN_000000af` a `FUN_000000e5` jsou zodpovědné za tento proces. `FUN_000000af` typicky iteruje přes načtené moduly (získané z `Process`

Environment Block (Blok prostředí procesu) (PEB) přes FS : [0x30]) a pro každou exportovanou funkci počítá hash (zahrnující bitové rotace ROL EDX, 0x7 a XOR DL, AL). Tento hash je pak porovnán s předdefinovanými hodnotami (např. uloženými v EBX). FUN_000000e5 se zdá být součástí tohoto mechanismu, pravděpodobně pro vyhledávání konkrétních funkcí na základě hashů a ukládání jejich adres (např. do ES:EDI pomocí STOSD).

Nástroj **Radare2** byl použit doplňkově pro vizualizaci grafů řídicího toku těchto rutin. **Identifikovaný deobfuskovaný payload** (na základě analýzy hashů a typických vzorů v dešifrovaném kódu, jak naznačuje FUN_00000016 a navazující volání):

- Hashe jako 0x4b1ffe8e (pro LoadLibraryA) a 0xd612ffcd (pro GetProcAddress) jsou použity k nalezení těchto klíčových funkcí.
- Pomocí těchto funkcí shellcode dynamicky získal adresy dalších funkcí, konkrétně CreateProcessA (pro spuštění nového procesu, viz volání s argumentem [EDX + 0x157], což je řetězec "cmd") a MessageBoxA (pro zobrazení zprávy, viz řetězec [EDX + 0x163] "Hello from shellcode!").
- Finální akcí bylo spuštění cmd.exe pomocí CreateProcessA, případně po zobrazení zprávy.

4.2.2 Dynamická Analýza

Dynamická analýza probíhala ve Windows 7 VM pomocí **x32dbg**, kam byl shellcode načten C zavaděčem.

- Breakpoint na začátku XOR dešifrovací smyčky (FUN_00000002): krokováním bylo pozorováno, jak se data v paměti (adresovaná EAX) mění na validní x86 instrukce.
- Breakpointy v API hashing rutině (např. v FUN_000000af na CMP EDX, EBX): sledováno, jak shellcode iteruje DLL a jejich exporty, počítá hashe a porovnává je. Potvrzeno mapování hashů na API funkce.
- Pozorováno volání LoadLibraryA("user32.dll"), následně GetProcAddress pro MessageBoxA a CreateProcessA.

- Potvrzeno zobrazení okna se zprávou "Hello from shellcode!" a následné spuštění procesu `cmd.exe`.

4.2.3 Výsledky Experimentální AI Asistence

Pro deobfuskované části shellcode byl stručně experimentálně použit plugin **r2ai** v Radare2. AI pomohla s generováním obecných popisů krátkých bloků kódu (např. identifikace smyčky jako "iterace přes data") a navrhla částečně relevantní názvy pro registry (např. `key_register` pro registr držící XOR klíč). Po manuálním zadání XOR klíče `0x97` a nařazení účelu byla schopna navrhnout Python skript pro deobfuskaci.

4.2.4 Vyhodnocení Analýzy Aplikace B

- **Kvalita rekonstruovaného kódu:** Po manuální deobfuskaci a anotaci byl dekompilovaný kód (API resolver) v Ghidře středně srozumitelný (3/5). Původní obfuskovaný kód byl téměř nečitelný (1/5).
- **Efektivita analytického procesu:** Časová náročnost střední až vysoká. Manuální intervence byla vysoká, zejména při interpretaci API hashingu a přípravě C zavaděče.
- **Splnění cílů analýzy:** Cíle plně splněny (identifikace obfuskace, deobfuskace payloadu, odhalení funkcionality).
- **Přínos nástrojů:** Ghidra (stat.), x32dbg (dyn.), Radare2 (vizual.), Python (skripty).

4.3 Analýza Aplikace C: GeometryMaster (Ukázková C++ Aplikace)

Analýza různých kompilovaných variant aplikace GeometryMaster se soustředila na porovnání dopadu optimalizačních úrovní GCC, absence symbolů a RTTI na výsledky RE C++ kódu v Ghidře. Aplikace implementuje hierarchii geometrických tvarů (`Shape`, `Circle`, `Rectangle`) s virtuálními metodami, dynamickou alokací a STL kontejnery (`std::vector<Shape*>`).

4.3.1 Specifický Postup Analýzy, Použité Nástroje a Nastavení Prostředí

Varianty GeometryMaster byly kompilovány pomocí GCC (verze 11.4.0) na Ubuntu 22.04 LTS. Analýza probíhala primárně v Ghidře (verze 11.0.3). Pro dynamickou analýzu, zejména u varianty V00 pro ověření struktur a volání, byl použit GDB (verze 14.1) s rozšířením GEF. Postup zahrnoval detailní statickou analýzu každé varianty (rekonstrukce tříd, vtables,

metod), dynamickou analýzu pro ověření a stručnou experimentální AI asistenci pro vybrané fragmenty kódu.

4.3.2 Výsledky Statické Analýzy a Vliv Optimalizací na Dekompilaci v Ghidře

Statická analýza jednotlivých variant v Ghidře jasně ukázala dramatický vliv úrovně optimalizace a přítomnosti symbolů na kvalitu a náročnost RE C++ kódu.

- **V00** (-m32 -g -O0 -fno-rtti -fno-exceptions): Díky ladicím symbolům (-g) Ghidra úspěšně demanglovala názvy tříd, metod a proměnných. Rekonstrukce tříd (Shape, Circle, Rectangle) a jejich vtables byla přímočará. Dekompilovaný kód byl velmi blízký původnímu zdrojovému kódu, s jasně identifikovatelnými voláními virtuálních metod. Čitelnost: 5/5.
- **V01** (-m32 -O1 -fno-rtti -fno-exceptions): Optimalizace úrovně 1 zavedla mírné inlinování jednoduchých funkcí (např. getters). Struktury tříd byly stále dobře rozeznatelné. Kvalita dekompile zůstala vysoká. Čitelnost: 4/5.
- **V02** (-m32 -O2 -fno-rtti -fno-exceptions): Znatelnější dopad optimalizací. Ghidra častěji zobrazovala proměnné typu undefined kvůli agresivnějšímu znovupoužití registrů. Některé menší funkce byly plně inlineovány, což zvětšilo komplexitu volajících funkcí. Rekonstrukce vtable byla náročnější bez symbolů. Čitelnost: 3/5.
- **V03** (-m32 -Os -s -fno-rtti -fno-exceptions): Optimalizace na velikost (-Os) a odstranění symbolů (-s) výrazně ztížily analýzu. Agresivní inlining a techniky sdílení kódu vedly k hustému a méně čitelnému kódu. Všechny názvy byly generické. Rekonstrukce tříd byla velmi obtížná. Čitelnost: 2/5.
- **V04** (-m64 -O2 -s -fno-rtti -fno-exceptions): Podobné V02, ale pro 64bitovou architekturu. Absence symbolů byla hlavním problémem. Rozdíly v konvencích volání pro x64 ovlivnily vzhled kódu. Čitelnost: 2.5/5.
- **V05** (-m64 -O3 -s -fno-rtti -flto): Nejobtížnější varianta. Optimalizace -O3 a LTO vedly k rozsáhlému inlinování, vektorizaci smyček a agresivnímu přeuspořádání instrukcí. Výstup dekompilátoru Ghidry byl často velmi vzdálený původní C++ logice. Například dekompile funkce main (poskytnutá v podkladech jako 'ulong

`main(void)‘)` ukazuje extrémní komplexitu s mnoha nepřímými voláními (např. `*(puVar12 + -8) = 0x11f1; fcn.00001c20(*0x20f8,*0x20f0,uVar5);‘`), obtížně sledovatelným tokem řízení a manipulací s ukazateli (např. `‘puVar26 = *(puVar13 + 0x38);‘`), kde původní smyčka přes vektor tvarů a volání virtuálních metod jsou silně transformovány. Rekonstrukce tříd byla extrémně náročná. Čitelnost: 1.5/5.

