Fuzz testování embedded software

Pavlína Kulhavá

Bakalářská práce
2022

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2021/2022

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Pavlína Kulhavá**
Osobní číslo:	**A19056**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Fuzz testování embedded software**
Téma práce anglicky:	**Fuzz Testing of Embedded Software**

Zásady pro vypracování

1. Zpracujte literární rešerši na téma fuzzing, fuzzery a využití genetických algoritmů ve fuzzingu a fuzzing frameworcích.
2. Analyzujte a popište vybrané fuzzing frameworky.
3. Vyberte projekt z prostředí MCUXpresso SDK a zpracujte pro něj strategie fuzzingu se zvoleným fuzzerem.
4. Implementujte navržené strategie na vybraný projekt.
5. Zhodnoťte dosažené výsledky a navrhněte další postupy pro fuzz testování.

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. HARPER, Allen, Daniel REGALADO, Ryan LINN, et al. Gray Hat Hacking: The Ethical Hacker's Handbook. 5th Edition. New York: McGraw-Hill Education, 2018. ISBN 978-1-26-010842-2.
2. KIM, Peter. THE HACKER PLAYBOOK 2: Practical Guide To Penetration Testing. South Carolina: Secure Planet, 2015. ISBN 978-1512214567.
3. SUTTON, Michael, Adam GREENE a Pedram AMINI. Fuzzing Brute Force Vulnerability Discovery. Indiana: Addison Wesley, 2007. ISBN 0-32-144611-9.
4. TAKANEN, Ari, Jared DEMOTT, Charlie MILLER a Atte KETTUNEN. Fuzzing for Software Security Testing and Quality Assurance. 2nd Edition. Massachusetts: Artech House, 2018. ISBN 978-1608078509.
5. ZELLER Andreas, GOPINATH Rahul, BOHME Marcel, FRASER Gordon, HOLLER Christian. The Fuzzing Book [online]. CISPA Helmholtz Center for Information Security, 2021 [cit. 2021-11-29]. Dostupné z: https://www.fuzzingbook.org/

Vedoucí bakalářské práce: **Ing. Jan Dolinay, Ph.D.**
Ústav automatizace a řídicí techniky

Datum zadání bakalářské práce: **3. prosince 2021**
Termín odevzdání bakalářské práce: **23. května 2022**

doc. Mgr. Milan Adámek, Ph.D. v.r.
děkan

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

Ve Zlíně dne 24. ledna 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou 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;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské 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 bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 16. 5. 2022

Pavlína Kulhavá, v.r
podpis studenta

ABSTRAKT

Cílem této práce bylo analyzovat volně dostupné fuzz frameworky, zjistit, zda je možné použít tyto frameworky při testování embedded softwaru a navrhnout možné strategie pro jejich fuzz testování. Analýza fuzz frameworků proběhla po teoretické stránce a jednalo se čistě o zjištění co jaký framework umí a jaké jsou jeho výhody a nevýhody. Pro fuzz testování byl vybrán volně dostupný projekt ze stránek MCUXpresso SDK, na kterém došlo k implementaci dvou navržených strategií pro fuzz testování. Z obou strategií nakonec vyšla úspěšnější druhá strategie, ve které došlo k separaci funkce z kódu a otestování ji samostatně místo toho, aby došlo k otestování celého programu, o což se snažila strategie číslo jedna.

Klíčová slova:

Fuzz, fuzz testování, fuzzery, emebeded software, fuzzing frameworky

ABSTRACT

The aim of this work was to analyze freely available fuzz frameworks, see if it's possible to use these frameworks on testing embedded software and design possible strategies for their fuzz testing. Analysis of fuzz framework was on the theoretical side and it was purely about finding out what frameworks can do and what are their advantages and disadvantages. For fuzz testing freely available project from MCUXpresso SDK website was selected. On this project two of the designed strategies were implemented. In the end a more successful strategy was the second one, in which the function was separated from the code and tested separately instead of testing the entire program what was that number one strategy trying to do.

Keywords:

Fuzz, Fuzz testing, Fuzzers, embedded software, fuzz frameworks

Tímto krátkým poděkováním bych ráda vyjádřila svůj dík panu Ing. Janu Dolinayovi, Ph.D za vedení práce, odborné rady a připomínky při psaní práce. Zároveň bych tímto chtěla poděkovat odbornému konzultantovi Ing. Samuelu Mudrikovi za ochotu zodpovědět každý dotaz a vedení v rámci praktické části práce.

Prohlašuji, že odevzdaná verze bakalářské/diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD ... 8
I TEORETICKÁ ČÁST ... 9
1 FUZZ TESTOVÁNÍ ... 10
1.1 HISTORIE ... 11
1.2 FÁZE FUZZ TESTOVÁNÍ ... 12
1.2.1 Identifikace cíle ... 13
1.2.2 Identifikace vstupů ... 13
1.2.3 Generování dat použitých pro Fuzz testování ... 13
1.2.4 Spuštění fuzz dat ... 13
1.2.5 Monitorování chování cíle ... 13
1.2.6 Zapsaní defektů ... 14
1.3 PROČ POUŽÍVAT METODU FUZZ TESTOVÁNÍ ... 14
1.4 VÝHODY A NEVÝHODY FUZZ TESTOVÁNÍ ... 15
2 VYUŽITÍ GENETICKÝCH ALGORITMŮ VE FUZZING FRAMEWORCÍCH ... 17
2.1 CO JSOU GENETICKÉ ALGORITMY ... 17
2.2 FÁZE GENETICKÉHO ALGORITMU ... 18
2.2.1 Inicializace ... 18
2.2.2 Funkce fitness ... 18
2.2.3 Výběr ... 18
2.2.4 Reprodukce ... 19
2.2.4.1 Křížení ... 19
2.2.4.2 Mutace ... 19
2.2.5 Nahrazení staré populace novou ... 19
2.2.6 Konvergence aneb ukončení algoritmu ... 20
2.3 POUŽITÍ VE FUZZ FRAMEWORCÍCH ... 20
3 NÁSTROJE PRO FUZZ TESTOVÁNÍ ... 21
3.1 DĚLENÍ FUZZ FRAMEWORKŮ ... 21
3.1.1 Mutující fuzzery ... 21
3.1.2 Generační fuzzery ... 21
3.1.3 Genetický/Evoluční fuzzer ... 22
3.2 LIBFUZZER ... 22
3.2.1 Výhody a nevýhody ... 25
3.3 AMERICAN FUZZY LOP++ ... 26
3.3.1 Výhody a nevýhody ... 29
3.4 BOOFUZZ ... 30
3.4.1 Výhody a nevýhody ... 32
3.5 CODE INTELLIGENCE FUZZ ... 33
3.5.1 Výhody a nevýhody ... 33
II PRAKTICKÁ ČÁST ... 35
4 STRATEGIE FUZZINGU PRO PROJEKT MAESTRO PLAYBACK Z PROSTŘEDÍ MCUXPRESSO SDK ... 36

4.1 Maestro Playback 36
4.2 Volba fuzz frameworku 36
4.3 Definice strategie pro testování 36
4.3.1 Strategie č.1: Testování celého binárního projektu 36
4.3.2 Strategie č.2: Vyseparování funkce z cílového programu 37
5 IMPLEMENTACE STRATEGIÍ V REÁLNÉM PROSTŘEDÍ 39
5.1 Příprava k implementaci strategií 39
5.1.1 Instalace AFL++ 39
5.1.2 Instalace Visual Studio Code 39
5.1.3 Instalace GCC ARM Embedded toolchain 40
5.1.4 Stažení projektu ze stránek MCUXpresso SDK 40
5.2 Implementace Strategie č. 1 41
5.2.1 Zhodnocení výsledků implementace strategie č.1 45
5.3 Implementace strategie č.2 46
5.3.1 Zhodnocení výsledků implementace strategie č.2 53
5.4 Návrh dalšího postupu pro fuzz testování 53
ZÁVĚR 56
SEZNAM POUŽITÉ LITERATURY 59
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK 64
SEZNAM OBRÁZKŮ 65
SEZNAM TABULEK 66
SEZNAM PŘÍLOH 67

ÚVOD

Fuzz testování (také označováno jako fuzzing) je čím dál oblíbenější technika testování softwaru, jenž spočívá v hledání chyb pomocí automatizovaného vkládání chybných, nebo polo poškozených dat na vstup [1]. Společnost Google uvádí že v roce 2020 pomocí fuzzingu odhalili přes 25 000 chyb v Google softwaru, např. Chrome, a cca 22 500 chyb ve více než 340 open source projektů propojených s OSS-Fuzz. [2]

Fuzz testování je v mnoha ohledech pohodlnější, levnější a někdy i rychlejší než ruční testování. Jinak řečeno tester nemusí vymýšlet různé vstupy, které by mohl programu poskytnout, stačí základní sada dat, kterou si fuzz framework, který celé fuzz testování automatizuje, sám nějakým způsobem poškodí nebo zmutuje.

V dnešní době existuje několik volně dostupných fuzz frameworků, také označovaných jako fuzzerů. Každý ze známějších fuzzerů se povětšinou zaměřuje hlavně na určitou oblast testování, například jeden fuzz framework se zaměřuje na otestování internetových protokolů a jiný se může zaměřovat na otestování správného ošetření vstupů při práci se soubory. Avšak v současné době neexistuje žádný volně dostupný fuzzer, který by se specializoval na testování softwaru pro embedded zařízení.

Je tedy otázka, zda je možné použít tyto volně dostupné fuzz frameworky, které nebyly primárně navrhnuté pro testování embedded softwaru, a zautomatizovat pomocí nich testování embedded softwaru.

Proto je jeden z cílů práce i navrhnutí a implementace možných strategií pro fuzz testování embedded softwaru za pomocí fuzz frameworku. Pokud by implementace těchto návrhů byla úspěšná je možné s volně dostupnými frameworky fuzz testovat embedded software, a kromě úspory času najít i chyby, které by mohly zůstat nenalezeny nebo by mohly být nalezeny až po nějaké době.

Každý fuzz framework má určité výhody oproti ostatním nebo v celkovém spektru, ale na druhou stranu má i nevýhody. Výhodami může být například rychlá a skoro okamžitá instalace nebo to, že fuzzer má vlastní docker image, což je soubor, který spouští kód uložený v Docker kontejneru. Docker kontejner je součást Dockeru který umožňuje právě automatizaci nasazení aplikací jako přenositelných samoobslužných kontejnerů, které můžou běžet například v cloudu [51]. Na druhou stranu jejich nevýhody povětšinou bývají téměř neexistující dokumentace. Další z cílů práce je proto tyto výhody a nevýhody zjistit a zaznamenat.