Absence RTTI (kvůli `-fno-rtti`) ve všech variantách znesnadnila automatickou rekonstrukci hierarchie tříd a identifikaci typů objektů za běhu, což vyžadovalo pečlivou manuální analýzu vtables a vzorů volání.

4.3.3 Výsledky Dynamické Analýzy

Dynamická analýza pomocí GDB s GEF byla nejpřínosnější pro variantu V00, kde umožnila snadné ověření struktur tříd, hodnot datových členů a skutečných cílů volání virtuálních metod díky přítomnosti ladicích symbolů. U optimalizovaných variant (V01-V05) pomohla sledovat tok řízení a hodnoty klíčových ukazatelů (např. `this` ukazatel, `vtable` ukazatele), i když ladění bylo podstatně náročnější kvůli absenci symbolů a transformacím kódu. Bylo možné potvrdit, že i v optimalizovaných verzích mechanismus virtuálních volání stále funguje přes `vtable`, i když samotná `vtable` mohla být optimalizována.

4.3.4 Výsledky AI-Asistované Analýzy (Experimentální)

Experimentální použití AI nástrojů na varianty Aplikace C bylo velmi omezené. Stručné testy s cloudovým API (Claude 3 Sonnet) na komplexních fragmentech dekompilovaného kódu z varianty V05 ukázaly občasnou schopnost identifikovat inlineované standardní knihovní funkce (např. části `std::vector` operací nebo matematických funkcí). Model však také projevoval tendenci k "halucinacím", kdy popisoval neexistující komplexní algoritmy na základě nesprávné interpretace optimalizovaných sekvencí instrukcí. Lokální modely (Mistral-7B-Instruct přes GhidrAssist) byly pro návrh smysluplných názvů proměnných nebo funkcí jen mírně nápomocné u optimalizovaného kódu, často navrhovaly příliš generické názvy.

4.3.5 Vyhodnocení Analýzy Aplikace C

Analýza Aplikace C potvrdila očekávané dopady optimalizací a absence symbolů na náročnost RE C++ kódu.

- **Kvalita dekompilovaného kódu (Ghidra):** Výrazně závislá na úrovni optimalizace a přítomnosti symbolů/RTTI. Od vynikající (V00: 5/5) po velmi špatnou (V05: 1.5/5).
- **Efektivita analytického procesu:** Časová náročnost se dramaticky zvyšovala s úrovní optimalizace. Pro V00 nízká, pro V02 střední a pro V05 vysoká až velmi vysoká, přičemž některé části zůstaly nejasné.
- **Splnění cílů analýzy:** Cíle splněny. Demonstrován silný vliv optimalizací, absence symbolů a RTTI na analyzovatelnost C++ kódu. OOP struktury byly identifikovány v různé míře úspěšnosti v závislosti na variantě.
- **Hlavní výzvy:** Rekonstrukce C++ tříd a vtables bez RTTI a symbolů, rozsáhlé inlinování funkcí, optimalizační transformace kódu (reordering, vektorizace), devirtualizace.
- **Přínos nástrojů:** Ghidra jako primární nástroj pro statickou analýzu a dekompilaci. GDB+GEF pro dynamické ověření a sledování. Srovnání různých kompilovaných variant bylo klíčové pro pochopení dopadů optimalizací.

Tabulka 4.1 Srovnání Charakteristik Variant Aplikace GeometryMaster (Vybrané Aspekty).

Varianta	Příznaky GCC	Rekonstrukce tříd (Ghidra)	Čitelnost kódu (1-5)
V00	-m32 -g -O0 -fno-rtti	Velmi dobrá	5
V02	-m32 -O2 -fno-rtti	Uspokojivá	3
V05	-m64 -O3 -s -fno-rtti -flto	Velmi obtížná	1.5

(zdroj: vlastní)

5 SROVNÁNÍ EFEKTIVITY METOD, NÁSTROJŮ A DISKUZE

Tato kapitola syntetizuje zjištění z praktické analýzy prezentované v Kapitole 4, srovnává efektivitu použitých metod a nástrojů reverzního inženýrství a diskutuje pozorované výsledky v kontextu teoretických předpokladů práce. Cílem je poskytnout kritické zhodnocení zvolených přístupů a identifikovat jejich silné a slabé stránky při analýze různých typů C/C++ desktopových aplikací. Tento proces reflexe je zásadní pro formulaci poučených závěrů a doporučení.

5.1 Srovnávací Analýza Použitých Metod

V rámci praktické části byly aplikovány různé metody reverzního inženýrství. Tato sekce srovnává jejich efektivitu a vzájemnou synergii na základě pozorování z analýzy vybraných aplikací.

5.1.1 Statická vs. Dynamická Analýza a Jejich Synergie

Praktická analýza všech tří aplikací jednoznačně potvrdila, že nejefektivnějšího přístupu k reverznímu inženýrství je dosaženo **kombinací statické a dynamické analýzy**.

- **Statická analýza** (primárně pomocí Ghidry) poskytla celkový přehled o struktuře programu, identifikovala funkce, datové struktury a základní logiku. Umožnila analyzovat kód bez jeho spuštění, což je klíčové pro bezpečnost u potenciálně škodlivého kódu (Aplikace B). U Aplikace A (bbbbloat) odhalila klíčovou porovnávací podmínku a mechanismus sestavení flagu. U Aplikace C (GeometryMaster) umožnila studovat dopad optimalizací na strukturu C++ kódu a identifikovat potenciální umístění vtables. Její hlavní limitací je obtížnost interpretace silně obfuskovaného nebo optimalizovaného kódu a nemožnost pozorovat runtime hodnoty a skutečný tok řízení v komplexních scénářích.
- **Dynamická analýza** (pomocí GDB+GEF a x32dbg) byla nezbytná pro ověření hypotéz ze statické analýzy, sledování skutečného toku řízení a hodnot proměnných v paměti. U Aplikace B (Shellcode) byla klíčová pro deobfuskaci XOR šifrovaného payloadu a potvrzení dynamicky resolvovaných API funkcí. U Aplikace A umožnila potvrdit správný vstupní údaj a pozorovat generování flagu. U Aplikace C pomohla ověřit

volání virtuálních metod a sledovat stav objektů za běhu, což bylo užitečné zejména u optimalizovaných variant. Jejím limitací je, že pokrývá pouze ty části kódu, které jsou během konkrétního běhu vykonány, a může být časově náročná pro komplexní aplikace s mnoha stavů.

- **Synergie:** Největší přínos byl pozorován tam, kde se obě metody doplňovaly. Například, statická analýza shellcode identifikovala XOR smyčku, ale až dynamická analýza potvrdila klíč a dešifrovaný obsah payloadu. U GeometryMaster, statická analýza ukázala potenciální volání virtuálních metod, dynamická analýza pak potvrdila, která konkrétní metoda byla pro daný objekt skutečně volána v konkrétním scénáři. Tento iterativní přístup, kde zjištění z jedné metody informují a zpřesňují druhou, je zásadní pro efektivní RE.

5.1.2 Specifické Techniky pro C++

Analýza Aplikace C (GeometryMaster) ukázala náročnost reverzního inženýrství C++ kódu, zejména bez RTTI a symbolů. Úspěšná analýza vyžadovala aplikaci specifických technik zaměřených na rekonstrukci objektově orientovaných konstrukcí.

- **Rekonstrukce tříd a vtables:** Manuální analýza vtables (identifikace ukazatelů na funkce v paměti) a vzorů přístupu k datovým členům (offsety od ukazatele `this`) byla nezbytná. Ghidra poskytla základní nástroje pro definování struktur a tříd, ale proces byl pracný a náchylný k chybám u optimalizovaných variant, kde kompilátor mohl měnit uspořádání členů nebo vtables. Identifikace konstruktorů a destruktorů (pokud nebyly inlineovány) a sledování ukazatele `this` pomohly při mapování datových členů.
- **Identifikace virtuálních volání:** Byla možná sledováním nepřímých volání (`call [reg+offset]`) přes ukazatele na vtable. Dynamická analýza byla užitečná pro potvrzení cílových funkcí těchto volání v konkrétních případech použití.
- **Vliv optimalizací:** Byl pozorován dramatický dopad. Inlinování funkcí, reordering instrukcí, optimalizace smyček a další transformace výrazně ztěžovaly mapování dekompilovaného kódu zpět na původní C++ konstrukce. Porozumění těmto transformacím a schopnost "vidět skrz ně" je klíčové pro úspěšnou analýzu optimalizovaného C++ kódu.