I. TEORETICKÁ ČÁST

1 FUZZ TESTOVÁNÍ

Fuzz testování je způsob negativního testování, jak objevit bezpečnostní zranitelnosti a chyby v softwarové aplikaci. V podstatě fuzzing spočívá v odesílání náhodných, nebo polonáhodných dat na vstup programu, s očekáváním odezvy, za účelem pádu aplikace a identifikací problému, který by nemusel být jinak na první pohled patrný. Jednoduše řečeno na vstup programu jsou posílaná nečekaná data a následně jsou zachytávané jakékoliv výjimky, které nastanou. [10] [6]

Fuzz testování se dá v základu rozdělit na Coverage-guided fuzz (fuzz řízeným pokrytím) a Behavioral fuzz (fuzz chování). Neboli podle toho, jestli máme nějaké očekávání, jak se má aplikace chovat, nebo chceme jenom zapříčinit neočekávané skončení aplikace neboli pád. [6]

Behavioral fuzz očekává, jak by se měla aplikace chovat a porovnává své očekávání s výsledkem získaným po přijetí náhodných dat na vstup. Pokud se očekávání neshoduje s realitou znamená to, že je to místo kde by potenciálně mohly být chyby nebo jiné bezpečnostní problémy softwaru. [6]

Naproti tomu Coverage-guided fuzz posílá náhodná data na vstup s jediným účelem, a to shodit software, nebo nějak jinak zamezit správnému fungování programu. [6]

Dále ale Fuzz testování můžeme dělit i podle toho co přesně testujeme anebo co posíláme na vstup. Zde můžeme Fuzz testování rozdělit do tří kategorií a to: fuzzování aplikace, protokolové fuzzování a fuzzing formátu souboru (File format fuzzing). [18]

U fuzzování aplikace se zaměřujeme na testování tlačítek nebo textových polích v uživatelském grafickém rozhraní nebo funkce programů příkazového řádku, a to většinou po stránce frekvence nebo rychlosti. Například mačkání tlačítka několikrát za sekundu. U textových polí například jejich naplnění velkým množstvím dat. [18]

Protokolové fuzování používáme právě proto, že veškerá přenášená data musí být ve specifickém formátu. Abychom mohli otestovat toto chování softwaru, při odeslání nesprávně formátovaného obsahu, použijeme tento typ fuzzingu. Jeho hlavním cílem je otestovat, aby odeslaný obsah nebyl náhodně považován za příkazy, které by mohly být následně provedeny na serveru. [18]

Poslední typ fuzz testování, ve formátu souborů, dovolují některé fuzz frameworky. Jednoduše řečeno fuzz frameworku poskytneme nepoškozený soubor, nebo sadu souborů, které si

on sám automaticky zmutuje a poškodí. Tyto upravené soubory pak zkouší posílat do cílové aplikace, které se s nimi snaží pracovat. Pokročilé verze umožňují testovat další implementované funkce jako například komprimace videí. [17][18]

1.1 Historie

Termín „fuzz“ poprvé použil v 80 letech profesor Barton Miller, jako označení pro náhodná, nestrukturovaná data. [7]

Ten se v roce 1988 snažil připojit dálkově ke kódu, přes vytáčecí připojení, během bouřky. Ta způsobovala rušení a pád programu. Tato myšlenka, že kód není schopen tolerovat externí rušení, ho inspirovala k zadání práce svým studentům na Univerzitě ve Wisconsinu, ve kterém studenti vytvořili základní fuzzer na ověření spolehlivosti unixových programu tím, že je zahltili náhodnými daty a monitorovaly jakékoliv pády. Došlo k zjištění, že při obdržení vstupu, který byl náhodně generován a nečekán došlo běžně k pádu programu. Miller a jeho výzkumný tým tyto pokusy opakovali následně každých pět let, bohužel vždy se stejným negativním výsledkem. [4][6][3]

První zmínky o metodě testování podobné Fuzz testování, se ale objevují už v roce 1980. Důkazem toho je nástroj zvaný The Monkey [5], vytvořený pro testování MacPaint a MacWrite pro Macintosh. Jednalo se o malé příslušenství, které používalo takzvané journaling hooks k podávání náhodných událostí do aktuální aplikace. Docházelo k dojmu, že Macintosh ovládá „naštvaná“ opice, která mlátí do klávesnice a myši nebo kliká na náhodná místa, včetně zavření okna aplikace. Později došlo i k rozšíření náhodně generovaných akcích, a to například o příkazy v menu anebo posun okna aplikace. [3]

Avšak v praxi se softwarovému testování pro bezpečnost a spolehlivosti dává větší důraz až později v roce 1990. Jeden z možných důvodů, proč se pády aplikací braly do té doby jako omluvitelný problém, mohl být ten, že před příchodem veřejné sítě nebo internetu, nebyla žádná jasná představa o útočníkovi. Zrod softwarové bezpečnosti se připisuje široce rozsáhlým útokům na přetečení vyrovnávací paměti, jako například jednomu z prvních červů distribuovaných přes internet: The Morris Worm v roce 1988. [3][8]

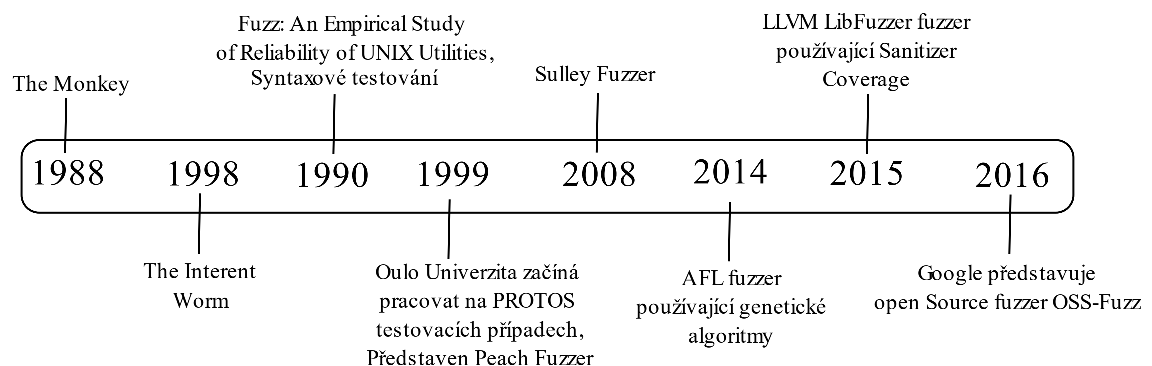
V roce 1990 publikuje profesor Barton Miller společně s univerzitou Wisconsin publikaci: Fuzz: An Empirical Study of Reliability of UNIX Utilities, kde dochází k popsání získaných výsledků při prvním použití fuzz techniky testování. Ve stejném roce, skoro ve stejný čas, ale dochází i k definici syntaxového testování, které bylo pravděpodobně vytvořeno, aby