5.2 Srovnání Efektivity a Použitelnosti Nástrojů

Tato sekce hodnotí efektivitu a použitelnost klíčových nástrojů použitých v praktické části, s důrazem na jejich silné a slabé stránky v kontextu analyzovaných aplikací a úkolů.

5.2.1 Hlavní Nástroje Statické Analýzy

- **Ghidra:** *Silné stránky:* Vynikající integrovaný dekompilátor (zejména pro C a méně optimalizovaný C++), rozsáhlé analytické schopnosti (analýza CFG, datových typů, křížových referencí), podpora skriptování (Python/Jython, Java), multiplatformní, zdarma. Velmi dobře si poradila s Aplikací A a poskytla dobrý základ pro analýzu Aplikace B a C. *Slabé stránky:* Strmější křivka učení pro pokročilé funkce, kvalita dekompilace C++ kódu výrazně klesá s úrovní optimalizace a absencí symbolů/RTTI. Uživatelské rozhraní může být pro začátečníky méně intuitivní. *UX:* Dobrá, ale vyžaduje čas na seznámení a přizpůsobení.
- **Radare2:** *Silné stránky:* Extrémně flexibilní a výkonný framework pro nízkoúrovňovou analýzu, vynikající skriptovací možnosti, široká podpora architektur. Užitečný pro rychlý průzkum, detailní inspekci binárních dat a alternativní pohled na disassembler u Aplikace B. *Slabé stránky:* Velmi strmá křivka učení pro příkazovou řádku (r2). Dekompilační schopnosti závisí na externích pluginech (např. r2ghidra-dec). *UX (CLI):* Nízká pro začátečníky; Cutter (GUI) ji zlepšuje, ale stále je patrná komplexita jádra.
- **GNU Binutils (objdump):** *Silné stránky:* Standardní, rychlý a spolehlivý nástroj pro základní inspekci ELF souborů (sekce, symboly, disassembler). Užitečný pro Aplikaci A a C (Linux varianty) pro rychlé ověření. *Slabé stránky:* Omezeno na statickou analýzu, neposkytuje dekompilaci ani pokročilé analytické funkce. *UX (CLI):* Dobrá pro uživatele zvyklé na příkazovou řádku.

5.2.2 Hlavní Nástroje Dynamické Analýzy

- **x32dbg / x64dbg:** *Silné stránky:* Velmi intuitivní a uživatelsky přívětivý debugger pro Windows s bohatými funkcemi (breakpointy, sledování paměti/registrů, patchování, skriptování). Vynikající pro analýzu Aplikace B (Shellcode loader) a případně

Windows variant Aplikace C. *Slabé stránky*: Omezeno na platformu Windows. *UX*: Vynikající.

- **GDB s GEF**: *Silné stránky*: Extrémně mocný a flexibilní debugger, standard pro Linux. GEF výrazně zlepšuje použitelnost pro RE (lepší vizualizace, užitečné příkazy). Nepostradatelný pro analýzu Aplikace A a C (Linux varianty). Silné skriptovací možnosti v Pythonu. *Slabé stránky*: Samotný GDB má strmou křivku učení; GEF ji zmírňuje, ale stále jde primárně o CLI nástroj. *UX*: GDB nízká, s GEF výrazně lepší.
- **Monitorovací nástroje (strace, ltrace)**: *Silné stránky*: Poskytují rychlý přehled o interakcích programu s operačním systémem (systémová volání, volání knihoven) bez nutnosti hlubokého ladění. Užitečné pro Aplikaci A. *Slabé stránky*: Poskytují pouze pohled "zvenčí", neumožňují detailní krokování kódu nebo inspekci interního stavu programu tak jako debugger. Výstup může být velmi objemný.

5.2.3 Hodnocení Experimentálního Využití AI/ML Metod

Experimentální a velmi omezené nasazení AI/ML nástrojů (jako r2ai s Mistral-7B-Instruct nebo GhidrAssist) ukázalo jejich potenciál jako doplňkových asistentů pro dílčí úkoly, jako je základní sumarizace jednoduchých funkcí nebo návrhy názvů proměnných. Jejich efektivita však výrazně klesala s komplexitou kódu, úrovní optimalizace a obfuskace. Výstupy často vyžadovaly pečlivou manuální validaci a korekci. V současné fázi vývoje tyto nástroje nenahrazují expertního analytika, ale mohou v některých případech urychlit dílčí, dobře definované kroky analýzy.

5.3 Diskuze Výsledků a Splnění Cílů

Praktická část práce poskytla řadu poznatků, které potvrzují, rozšiřují nebo konfrontují teoretické předpoklady o reverzním inženýrství C/C++ aplikací.

- **Potvrzení teoretických předpokladů**: Analýzy potvrdily negativní dopad kompilátorových optimalizací a absence symbolů/RTTI na analyzovatelnost C++ kódu. Byla potvrzena nezbytnost kombinace statických a dynamických metod, zejména u obfuskovaného kódu. Byly demonstrovány specifické výzvy spojené s RE C++ konstrukcí.
- **Efektivita moderních dekompilátorů**: Nástroje jako Ghidra jsou velmi schopné, ale

jejich efektivita a přesnost dekompilace klesá s komplexitou kódu a úrovní optimalizace. Manuální analýza a interpretace zůstávají nezbytné.

- **Výzvy analýzy různých typů kódu:** Každá analyzovaná aplikace představovala unikátní výzvy: algoritmická logika a hledání "flagu" u CTF úlohy (Apl. A), deobfuskace a API hashing u shellcode (Apl. B), a rekonstrukce OOP struktur u různě optimalizovaných C++ variant (Apl. C).
- **Limity provedené praktické analýzy:** Rozsah analyzovaných aplikací byl omezený. Nebylo provedeno srovnání s komerčními dekompilátory (např. IDA Pro). Testování AI/ML nástrojů bylo pouze stručné a experimentální. Hodnocení časové náročnosti bylo kvalitativní a subjektivní.
- **Splnění cílů praktické části:** Cíle definované v zadání práce (body 3, 4, 5) byly naplněny prostřednictvím výběru a charakterizace aplikací, jejich detailní analýzy a vyhodnocení výsledků, a srovnáním efektivit použitých metod a nástrojů.

Celkově praktická část poskytla cenné vhledy do problematiky RE C/C++ aplikací a demonstrovala aplikaci teoretických znalostí a praktických dovedností v této oblasti.

Tabulka 5.1 Subjektivní Hodnocení Uživatelské Přívětivosti Klíčových Analytických Nástrojů [1]

Nástroj	Náročnost Počátečního Nastavení	Křivka Učení (5=strmá)	Intuitivnost UI	Efektivita pro Statickou Analýzu	Efektivita pro Dynamickou Analýzu	Vhodnost pro C++ Analýzu (1-5)
Ghidra	2	4	3	5	3 (přes debugger)	4
Radare2 (+Cutter)	3	5 (r2), 3 (Cutter)	2 (r2), 4 (Cutter)	4	4 (přes debugger)	3
x64dbg/x32dbg	1	2	5	2	5	3
GDB (+GEF)	2	4 (GDB), 3 (GEF)	2 (GDB), 4 (GEF)	1	5	3

(zdroj: vlastní)

Poznámka: Hodnocení je subjektivní na základě "zkušeností" z této práce. 1 = nejhorší/nejtěžší, 5 = nejlepší/nejsnadnější (kromě Křivky Učení, strmá).

6 ZÁVĚRY, DOPORUČENÍ A NÁVRHY DALŠÍ PRÁCE

Tato závěrečná kapitola shrnuje klíčové poznatky získané během celé práce, se zvláštním důrazem na výsledky praktické části. Na základě těchto zjištění jsou formulována obecná doporučení pro efektivní reverzní inženýrství C/C++ aplikací a jsou navrženy možné směry pro další výzkum a práci v této dynamicky se rozvíjející oblasti. Cílem je nejen uzavřít provedený výzkum, ale také poskytnout prakticky využitelné rady a inspirovat k dalšímu bádání.