řešilo stejný problém jako fuzz testování, tj. ověřovat, zda aplikace umí tolerovat, nebo i přijmout neplatná data poslaná na její vstup a poslání takových neplatných dat nezapříčiní její pád či nějak jinak neovlivní její chod. [3][9]

Okolo roku 1999 na univerzitě v Oulu začíná práce na jejich PROTOS testovacích případech. Tyto testovací případy, které byly navrhnuty tím, že prvně došlo k analýze specifikace protokolů a následně produkcí paketu, který buď porušil onu specifikaci nebo s ním nebylo zacházeno podle specifikací protokolu. Sice vytváření takovýchto testovacích případů zabralo dost času, na druhou stranu mohlo dojít k jejich opětovanému spouštění na několika různorodých projektech od různých výrobců. PROTOS kombinuje black box a white box přístup k testování a zároveň se jedná o jeden z velkých milníku vývoje Fuzz testování, právě kvůli objevení množství chyb, které by jinak mohli zůstat neodhaleny. [10]

V následujících letech proto nastal větší zájem o vývoj fuzz frameworků, taktéž označovaných jako fuzzerů, a jejich vylepšování. Například umožněním File format fuzzingu.

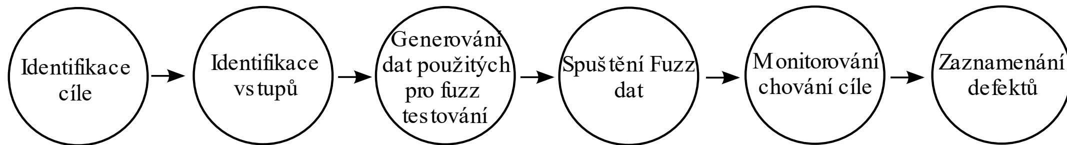
Začaly vznikat jak open source fuzz frameworky jako například OSS-Fuzz [11] nebo AFL [12], ale také komerčních do nedávné doby například The Peach protocol fuzzer [13], ten byl ale v roce 2020 odkoupen firmou GitLab, který z něj taktéž udělala volně dostupná fuzz framework [14].



Obrázek 1. Vývoj Fuzz testování

1.2 Fáze Fuzz testování

V závislosti na mnoha faktorech (cíl Fuzz testování, dovednosti testera, formát fuzzovaných dat) může být volba fází použitých pro Fuzz testování různá. Avšak existuje několik fází, které by měly být absolvovaný. [10]



Obrázek 2. Fáze Fuzz testování

1.2.1 Identifikace cíle

Identifikace cíle je důležitý krok při Fuzz testování. Bez znalosti cíle nelze totiž pořádně ani určit jaký nástroj pro Fuzz testování bude nejlepší využít, či jaká technika a taktika bude nejvíce efektivní. [10]

1.2.2 Identifikace vstupů

Další krok je zjistit všechny možné vstupy, které je potřeba otestovat. Vlastně cokoliv, co je klientem posláno do cíle by mělo být považováno za nějaký vstupní vektor. Vstupní vektor může obsahovat hlavičky, jména souborů, názvy proměnných, klíče a mnoho dalších věcí. [10]

1.2.3 Generování dat použitých pro Fuzz testování

Jakmile identifikujeme všechny možné vstupy, které můžeme otestovat, je potřeba pro tyto vstupy vygenerovat nějaké data, která na ně pošleme. Můžeme si vygenerovat svoji sadu dat, zmodifikovat již před vytvořená data nebo dynamicky vytvořit náhodnou sadu dat. Tato část by měla být nejlépe zautomatizována. [10]

1.2.4 Spuštění fuzz dat

V této fázi Fuzz testování se ze slova fuzz stává sloveso. Pod slovem spuštění se může schovávat posílání vygenerovaných fuzz dat na vstup, ale také otevírání souboru nebo i spuštění cíleného procesu. Tato fáze by měla být co nejvíce zautomatizována. Pokud by tak nebylo, je na místě uvažovat, jestli se pořád bavíme o Fuzz testování. [10]

1.2.5 Monitorování chování cíle

Při spuštění Fuzz testování je důležité monitorovat cíl po celou dobu probíhání testů. Je totiž neužitečné, když na vstup přijdou data, která vyvolají výjimku, nebo nějak jinak naruší běh cíle, ale nelze určit při jakých datech k tomu zrovna došlo. Monitorování může probíhat v hodně formách, vždy totiž záleží na tom, jaký cíl zrovna Fuzz testujeme a jakým stylem. [10]

1.2.6 Zapsaní defektů

Finální fáze jak Fuzz testování, tak i normálního testování, je zapsaní nalezených defektů. Většina fuzz frameworků má tuto část automatizovanou, automaticky tedy zapisují, na čem došlo k chybě a některé i v jaké části došlo k pádu, nebo vyvolání výjimky. [10]

1.3 Proč používat metodu Fuzz testování

Jak již bylo psáno v úvodu pomocí Fuzz testování google odhalil u svého distribuovaného softwaru přes cca 25 000 chyb [2]. Další zdroje, například Code Intelligence ve svém propagačním materiálu uvádí, že společnost Google najde pomocí Fuz testování okolo 80 % chyb ve svých kódech. [15]