6.1 Hlavní Závěry Vyplývající z Práce

Na základě teoretického studia a především praktické analýzy vybraných aplikací lze formulovat následující hlavní závěry, které reflektují současný stav a výzvy v oblasti reverzního inženýrství C/C++ aplikací:

1. **Kombinace statické a dynamické analýzy je nejefektivnější:** Pro komplexní porozumění softwaru, zejména pokud je obfuskovaný nebo silně optimalizovaný, je nezbytné synergické využití obou přístupů. Statická analýza poskytuje celkový obraz a umožňuje identifikovat klíčové struktury, zatímco dynamická analýza umožňuje ověřit hypotézy, sledovat skutečné chování za běhu a překonat dynamické obfuskační techniky.
2. **Kompilátorové optimalizace a absence symbolů jsou hlavní překážky:** Analýza aplikace GeometryMaster jasně demonstrovala, že agresivní kompilátorové optimalizace (zejména -O2, -O3, -Os) a odstranění ladicích symbolů a RTTI dramaticky zvyšují náročnost reverzního inženýrství C++ kódu. Tyto faktory vedou ke ztrátě sémantických informací, znesnadňují rekonstrukci původních struktur a snižují čitelnost dekompilovaného kódu.
3. **Reverzní inženýrství C++ zůstává výzvou:** Specifika jazyka C++, jako jsou objektově orientované paradigma (třídy, dědičnost, polymorfismus), virtuální metody, šablony a použití standardní knihovny šablon (STL), kladou na analytiku a nástroje vysoké nároky, obzvláště při absenci RTTI a symbolů. Rekonstrukce třídních hierarchií a tabulek virtuálních metod vyžaduje značné manuální úsilí a hluboké porozumění tomu, jak kompilátory tyto konstrukce implementují.

4. **Moderní dekompilátory jsou mocné, ale ne všemocné:** Nástroje jako Ghidra představují obrovský pokrok v oblasti automatizované analýzy a dekompilace. Jejich schopnosti však mají své limity, zejména u komplexního, silně optimalizovaného nebo "nepřátelského" (záměrně obfuskovaného) kódu. Manuální analýza, expertní znalosti a kritické myšlení analytika jsou stále nenahraditelné.
5. **Obfuskační techniky vyžadují cílené postupy:** Analýza shellcode ukázala, že techniky jako XOR šifrování payloadu a API hashing vyžadují specifické metody pro deobfuskaci a pochopení. Často je nutné kombinovat statickou analýzu pro identifikaci vzorů s dynamickou analýzou pro sledování deobfuskačních rutin a odhalení skryté logiky. Pomocné skripty jsou často nezbytné pro automatizaci těchto procesů.
6. **AI/ML nástroje mají potenciál jako asistenti, ale vyžadují kritické hodnocení:** Experimentální použití AI/ML nástrojů naznačilo jejich schopnost pomoci s dílčími úkoly, jako je sumarizace jednoduchých funkcí nebo návrhy názvů proměnných. Jejich současné schopnosti jsou však omezené u komplexního kódu a jejich výstupy je nutné vždy pečlivě ověřovat. Nemohou nahradit lidskou expertízu a hluboké porozumění. AI models: As the models progress I am finding AI increasingly helpful. It's never going to get the 'big picture' but at a function level I'm getting some value out of it. I can feed it register and stack contents at the start and end of some assembly and it's not bad at relating the function of the code, with guidance.
7. **Výběr nástrojů závisí na kontextu:** Neexistuje jeden univerzální nejlepší nástroj pro všechny úlohy RE. Volba by měla záviset na typu analyzované aplikace, cílové platformě, cílech analýzy a preferencích a zkušenostech analytika. Znalost více nástrojů a schopnost je efektivně kombinovat je klíčovou dovedností.

6.2 Vypracování Obecných Doporučení pro Reverzní Inženýrství C/C++ Aplikací

Na základě získaných zkušeností a zjištění z praktické části lze formulovat následující obecná doporučení pro praxi reverzního inženýrství C/C++ aplikací. Tato doporučení jsou zaměřena na zvýšení efektivity a úspěšnosti analytického procesu.

- **Systematický a iterativní přístup:** Postupujte metodicky od prvotního průzkumu přes detailní statickou a dynamickou analýzu. Buďte připraveni iterovat mezi fázemi, vra-

cet se k předchozím krokům a zpřesňovat své hypotézy na základě nových zjištění. Neexistuje lineární cesta k pochopení komplexního softwaru.

- **Kombinujte metody a nástroje:** Využívejte silné stránky různých přístupů a nástrojů. Statická analýza pro celkový přehled a identifikaci struktur, dynamická analýza pro ověření hypotéz a sledování runtime chování. Používejte více nástrojů pro křížovou validaci zjištění a pro získání různých perspektiv na analyzovaný kód.
- **Důkladná dokumentace:** Pečlivě dokumentujte všechny své kroky, pozorování, formulované hypotézy (i ty, které se ukázaly jako nesprávné) a získané výsledky. Anotujte kód v analyzátorech (přejmenovávání funkcí, proměnných, datových typů, přidávání komentářů). To je klíčové pro orientaci v komplexním kódu, pro budoucí návrat k analýze nebo pro sdílení poznatků s ostatními.
- **Vytvářejte izolovaná analytická prostředí:** Zejména při analýze potenciálně škodlivého kódu nebo neznámých aplikací používejte virtuální stroje s omezeným nebo žádným síťovým připojením. Pravidelně vytvářejte snapshoty pro možnost snadného návratu k čistému stavu systému.
- **Zaměřte se na pochopení "proč", nejen "co":** Snažte se nejen identifikovat jednotlivé instrukce nebo funkce, ale pochopit jejich účel v kontextu celé aplikace a původní záměr vývojáře. Hledejte vysokoúrovňové vzory, abstrakce a designové principy.
- **Specifika C++ kódu:**
 - Pokud je to možné, pokuste se získat méně optimalizovanou verzi aplikace nebo verzi se symboly pro snazší počáteční orientaci a pro pochopení základních struktur.
 - Aktivně hledejte C++ specifické vzory: vtables, ukazatel `this`, konstruktory/destruktory, manglovaná jména (a nástroje pro jejich demangling), použití STL kontejnerů a algoritmů.
 - Buďte si vědomi dopadu optimalizací (inlinování, devirtualizace, optimalizace smyček) na strukturu kódu a připravte se na to, že dekompilovaný kód nemusí přímo odpovídat původnímu zdrojovému kódu.

- Věnujte pozornost správě paměti (`new/delete`, `malloc/free`) a potenciálním problémům s ní spojeným (`memory leaks`, `use-after-free`).
- **Využívejte skriptování pro automatizaci** repetitivních úloh. Mnoho nástrojů (Ghidra, Radare2, debuggery) nabízí skriptovací rozhraní (Python, Java atd.). Využijte je pro automatizaci repetitivních úloh (např. deobfuskace, extrakce dat, anotace, vyhledávání vzorů).
- **Bud'te kritičtí k výstupům nástrojů (včetně AI):** Dekompilátory a AI asistenti mohou generovat nepřesnosti nebo chyby. Vždy ověřujte jejich výstupy a spoléhejte na svůj úsudek a znalosti. Neberte žádný výstup jako absolutní pravdu.
- **Neustále se vzdělávejte:** Oblast reverzního inženýrství, stejně jako techniky kompilátorů a obfuskace, se neustále vyvíjí. Sledujte nové nástroje, techniky, výzkumné publikace a bezpečnostní konference.

6.3 Návrhy pro Další Práci a Výzkum

Tato diplomová práce otevřela několik oblastí, které by si zasloužily další, hlubší prozkoumání a představují potenciální směry pro budoucí výzkumné aktivity.

- **Hlubší analýza vlivu moderních C++ standardů** (C++17, C++20 a C++23) na analyzovatelnost kódu. Prozkoumat dopad konstrukcí jako moduly, koncepty, coroutines, ranges, `std::format` na schopnosti současných RE nástrojů.
- **Studium pokročilých obfuskačních technik** a anti-analytických postupů. Zaměřit se na sofistikovanější metody obfuskace (např. virtualizace kódu, polymorfismus, metamorfismus, pokročilé anti-debugging a anti-disassembly techniky) a zkoumat možnosti jejich (polo)automatizované detekce, analýzy a deobfuskace.
- **Rigorózní testování a srovnání AI-asistovaných RE nástrojů.** Provést systematické srovnání různých LLM modelů a pluginů pro RE na širší sadě benchmarků, hodnotit jejich přesnost, robustnost a praktickou využitelnost pro konkrétní RE úlohy. Zkoumat možnosti fine-tuningu LLM modelů na RE-specifická data.
- **Analýza dopadu Link-Time Optimization (LTO) a Whole Program Optimization (WPO)** na RE. Detailněji prozkoumat, jak tyto pokročilé optimalizační techniky mění

strukturu kódu napříč kompilačními jednotkami.

- **Vývoj specifických skriptů a pluginů pro RE C++.** Vytvořit nebo rozšířit existující nástroje pro lepší automatizaci rekonstrukce C++ tříd, vtables, šablon a dalších OOP struktur, zejména při absenci RTTI a symbolů.
- **RE kódu kompilovaného z jiných moderních jazyků** (např. Rust, Go) a srovnání výzev s C++.
- **Srovnání s komerčními nástroji** (např. IDA Pro, Binary Ninja) pro posouzení jejich silných a slabých stránek ve srovnání s open-source alternativami.
- **Vliv různých kompilátorů a jejich verzí** (GCC, Clang, MSVC) na analyzovatelnost kódu při stejných úrovních optimalizace.

Oblast RE je dynamická a nabízí mnoho zajímavých výzev pro budoucí výzkum.

ZÁVĚR

Práce komplexně zkoumala, analyzovala a hodnotila metody, nástroje a výzvy RE C/C++ aplikací. Propojila teorii s praxí na konkrétních příkladech pro ucelený pohled na tuto dynamickou disciplínu, včetně fundamentálních principů, specifik C/C++ a kritického pohledu na moderní pokroky jako AI a jejich dopad.

Klíčové poznatky: Praktické analýzy (bbbbloat, shellcode, GeometryMaster) potvrdily, že nejlepšího porozumění je dosaženo kombinací statické (Ghidra) a dynamické analýzy (GDB, x32dbg), což je fundament úspěšného RE. Zkoumání GeometryMaster prokázalo zásadní negativní vliv kompilátorových optimalizací a absence symbolů na RE C++ kódu, vedoucí ke ztrátě sémantické jasnosti a obtížné rekonstrukci (např. main ve V05, jak bylo demonstrováno v praktické části). Přetrvávají výzvy v rekonstrukci C++ OOP konstrukcí bez RTTI/symbolů, vyžadující manuální analýzu vtables a ukazatele this. Ghidra se osvědčila jako schopný open-source nástroj, s limity u silně optimalizovaného C++ a obfuskace. Analýza shellcode demonstrovala efektivní strategie proti obfuskaci (XOR, API hashing) pomocí kombinace přístupů. Experimentální nasazení AI/ML ukázalo potenciál jako asistenční technologie, avšak s nutností kritické validace výstupů kvůli riziku „halucinací“; směřuje k modelu „human-in-the-loop“.

Omezení výzkumu: Malý rozsah analyzovaných aplikací (více studií, např. komerčního softwaru/malware, by bylo přínosné). Zaměření na open-source nástroje (srovnání s komerčními jako IDA Pro by rozšířilo perspektivu). Limitované a explorativní experimenty s AI, jejichž hlubší nasazení bylo ztíženo praktickými aspekty, jako je náročnost na výpočetní výkon pro provozování vlastních pokročilejších modelů a komplexita spojená s konfigurací některých externích API; počet ověřovacích vzorků pro testování těchto AI metod byl rovněž omezený, což je třeba zohlednit při interpretaci jejich efektivity. Kvalitativní hodnocení časové náročnosti.

Obecná doporučení pro praxi RE C/C++: Systematický, iterativní přístup, kombinace statických/dynamických metod a nástrojů. Hluboké porozumění C++, kompilátorům a optimalizacím. Pečlivá dokumentace a anotace. Využití skriptování. Izolovaná prostředí pro škodlivý kód. Kritický přístup k výstupům nástrojů (včetně AI). Kontinuální vzdělávání.

Další výzkum a vývoj: Zlepšení dekompilace moderních C++ standardů (C++17/20/23).

Pokrok v AI-řízené sémantické rekonstrukci (názvy, typy, komentáře, logika). Robustnější AI-asistované deobfuskační techniky (např. pro virtualizaci kódu). Vývoj a standardizace komplexních benchmarků. Výzkum XAI pro RE. RE zůstává v cyklickém vývoji se softwarem, zajišťujíc jeho relevanci.

SEZNAM POUŽITÉ LITERATURY

- [1] ISO/IEC/IEEE, sep 2017. Systems and software engineering — Vocabulary. International Standard ISO/IEC/IEEE 24765:2017(E), International Organization for Standardization; International Electrotechnical Commission; Institute of Electrical and Electronics Engineers. Dostupné z: <https://doi.org/10.1109/IEEESTD.2017.8016712>. Publikováno 28. srpna 2017; nahrazuje ISO/IEC/IEEE 24765:2010.
- [2] CHIKOFSKY, Elliot J. a CROSS, II J. H., 1990. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*. 1990. 7, č. 1, s. 13–17. Dostupné z: 10.1109/52.43044.
- [3] NATIONAL SECURITY AGENCY, 2019. Ghidra. Open Source Software. Dostupné z: <https://ghidra-sre.org>. Archived at: <https://ghidra-sre.org> and <https://www.nsa.gov/resources/everyone/ghidra/> (original URLs).
- [4] TAN, Cheng; LI, Yinuo; PENG, Siyuan; LI, Lichao; ZHOU, Ming et al., Mar 2024. LLM4Decompile: A Large Language Model for Decompilation. arXiv preprint arXiv:2403.05286. Dostupné z: <https://arxiv.org/abs/2403.05286>.
- [5] SCIENCE OF SECURITY VIRTUAL ORGANIZATION, 2024. 2024 Q1 Presentations: VarBERT: A Neural Model for Variable Name Recovery in Decompiled Code (Purdue University) and SAILR: Structure-Aware Intermediate Language Restorer for Decompiled C Code (Purdue University). <https://sos-vo.org/index.php/node/106983>. Přístup dne: 2025-05-30.
- [6] LI, Zirui; SPENSKY, Chad a TAN, Lin, 2024. ReSym: A Hybrid Approach for Binary Local Variable Recovery and User-Defined Structure Reconstruction. Technical report, Purdue University, Department of Computer Science. Dostupné z: <https://www.cs.purdue.edu/homes/lintan/publications/resym-ccs24.pdf>. Pravděpodobně odkazuje na dokumentaci k příspěvku pro CCS 2024. Přesná citace by vyžadovala finální publikaci.
- [7] NAGY, Stefan a KEMERLIS, Vasileios P., 2025. BIN2WRONG: A Unified Fuzzing Framework for Uncovering Semantic Errors in Binary-to-C Decompilers. In: *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC '25)*. Dostupné z: <https://users.cs.utah.edu/~snagy/papers/25ATC.pdf>. Předpoklad na základě "25ATC.pdf", přesné informace o konferenci se mohou lišit.
- [8] OPENAI, 2025. ChatGPT. <https://chatgpt.com/>. Datum přístupu: 2. června 2025.
- [9] MICROSOFT, 2025. Microsoft Copilot. <https://copilot.microsoft.com/>. Datum přístupu: 2. června 2025.
- [10] GOOGLE, 2025. Gemini. <https://gemini.google.com/>. Datum přístupu: 2. června 2025.
- [11] IEEE COMPUTER SOCIETY, oct 1998. *IEEE Standard for Software Maintenance*. Online. IEEE. Full designation: IEEE Std 1219-1998.
- [12] EILAM, Eldad, 2005. *Reversing: Secrets of Reverse Engineering*. Wiley Publishing. ISBN 978-0-7645-7481-8.
- [13] PFLEEGER, Shari Lawrence a ATLEE, Joanne M., 2009. *Software Engineering: Theory and Practice*. Prentice Hall. 4th vyd. ISBN 978-0136061694.
- [14] MANUEL, David a AGUILAR, Antonio, 2024. DeBinVul: Advancing Binary Vulnerability Detection with LLMs through Transfer Learning, Code Augmentation, and Context-aware Prompting. *arXiv preprint arXiv:2403.07134*. 2024. Dostupné z: 10.48550/arXiv.2403.07134. Dostupné z: <https://arxiv.org/abs/2403.07134>.