V roce 1995, tedy pět let poté, co profesor Barton Miller poprvé popsal výsledky získané při prvním Fuzz testování, svůj pokus, spolu s kolektivem, zopakoval. Kromě zhodnocení výsledků popsal do této publikace i čtyři nejčastější příčiny pádů aplikací viz tabulka č.1. Ve výsledcích testů dominovali chyby v použití ukazatelů a indexů polí. Vesměs šlo o chyby, kdy programátor učinil implicitní předpoklady o obsahu zpracovaných dat, ale nedošlo k dostatečné kontrole, jestli data opravdu splňují předpoklady o obsahu. [16]

Tabulka 1. Nejčastější chyby nalezené s použitím Fuzz testování v roce 1995[16]

1	Chyby v použití ukazatelů a indexů u polí. Nejčastější chyba, programátor učinil předpoklad o obsahu zpracovaných dat, ale nepodrobil data kontrole, jestli jeho předpoklad opravdu splňují.
2	Nebezpečné používání vstupních funkcí. Například funkce gets(), která nemá parametr, který by definoval maximální délku vstupních dat.
3	Znaky se znaménkem. Obzvlášť převod čísel z jedné velikosti na druhou může být problémový, zvlášť problém použitím znaků v jejich obou formách jak symbolické i tak i číselné. Prezence znakového bitu může být totiž matoucí a náchylná k chybám při arytmických operacích.
4	End-of-File (EOF) kontrola. Jedna z dalších chyb, kde programátor učinil předpoklad o struktuře vstupních dat. Je sice obvyklé, ale taky nebezpečné, že konec souboru (EOF) nastane až po úplném zadaní řádku. Jinak řečeno že na konci souboru bude vždy následovat znak pro nový řádek. Tento předpoklad sice může

	zjednodušit strukturu kódu, ale aplikaci ponechá zranitelnou vůči zhroucení nebo zablokování.

Přestože testování probíhalo mezi lety 1990 a 1995 je jistá šance, že výše zmíněné chyby se mohou vyskytovat i v dnešních aplikacích a mohou být právě díky Fuzzingu odhaleny.

Jeden z dalších důvodu, proč je dobré využívat Fuzz testování je i ten fakt, že fuzzování dat je oblíbená metoda mezi hackery. Automatické posílání náhodně generovaných dat přímo láka k využití k dostání do systému, nebo podobně. Jedná se také o řešení, které má dobrý poměr cena ku výkonu, aby došlo k nalezení nebezpečných bezpečnostních chyb.[17]

1.4 Výhody a nevýhody Fuzz testování

I když je Fuzz testování velmi užitečné není to všelék, který by vyřešil, v tomto případě našel, každý problém v softwaru. A stejně jako některé jiné metody testování není vhodný na všechno.

Tabulka 2. Výhody a nevýhody fuzz testování [15] [18]

Výhody	Nevýhody
Zvyšuje bezpečnost softwaru.	Méně efektivní, pokud má, co dočinění s něčím, co nezaviňuje pád aplikace, například viry nebo červy.
Pomáhá k nalezení vážných bezpečnostních problémů jako je například memory leak, nebo neošetřené výjimky.	Časově nevýhodné řešení. Fuzzer může běžet klidně i měsíce, než možná narazí na chybu.
Skoro kompletně automatizovaná testovací technika.	Open-source fuzzing nástroje vyžadují ve větší míře většího úsilí k získání efektivního výsledku testování.
I když se jedná o časově nevýhodné řešení na druhou stranu, pokud dojde ke spuštění Fuzz testování může běžet i měsíce bez toho, aniž by to toho bylo potřeba zasahovat manuálně.	Integrace fuzz technologií do procesu vývoje vyžaduje znalosti v oblasti testování a bezpečnosti.

Najde chyby a zranitelnosti, které by bylo obtížné, nebo i nemožné najít pomocí jiných technik například Unit testů.	
Prakticky nedochází k vytváření žádných falešných poplachů. Pokud fuzzer něco najde, jedná se o potvrzený problém.	
Poskytuje celkový obraz o testovaném softwar.	
Vysoce škálovatelná testovací technologie. Zdrojový kód lze testovat současně.	

2 VYUŽITÍ GENETICKÝCH ALGORITMŮ VE FUZZING FRAMEWORCÍCH

Občas nás nezajímá pouze fuzz testování co nejvíce možných vstupů programu, ale také i odvození co nejvíce specifických testovacích vstupů dosahující nějakého cíle, například určitého stavu programu. Vyhledávací algoritmy jsou sice jádrem počítačové vědy, ale použít v tomto případě klasických vyhledávacích algoritmů, které například prohledávají nejdříve šířku nebo hloubku, je neefektivní a nerealistické. Tyto algoritmy totiž vyžadují vidět všechny možné vstupy a tento požadavek nemůžeme nikdy dokonale splnit. Avšak k vyřešení tohoto problému nám může pomoc heuristika, pokud totiž zvládneme odhadnout například který z několika různých vstupů je blíže k tomu co hledáme, můžeme dosáhnout cíle rychleji, než kdybychom zkoušeli postupně všechny vstupy. [7]

2.1 Co jsou genetické algoritmy

Genetické algoritmy jsou algoritmy používané k řešení optimalizačních problému ve strojovém učení. Tyto algoritmy řeší takové problémy, u kterých by jiné řešení trvalo příliš dlouho. [19]

Vznikly na základě přirozeného evolučního procesu v přírodě a jsou součástí optimalizačních algoritmů, které se používají k nalezení nejlepšího řešení ze všech dostupných možných řešení za stávajících omezení. Používají základní myšlenku přirozeného výběru, silnější přežije, a genetické dědičnosti. Využívají, na rozdíl od jiných algoritmů, takzvané řízené náhodné prohledávání. [20]