- [15] QI, Yuyang; CONTINELLA, Andrea; LI, Zhaofeng; WANG, Ruotong; ZHOU, Shiqing et al., March 2024. LogicMEM: Automatic Forensics of Memory Images with Logic Programming and Abstract Interpretation. *IEEE Transactions on Dependable and Secure Computing*. March 2024. 21, č. 2, s. 1067–1082. Dostupné z: 10.1109/TDSC.2022.3210673. Published early access in 2022. Final publication in Vol. 21, No. 2, March-April 2024.
- [16] HU, Qiyi; PENG, Xiaofei; LI, Zhengzi; SUN, Jun; ZOU, Deqing et al., 2024. DeGPT: Large Language Models for Decompilation. *arXiv preprint arXiv:2402.01067*. 2024. Dostupné z: 10.48550/arXiv.2402.01067. Dostupné z: <https://arxiv.org/abs/2402.01067>.
- [17] JIA, Yifan; WU, Xiaoyu; ZHAO, Wentao a LI, Xuan, July 2024. A Comprehensive Review on Software Re-engineering: Techniques, Applications, Challenges and Future Trends. *Journal of Software: Evolution and Process*. July 2024. 36, č. 7, s. e2664. Dostupné z: 10.1002/smr.2664. The year in the key KEY_Jia2025ReEngReview (2025) might indicate intended final publication or volume year; the article was published in Volume 36, Issue 7, July 2024.
- [18] WASHIZAKI, Hironori (Ed.), 2024. *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide)*. IEEE Computer Society. Version 4.0 vyd. Dostupné z: <https://www.swebok.org>. Accessed 2025-05-19.
- [19] ROSENBLATT, Alfred, March 1987. Phoenix's role in the complex world of PC-compatibles. *Electronics*. March 1987. 60, č. 6, s. 67–68.
- [20] UNITED STATES COURT OF APPEALS, NINTH CIRCUIT, 1993. *Sega Enterprises Ltd. v. Accolade, Inc.*, 977 F.2d 1510 (9th Cir. 1992, opinion amended 1993). *Federal Reporter, 2nd Series*. 1993. 977, s. 1510. The primary opinion was issued in 1992 and amended in 1993. Commonly cited as 977 F.2d 1510.
- [21] KAPLAN, David A., January 2000. The Cutting Edge: Hollywood's hacker hell. *Newsweek*. Refers to the DeCSS controversy and DVD CCA v. Bunner. While an exact DOI for a legal case or a specific news article might be hard, this period covers the DeCSS discussions.
- [22] LANGNER, Ralph, 2011. Stuxnet: Dissecting a Cyberwarfare Weapon. In: *2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET '11)*. While Langner gave many talks, for a citable paper, his Black Hat presentation or IEEE Security & Privacy articles are also relevant. This reference is often to his broader analysis presented in various forums.
- [23] TELEA, Alexandru C., 2007. *An Introduction to Source Code Analysis and Manipulation*. World Scientific Publishing Co. Pte. Ltd. ISBN 978-9812708339. Dostupné z: 10.1142/6553.
- [24] MCCONNELL, Steve, 2004. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press. 2nd vyd. ISBN 978-0735619678.
- [25] PRESSMAN, Roger S. a MAXIM, Bruce R., 2010. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education. 7th vyd. ISBN 978-0073375977.
- [26] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph a VLISSIDES, John, 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional. ISBN 978-0201633610.
- [27] MCCABE, Thomas J., 1976. A Complexity Measure. *IEEE Transactions on Software Engineering*. 1976. SE-2, č. 4, s. 308–320. Dostupné z: 10.1109/TSE.1976.233837.
- [28] SHNEIDERMAN, Ben, 1980. *Software Psychology: Human Factors in Computer and Information Systems*. Little, Brown and Company (Winthrop Publishers). ISBN 978-0876268162.
- [29] MUCHNICK, Steven S., 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc. ISBN 978-1558603202.
- [30] JIANG, Zhou; LE, Wei a GAO, Dezhi, 2024. SYMGEN: Automated Generation of Semantic Names for Functions in Stripped Binaries. *arXiv preprint arXiv:2404.03635*. 2024. Dostupné z: 10.48550/arXiv.2404.03635. Dostupné z: <https://arxiv.org/abs/2404.03635>. The year in the key KEY_Jiang2025SYMGEN (2025) might indicate intended final publication; preprint appeared in 2024.

- [31] VANDEVOORDE, David; JOSUTTIS, Nicolai M. a GREGOR, Douglas, 2017. *C++ Templates: The Complete Guide*. Addison-Wesley Professional. 2nd vyd. ISBN 978-0321714121. This entry refers to the 2nd edition published in 2017. The first edition by Vandevoorde and Josuttis was published in 2002. The key Vandewalle2007 might be used to refer to concepts discussed across editions or stemming from the earlier work.
- [32] OLSSON, Mikael, 2012. *The Lost Art of C++ Dev: A Practical Guide to Building Reliable, Maintainable, and Scalable Applications with C++*. John Wiley & Sons. ISBN 978-1118109890. This is a general C++ book; specific ABI details are usually in compiler documentation or the Itanium C++ ABI specification. This might be a more general reference to complexities in C++ like inheritance.
- [33] MULTIPLE AUTHORS FROM HP, INTEL, RED HAT, SGI, AND OTHERS, 2001. Itanium C++ ABI. Technical report, Itanium C++ ABI Group (see URL). Dostupné z: <https://itanium-cxx-abi.github.io/cxx-abi/abi.html>. Often refers to the specification. The original specification is older but actively maintained/referenced.
- [34] SIKORSKI, Michael a HONIG, Andrew, 2012. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press. ISBN 978-1593272906.
- [35] WANG, Shuaifu; WANG, Pei a ZHAO, Ming, 2017. A Survey on Software Obfuscation. In: *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)*. s. 757–762. Dostupné z: 10.1109/ICBDA.2017.8078772.
- [36] JIN, Peng; ZHANG, Hongbo; DUAN, Yue; LIU, Yang; SUN, Kun et al., 2024. Quantifying the Semantic Difference in Obfuscated Code via Probabilistic Symbolic Execution. *IEEE Transactions on Software Engineering*. 2024. Dostupné z: 10.1109/TSE.2024.3399812. Published early access in 2024.
- [37] *Assembly Programming Tutorial*, 2024. Online. Dostupné z: https://www.tutorialspoint.com/assembly_programming/index.htm. Accessed on 2024-10-22.
- [38] YURICHEV, Dennis, 2015. *Understanding Assembly Language*. Dostupné z: <https://bdigital.uvhm.edu.mx/wp-content/uploads/2020/07/Reverse-Engineering-for-Beginners.pdf>. Copyright 2013-2015. The hosting PDF is titled "Reverse Engineering for Beginners". Accessed on 2024-10-22.
- [39] ANDRIESSE, Dennis, 2019. *Practical binary analysis: build your own Linux tools for binary instrumentation, analysis, and disassembly*. No Starch Press. ISBN 978-1-59327-912-7.
- [40] DANG, Bruce; GAZET, Alexandre; BACHAALANY, Elias a JOSSE, Sébastien, 2014. *Practical reverse engineering: x86, x64, ARM, Windows Kernel, reversing tools, and obfuscation*. John Wiley. ISBN 978-1-118-78725-0. Dostupné z: <https://ebookcentral.proquest.com/lib/natl-ebooks/detail.action?docID=1629173>. Accessed on 2024-10-22 via ProQuest Ebook Central.
- [41] JIANG, Jiacheng; DUAN, Yue; WANG, Zhiyao; WU, Chou a ZHANG, Xiang, Apr 2025. Neural Network Assisted Reverse Engineering of Binary Programs: A Survey. arXiv preprint arXiv:2504.21803. Dostupné z: <https://arxiv.org/pdf/2504.21803>.
- [42] ALOMAR, Eman Abdullah; MKAOUER, Mohamed Wiem; KESSENTINI, Marouane a OUNI, Ali, 2024. Model-Driven Approaches for Reverse Engineering - A Systematic Literature Review. *Information and Software Technology*. 2024. 173, s. 107493. Dostupné z: 10.1016/j.infsof.2024.107493. Dostupné z: https://www.researchgate.net/publication/380197108_Model-Driven_Approaches_for_Reverse_Engineering_-_A_Systematic_Literature_Review. Původní odkaz byl na ResearchGate, zde je finální publikace.
- [43] WONG, Reginald, 2018. *Mastering Reverse Engineering*. Packt Publishing Limited. ISBN 978-1788838849. Dostupné z: <https://download.packt.com/free-ebook/9781788838849>. Accessed on 2024-10-22.
- [44] HU, Xinyu; HE, Zhibo; SU, Zhendong a LIU, Yang, May 2025. DecompileBench: A Comprehensive Benchmark for Evaluating Decompilers in Real-World Scenarios. arXiv preprint arXiv:2505.11340. Dostupné z: <https://arxiv.org/pdf/2505.11340>.