Existuje základní terminologie genetických algoritmů, která nám může pomoc porozumět, jak také algoritmy fungují. Do této terminologie patří výraz populace, který označuje podmnožinu všech pravděpodobných řešení, která mohou vyřešit daný problém. Dále chromozomy, které se často zobrazují binárně pomocí 0 a 1 ale přístupné jsou i jiné možnosti [20], představují jedno z řešení v populaci, naopak gen je jeden prvek v chromozomu. Každý gen, který se nachází v konkrétním chromozomu, má přiřazenou svou vlastní hodnotu, tomu se říká Alela. Funkce fitness je funkce, která pomocí specifického vstupu vytváří vylepšený výstup, řešení se poté používá jako vstup, zatímco výstup je ve formě vhodnosti řešení. Poslední, co je potřeba znát je termín Genetičtí operátoři. V genetických algoritmech totiž dochází k tomu, že se nejlepší jednici spáří, aby reprodukovali potomstvo, které bude lepší než

jeho rodiče. Genetické operátory se používají právě pro změnu genů této příští generace. [19]

2.2 Fáze genetického algoritmu

Genetické algoritmy využívají evoluční generační cyklus k tomu, aby dosáhly vysoce kvalitních řešení. Pomocí různých operacích, které zvyšují nebo nahrazují populaci, k vylepšení a nalezení nejlépe pasujícího řešení. [19]

2.2.1 Inicializace

Každý genetický algoritmus jako první vygeneruje počáteční populaci. Ta se skládá ze všech pravděpodobných řešení daného problému. Nejoblíbenější technikou pro počáteční vygenerování populace jsou náhodné binární řetězce. [19]

2.2.2 Funkce fitness

V této fázi dochází k přiřazení fitness hodnoty všem chromozómům v populaci. [20]

K zjišťování zdatnosti všech jedinců v populaci nám pomáhá funkce fitness. Ta každému jedinci přiřadí skóre zdatnosti, toto skóre také určuje pravděpodobnost, že jedinec bude vybrán k reprodukci. Obecně platí, že čím vyšší skóre zdatnosti tím vyšší šance k vybrání pro reprodukci. [19]

2.2.3 Výběr

V této fázi dochází k výběru jedinců pro reprodukci potomstva. Vybraní jedinci jsou následně uspořádaní do párů, aby se zlepšila reprodukce. Tito jedinci předávají své geny další generaci. [19]

Hlavním cílem je vytvořit region, který bude mít vysokou šanci vygenerovat nejlepší řešení problému, to znamená řešení lepší, než s jakým přišla předchozí generace. Používá se technika proporcionálního výběru způsobilosti. Ta zajistí že užitečná řešení budou při rekombinaci použita. [19]

Technika proporcionálního výběru se používá právě proto, že výběr příliš silných jedinců může vést k suboptimálním řešením. Naopak výběr příliš slabých jedinců vede k nesoustředěnému vyhledávání. [20]

2.2.4 Reprodukce

V této fázi dochází k vytvoření potomka. Algoritmus využívá variační operátory, které aplikuje na populaci rodičů. Dva hlavní operátory v této fázi jsou: mutace a křížení. [19]

Nově vytvoření potomci jsou přidáni do populace. [20]

2.2.4.1 Křížení

Jedná se o nejdůležitější fázi genetického algoritmu. [20]

Během této fáze je náhodně vybrán rodičovský pár, u kterého se vymění genetická informace, za účelem reprodukce potomka. Vytvoří se populace potomků, která je stejně velká jako velikost populace rodičů. [19]

Křížení se dá rozdělit na tři základní typy.

Jednobodové křížení, je takové křížení, při kterém je vybrán náhodně bod na chromozomech obou rodičů a je označen jako „bod křížení“. Následně se vyměňují bity vpravo, a to mezi dvěma rodičovskými chromozomy. [20]

Dvoubodové křížení náhodně vybere dva body z rodičovských chromozomů. Z toho logicky vyplývá, že dochází k výměně bitů, které se nachází mezi těmito body. [20]

Poslední typ křížení se nazývá jednotné křížení. U tohoto typu křížení je obvyklé, že se vybere každý bit od obou rodičů se stejnou pravděpodobností. [20]

2.2.4.2 Mutace

V několika nově vytvořených potomcích může dojít, s nízkou náhodnou pravděpodobností, k mutaci. Znamená to, že může dojít k převrácení některých bitů v bitovém chromozomu. Jinak řečeno na pozici, kde se předtím nacházelo nula se může po mutaci nacházet hodnota jedna a naopak. Mutace se postará o diverzitu mezi populacemi a zastaví předčasnou konvergenci neboli totožností jedinců. [20]

2.2.5 Nahrazení staré populace novou

Jakmile dojde ke spáření rodičů a vytvoření nové generace, pomocí křížení a mutací, nahradí se staré generace novou. Nová generace by měla dosahovat lepšího a vyššího fitness skóre, což značí vylepšení generovaného řešení. [19]

2.2.6 Konvergence aneb ukončení algoritmu

Jakmile dojde u potomků k jasně viditelné konvergenci anebo akceptovanému řešení, které může být dáno například určením maximální hranice fitness bodů, může dojít k zastavení genetického algoritmu. K zastavení může dojít i pokud jsme dosáhli časového limitu, nebo maximálního počtu generací. [19] [20]

Genetický algoritmus můžeme zastavit klidně i po prvním cyklu, ale šance že dostaneme to nejlepší řešení, které hledáme, je nízká.

2.3 Použití ve fuzz frameworcích

Genetický algoritmus nemusí být součástí fuzz frameworků ani použit v rámci fuzz testování. Jedná se pouze o určité vylepšení, které ale může zvýšit šance na nalezení chyby, nebo nalezení chyby v programu urychlit.

Fuzz frameworkům, které právě využívají genetické algoritmy se říká genetické/evoluční fuzzery (Genetics/Evolutionary fuzzers). [22]

Implementace genetických algoritmů do fuzz frameworků není sice povinné, ale dosti časté, protože oproti čistým black box fuzzerům přináší zásadní výhodu. Black box fuzzery totiž postrádají jednu důležitou věc a tou je zvolení vstupu v závislosti na zpětné vazbě týkající se pokroku v rámci testované programové logiky. Příkladem může být paket přijatý přes síťové připojení aplikací, který je následně odeslán do nějaké API funkce, o které je ale známo, že u ní může dojít k přetečení vyrovnávací paměti. Avšak abychom mohli říct, že tato zranitelnost je zneužitelná hrozba musíme dokázat, že se k ní dá dostat přes vstupy od uživatele. [21]

V roce 2007 Sparks et al. provedl výzkum, kde implementoval genetický algoritmus, který se zaměřoval na prozkoumávání méně běžných bloků kódu. Na začátku došlo k analýze uzlů a pravděpodobnosti přechodu mezi bloky. Jinak řečeno, jaká je šance, že se z bloku A dostanu do bloku B. Poté byl do aplikace poslán vstup, a z cesty, kterou vstup ušel během vykonávání, byla analyzována funkce fitness. Velikost funkce určovala, jak moc se vstup dostal do zřídka spuštěných větví programu. Tedy čím vyšší byla velikost funkce fitness tím více se vstup dostal do zřídka pouštěných větví. Jak se dalo očekávat, předváděna technika gray box fuzzeru, dosáhla vyššího pokrytí a dostala se mnohem rychleji a hlouběji do uzlů než čistý black box fuzzer. [21]

3 NÁSTROJE PRO FUZZ TESTOVÁNÍ

V dnešní době se na trhu nachází nespočet nástrojů pro fuzz testování, označovány taktéž jako fuzz frameworky nebo fuzzery. Některé z nich jsou open source jiné naopak komerční. Fuzz frameworky usnadňují fuzz testování softwaru a celý proces sami zautomatizují a zjednoduší.

Fuzz frameworky obvykle zautomatizují fáze 3 až 6, viz Obrázek 2. Některé například potřebují, ale nějaký počáteční nepoškozený vstupní soubor, který by mohli dále mutovat. Jiným stačí napsat například pouze funkci, ve které se specifikuje, co se má stát s daty, a fuzzer se sám postará o zbytek procesu, třeba tím že v případě zjištění chyby na tento fakt upozorní a uloží jaká data byla v tom momentě na vstupu aplikace.

3.1 Dělení Fuzz frameworků

Fuzzery se dají rozdělit podle několika kritérií do několika kategorií. Jedno ze základních dělení je, jestli jsou nástroje dostupné ve formě open source nebo jestli se jedná o komerční projekty. Většina nástrojů pro fuzz testování spadá do kategorie open source.

Hlavní dělení ale spočívá v tom, jakým způsobem modifikuje fuzz framework data.

Zde můžeme takový fuzzer rozdělit do tří kategorií: Mutující fuzzer (Mutation Fuzzer), Generační Fuzzer (Generation Fuzzer) a Genetický/Evoluční fuzzer (Genetic/Evolutionary fuzzers). [22]

3.1.1 Mutující fuzzery

Těmto fuzzerům se také říká hloupé fuzzery, anglicky dumb fuzzers. Jsou to fuzz frameworky, které jsou nejblíže původní myšlence randomizace vstupních dat a zároveň jsou nejjednodušší variantou. Název pochází ze změny (mutace) dat, většinou náhodným způsobem, tyto zmutována data se následně použijí jako vstup pro cílený software za účelem jeho shození. [22]

3.1.2 Generační fuzzery

Generační fuzzery se také často nazývají jako white box fuzz testování nebo fuzz testování založené na gramatice, z toho důvodu že předem známe vnitřní fungování protokolu. To znamená, že hlavní předpoklad je, že známe a chápe vnitřní fungování cíle. Jinak by mohlo být testování neúčinné. [22]

Generační fuzzery nepotřebují příklady platných vstupních dat nebo protokolu na rozdíl od fuzzerů založených na mutaci. Jsou totiž schopný generovat testovací data v závislosti na datových modelech, které popisují strukturu dat nebo protokolu. Tyto modely bývají obvykle napsané jako konfigurační soubory a jejich formát se liší v závislosti na použitých nástrojů pro fuzz testování. [22]

Z toho vyplývá i hlavní problém generačních fuzzerů a to je právě psaní datových modelů. Problém to sice není pro jednoduché protokoly nebo datové struktury, které mají k dispozici dokumentaci, avšak takové případy jsou vzácné a kvůli jednoduchosti nejsou tak zajímavé. Ve skutečnosti je to mnohem složitější a dostupnost specifikací a dokumentace stále vyžaduje značné úsilí pro správný převod do fuzzing modelu. Věci se mohou ještě zkomplikovat, když softwarová společnost nedodržuje specifikace a mírně si je upraví nebo dokonce zavede nové funkce, které nejsou zmíněny ve specifikacích. V takových případech je pak nutné upravit model, tak aby pasoval na cílený software, což vyžaduje další práci a úsilí. [22]

3.1.3 Genetický/Evoluční fuzzer

Genetickému fuzzeru se taktéž říká evoluční, protože určí tu nejlepší sadu vstupních testů. To celé zakládá na maximalizaci pokrytí kódu v průběhu času. Fuzzer vlastně upozorní na vstupní mutace, které dosáhnou nových bloků v kódu a tyto vstupní mutace uloží do těla testů. Tímto způsobem se může nástroj pro fuzz testování učit způsobem přežití nejschopnějších. Tedy proto termín genetický nebo evoluční fuzz testování.