- [45] 0XINFECTION, 2021. *Reverse Engineering For Everyone!* Online. Dostupné z: <https://0xinfection.github.io/reversing/>. Accessed on 2024-10-22.
- [46] APRIORIT, Oct 2023. Best Reverse Engineering Tools. <https://www.apriorit.com/dev-blog/366-software-reverse-engineering-tools>. Datum publikace na stránce není explicitně uvedeno. Přístup dne: 2025-05-30.
- [47] 2024. *Welcome to Hex-Rays docs*. Hex-Rays. Dostupné z: <https://docs.hex-rays.com/gitbook/pdf>. Documentation version: 2024-10-01. Accessed on 2024-10-22.
- [48] HEX-RAYS, May 2024. Unveiling IDA Pro 9.0: C++ Exceptions Support in the Decompiler. <https://hex-rays.com/blog/unveiling-ida-pro-9-0-c-exceptions-support-in-the-decompiler/>. Přístup dne: 2025-05-29.
- [49] HEX-RAYS. Pricing. <https://hex-rays.com/pricing>. Stránka s ceníkem, rok vydání není relevantní.
- [50] LIN, Ziyi; CHEN, Zishun; LIU, Hanzhi; MA, Lin a SU, Zhendong, 2024. Evaluating the Effectiveness of Decompilers. In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. ACM. Dostupné z: <https://2024.issta.org/details/issta-2024-papers/40/Evaluating-the-Effectiveness-of-Decompilers>.
- [51] NATIONAL SECURITY AGENCY. Ghidra. <https://github.com/NationalSecurityAgency/ghidra>. Oficiální repozitář, rok vydání není pevný.
- [52] SYGNIA, 2023. Reversing DoSvc C++ using Ghidra – Part 1. <https://www.sygnia.co/blog/reversing-dosvc-c-using-ghidra-part-1/>. Přístup dne: 2025-05-29.
- [53] RIZINORG. Cutter - Free and Open Source Reverse Engineering Platform powered by Rizin. <https://cutter.re/>. Webová stránka projektu, rok vydání není relevantní.
- [54] RADAREORG. r2ghidra: Native Ghidra Decompiler for r2. <https://github.com/radareorg/r2ghidra>. Repozitář projektu, rok vydání není pevný.
- [55] THE RADARE2 COMMUNITY. Decompilers - The Official Radare2 Book. <https://book.rada.re/arch/decompile.html>. Online dokumentace, rok vydání není pevný.
- [56] X64DBG TEAM. x64dbg - An open-source x64/x32 debugger for Windows. <https://x64dbg.com/>. Webová stránka projektu, rok vydání není relevantní.
- [57] SIKORSKI, Michael a HONIG, Andrew, 2012. *Practical malware analysis: the hands-on guide to dissecting malicious software*. No Starch Press. ISBN 1593272901. Dostupné z: <https://ebookcentral.proquest.com/lib/natl-ebooks/detail.action?docID=1137570>. Accessed on 2024-10-22 via ProQuest Ebook Central.
- [58] VECTOR 35 INC., . Binary Ninja. <https://binary.ninja/>. Webová stránka produktu, rok vydání není relevantní.
- [59] VECTOR 35 INC., . Purchase - Binary Ninja. <https://binary.ninja/purchase>. Stránka s ceníkem, rok vydání není relevantní.
- [60] PNF SOFTWARE. JEB Decompiler - Reverse Engineering Platform for Professionals. <https://www.pnfsoftware.com/>. Webová stránka produktu, rok vydání není relevantní.
- [61] VINCENT BÉNONY. Hopper Disassembler. <https://www.hopperapp.com/>. Webová stránka produktu, rok vydání není relevantní.
- [62] AVAST. retdec: RetDec is a retargetable machine-code decompiler based on LLVM. <https://github.com/avast/retdec>. Repozitář projektu, rok vydání není pevný.
- [63] ALVAREZ, Sergi (Pancake), Apr 2025. Malware analysis assisted by AI with R2AI. arXiv preprint arXiv:2504.07574. Dostupné z: <https://arxiv.org/abs/2504.07574>.

- [64] VECTOR 35 INC., Nov 2024. 2024 Reverse Engineering Survey: The State of the Art. <https://binary.ninja/2024/11/08/user-survey-results.html>. Přístup dne: 2025-05-30. Datum je hypotetické, jelikož je v budoucnosti.
- [65] STUDSEC. Reverse Engineering. <https://wiki.studsec.nl/books/ctf-guides/page/reverse-engineering>. Wiki stránka, rok vydání není pevný.
- [66] TREBLEDJ, 2023. HKCERT CTF 2023 – Decompetition: Vitamin C++. <https://trebledj.me/posts/hkcert-2023-decompetition-vitamin-cpp/>. Přístup dne: 2025-05-30.
- [67] RECON, 2025. Recon Training - Reversing Modern Binaries: Practical Rust & Go. <https://www.recon.cx/2025/trainingReversingModernBinaries.html>. Přístup dne: 2025-05-30. Odkazuje na školení v roce 2025.
- [68] PEARCE, Logan; FARHANG, Sina; MCCOY, Damon a HUANG, Hsun, Mar 2025. Coding Malware in Fancy Programming Languages for Fun and Profit. arXiv preprint arXiv:2503.19058. Dostupné z: <https://arxiv.org/abs/2503.19058>.
- [69] OCEAN TOMO. Software and Firmware Reverse Engineering & Testing. <https://oceantomo.com/insights/software-and-firmware-reverse-engineering-testing/>. Webová stránka, rok vydání není jasně uveden.
- [70] SCIENCE SOFT, May 2024. Best Practices for Reverse Engineering of Legacy Applications. <https://www.scnsoft.com/software-development/about/how-we-work/reverse-engineering>. Přístup dne: 2025-05-30.
- [71] HU, Xinyu; HE, Zhibo; SU, Zhendong a LIU, Yang, May 2025. DecompileBench: A Comprehensive Benchmark for Evaluating Decompilers in Real-World Scenarios. arXiv preprint arXiv:2505.11340v1. Dostupné z: <https://arxiv.org/html/2505.11340v1>.
- [72] CINISELLI, Matteo; LOVETRO, Jonathan; MARCUS, Solomon; TITOV, Ivan; ZAMPETTI, Fabio et al., Mar 2025. Large Language Models (LLMs) for Source Code Analysis: applications, models and datasets. arXiv preprint arXiv:2503.17502. Dostupné z: <https://arxiv.org/html/2503.17502v1>.
- [73] PEARCE, Logan; FARHANG, Sina; MCCOY, Damon a HUANG, Hsun, Mar 2025. Coding Malware in Fancy Programming Languages for Fun and Profit. arXiv preprint arXiv:2503.19058. Dostupné z: <https://arxiv.org/pdf/2503.19058.pdf>.
- [74] LKIMUK, 2023. `cpp-resources/2023/videos/2023-all-in-one.md` at main. <https://github.com/lkimuk/cpp-resources/blob/main/2023/videos/2023-all-in-one.md>. Přístup dne: 2025-05-30.
- [75] YESHIVA UNIVERSITY KATZ SCHOOL BLOG, 2022. The 5 Best Reverse Engineering Software for 2022. <https://online.yu.edu/katz/blog/best-reverse-engineering-software-2022>. Přístup dne: 2025-05-30.
- [76] OXDEVALIAS, 2023. Some notes, tools, and techniques for reverse engineering macOS binaries. <https://gist.github.com/Oxdevalias/256a8018473839695e8684e37da92c25>. Přístup dne: 2025-05-30.
- [77] RADARE2, 2020. GSOC 2020 Ideas. <https://rada.re/gsoc/2020/ideas.html>. Přístup dne: 2025-05-30.
- [78] WINITOR. PESTudio - Static Analysis Tool for Executables. <https://www.winitor.com/>. Webová stránka nástroje, rok vydání není relevantní.

SEZNAM TABULEK

Tabulka 1.1	Srovnání klíčových pojmů souvisejících s Reverzním inženýrstvím.....	20
Tabulka 2.1	Srovnání Hlavních Dekompilačních Nástrojů pro C/C++ (se zaměřením na získávání pseudokódu).....	66
Tabulka 2.2	Přehled Metrik pro Hodnocení Efektivity Dekompilace a Jejich Aplikace	67
Tabulka 3.1	Přehled Analyzovaných Aplikací a Jejich Metodologických Cílů	70
Tabulka 3.2	Přehled nástrojů RE použitých v praktické části.....	72
Tabulka 4.1	Srovnání Charakteristik Variant Aplikace GeometryMaster (Vybrané Aspekty).	85
Tabulka 5.1	Subjektivní Hodnocení Uživatelské Přívětivosti Klíčových Analytických Nástrojů [1]	90

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

RE Reverzní Inženýrství (Reverse Engineering)

API Application Programming Interface (Rozhraní pro programování aplikací)

TCO Total Cost of Ownership (Celkové náklady na vlastnictví)

SDLC Software Development Lifecycle (Životní cyklus vývoje softwaru)

UML Unified Modeling Language (Jednotný modelovací jazyk)

GCC GNU Compiler Collection

MSVC Microsoft Visual C++ (Compiler)

NSA National Security Agency (Národní bezpečnostní agentura)

CSS Content Scramble System

DVD Digital Versatile Disc (Digitální víceúčelový disk) nebo Digital Video Disc (Digitální videodisk)

BIOS Basic Input/Output System

PDB Program Database

DWARF Debugging With Attributed Record Formats

ELF Executable and Linkable Format

RTTI Run-Time Type Information (Informace o typu za běhu)

SEH Structured Exception Handling (Strukturované zpracování výjimek)

ABI Application Binary Interface (Aplikační binární rozhraní)

VM Virtual Machine (Virtuální stroj)

DRM Digital Rights Management (Správa digitálních práv)

SWEBOK Software Engineering Body of Knowledge

DBI *Dynamic Binary Instrumentation (Dynamická binární instrumentace)*

ABI *Aplikační binární rozhraní*

AI *Artificial Intelligence*

API *Application Programming Interface*

ARM *Advanced RISC Machine*

ARM64 *ARM 64-bit*

AST *Abstract Syntax Tree*

BNIL *Binary Ninja Intermediate Languages*

CFG *Control Flow Graph*

CFGED *Control Flow Graph Edit Distance*

CLI *Command Line Interface*

CTF *Capture The Flag*

DLL *Dynamic Link Library*

DRM *Digital Rights Management*

DWARF *Debugging With Attributed Record Formats*

ELF *Executable and Linkable Format*

EXE *Executable File*

GPL *GNU General Public License*

GPLv3 *GNU General Public License version 3*

HLIL *High-Level Intermediate Language*

IAT *Import Address Table*

ICS *Industrial Control Systems*

IL *Intermediární Jazyk (Intermediate Language)*

IoT *Internet of Things*

ISP *In-System Programming*

JTAG *Joint Test Action Group*

KDM *Knowledge Discovery Metamodel*

LGPLv3 *GNU Lesser General Public License version 3*

LLM *Large Language Models*

MDE *Model-Driven Engineering*

MDRE *Model-Driven Reverse Engineering*

MIPS *Microprocessor without Interlocked Pipeline Stages*

MIT *Massachusetts Institute of Technology*

ML *Machine Learning*

MMIO *Memory-Mapped I/O*

MSVC *Microsoft Visual C++*

NSA *National Security Agency*

OEP *Original Entry Point*

OS *Operating System*

PPC *PowerPC*

PDB *Program Database*

PE *Portable Executable*

RE *Reverzní Inženýrství*

RISC-V *RISC Five*

RTTI *Run-Time Type Information*

SDK *Software Development Kit*

SEH *Structured Exception Handling*

SMT *Satisfiability Modulo Theories*

STL *Standard Template Library*

TTD *Time Travel Debugging*

UI *User Interface*

UML *Unified Modeling Language*

VM *Virtual Machine*

XAI *Explainable Artificial Intelligence* Fragment kódu

AI Artificial Intelligence (Umělá inteligence)

API Application Programming Interface (Rozhraní pro programování aplikací)

CFG Control Flow Graph (Graf řídicího toku)

CLI Command Line Interface (Rozhraní příkazového řádku)

CTF Capture The Flag (Soutěž typu "Získej vlajku")

DFG Data Flow Graph (Graf datového toku)

DIE Detect It Easy (Nástroj pro detekci typu souboru/packerů)

DLL Dynamic Link Library (Dynamicky linkovaná knihovna)

ELF Executable and Linkable Format (Spustitelný a linkovatelný formát)

GCC GNU Compiler Collection (Kolekce kompilátorů GNU)

GDB GNU Debugger (GNU ladicí program)

GEF GDB Enhanced Features (Rozšířené funkce GDB)

GUI Graphical User Interface (Grafické uživatelské rozhraní)

IAT Import Address Table (Importní tabulka adres)

LLM Large Language Model (Velký jazykový model)

LTO Link-Time Optimization (Optimalizace v čase linkování)

ML Machine Learning (Strojové učení)

MSVC Microsoft Visual C++ (Kompilátor C++ od Microsoftu)

OOP Object-Oriented Programming (Objektově orientované programování)

OS Operating System (Operační systém)

PEB Process Environment Block (Blok prostředí procesu)

PIC Position Independent Code (Pozičně nezávislý kód)

RE Reverse Engineering (Reverzní inženýrství)

RTTI Run-Time Type Information (Informace o typu za běhu)

STL Standard Template Library (Standardní knihovna šablon)

UI User Interface (Uživatelské rozhraní)

UX User Experience (Uživatelský prožitek / Uživatelská přívětivost)

VM Virtual Machine (Virtuální stroj)

vCPU Virtual Central Processing Unit (Virtuální centrální procesorová jednotka)

WPO Whole Program Optimization (Optimalizace celého programu)

SEZNAM PŘÍLOH

PŘÍLOHA P I: Struktura projektových složek

PŘÍLOHA P I: STRUKTURA PROJEKTOVÝCH SLOŽEK

Tato příloha uvádí přehled souborů v přiloženém zip souboru:

- bninja_projects
- fuzzing_try
 - input (složka)
 - run1 (složka)
 - run2 (složka)
 - aflccOut.txt
 - target (soubor bez přípony)
 - target.c
- ghidra_projects
- screenshots_all
- scripting
 - ghidra_scripts-main.zip
 - readme.txt
 - xor_string_deobfuscator.py
- sources
 - 0_bbbbloat
 - 1_geometry_master
 - 2_shell_code
 - 3_win7calc