3.2 LibFuzzer

LibFuzzer je fuzzer, který je součástí knihovny LLVM (vzhledem k tomu že je celá knihovna volně dostupná je i fuzzer volně dostupný), je tedy pro práci s ním potřeba nainstalovat celou tuto knihovnu, hlavně tedy kompilátor Clang, zároveň ale není potřeba instalovat další knihovny, jelikož libFuzzer pracuje s knihovnami obsaženými už v LLVM knihovně. Jednou takovou knihovnou, s kterou libFuzzer pracuje, je například SanitizerCoverage instrumentation, který se stará o informace o pokrytí kódu testy. [23]

Jedná se o průběžný, pokrytím řízený evoluční fuzzer. [23]

Dříve byl podporován pouze pro systémy s Linux nebo MacOS, ale od Clang verze 9.0 podporuje i Windows, avšak nepodporuje použití bez ASANu. [23]

LibFuzzer pracuje s funkci zvanou fuzz target, která přijme pole bytu a následně s nimi provede to co je definováno v těle funkce. Fuzz target je v základě tedy extérní funkce libFuzzeru, přes kterou dostaneme data na vstup. V těle je následně definováno, co se bude s daty dále dít. [23]

Je důležité vědět, že fuzzer bude spouštět fuzz target několikrát pokaždé s jinými vstupy, z toho vyplývá že musí tolerovat jakékoliv vstupy (velké, prázdné, poškozené a další) zároveň, ale nesmí vystoupit z funkce na žádném vstupu. [23]

Fuzz target může používat vlákna, ideálně avšak tak aby všechna vlákna byla spojena na konci funkce. Měla by být taky co nejvíce deterministická, protože nedeterminismus (Například náhodná rozhodnutí, která nebudou založena na vstupu) může způsobit neefektivnost fuzzingu. V ideálním případě by nemělo docházet k změně jakéhokoliv globálního stavu, ale tato věc není striktní. [23]

V poslední řadě by funkce fuzz target měla být rychlá. Je dobré se vyhnout větší složitosti protokolování nebo nadměrné spotřebě paměti. Z toho vyplývá že obvykle platí čím užší cíl tím lepší. Pokud třeba budeme chtít u cíle analyzovat několik formátů dat, můžeme je rozdělit na několik menších cílů, vždy jeden pro každý formát. [23]

```
cmake -DLLVM_ENABLE_PROJECTS=clang -G "Unix Makefiles" ../llvm
```

Obrázek 3. Cmake pro build LLVM z dokumentace Clang [25]

K sestavení fuzzeru binárně se používá -fsanitize=fuzzer flag, ten se použije během kompilování a linkování. Ve většině případu chceme použít libFuzzer s kombinací ASAN (AddressSanitizer), UBASAN (UndefinedBehaviorSanitizer) nebo obojím. Můžeme také použít sestavní s MSAN (MemorySanitizer), zde je ale podpora pouze experimentální. [23]

ASAN rychle detekuje chyby v paměti. Umí detekovat například uniky v paměti nebo přístup mimo u haldy nebo zásobníku. [26]

UBSan rychle detekuje nedefinované chování. Umí detekovat například přetečení celého čísla se znaménkem nebo dolní index pole mimo meze, kde lze mez statisticky určit. [27]

MSAN je detektor neinicializovaných čtení. Při jeho použití dochází ke zpomalení. [28]

Níže na obrázku 4, 5 a 6 je ukázka použití libFuzzeru. Zde externí funkce přijímá pole bytu, s tím, že pokud se na vstupu objeví: „Hi!“ dojde k abnormálnímu pádu aplikace. Ve

výstupním souboru, který fuzzer vytvoří sám automaticky, se pak nachází data, která byla na vstupu v moment pádu aplikace. V tomto případě se v souboru crash bude nacházet slovo „Hi!“, je možné že za vykřičníkem budou následovat další znaky, protože extérní funkce padne jakmile první 3 data uložená v poli dají slovo Hi!.

```
cat << EOF > test_fuzzer.cc
#include <stdint.h>
#include <stddef.h>
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  if (size > 0 && data[0] == 'H')
    if (size > 1 && data[1] == 'I')
       if (size > 2 && data[2] == '!')
       __builtin_trap();
  return 0;
}
EOF
# Build test_fuzzer.cc with asan and link against libFuzzer.
clang++ -fsanitize=address,fuzzer test_fuzzer.cc
# Run the fuzzer with no corpus.
./a.out
```

Obrázek 4. Ukázkový příklad použití LibFuzzeru z oficiální dokumentace [23]

Po spuštění se po chvíli objeví výpis při detekci pádu aplikace, který vypadá následovně:

```
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 2577454258
INFO: Loaded 1 modules   (8 inline 8-bit counters): 8 [0x54bec0, 0x54bec8),
INFO: Loaded 1 PC tables (8 PCs): 8 [0x524070,0x5240f0),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger
than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
...
NOTE: libFuzzer has rudimentary signal handlers.
      Combine libFuzzer with AddressSanitizer or similar for better crash
reports.
SUMMARY: libFuzzer: deadly signal
MS: 1 InsertRepeatedBytes-; base unit:
253420c1158bc6382093d409ce2e9cff5806e980
0x48,0x49,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0
x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x
21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x21,0x2
1,0x21,0x21,0x21,0x21,
HI!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
artifact_prefix='./'; Test unit written to ./crash-
d2b543f8ac86df21b9a8aeb5ce58595f31428865
Base64: SEkhISEhISEhISEhISEhISEhISEhISEhISEhISEhISEhISEhISEhISEhISEhISE=
```

Obrázek 5. Výpis terminálu při detekci pádu aplikace

```
$ cat crash-d2b543f8ac86df21b9a8aeb5ce58595f31428865
HI!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Obrázek 6. Obsah souboru crash vytvořeného po pádu aplikace

3.2.1 Výhody a nevýhody

Od verze Clang 6.0 je libFuzzer jeho součástí a není potřeba žádných dalších instalací, stačí tedy nainstalovat pouze Clang.

Podporuje další knihovny užitečné při fuzzingu (ASAN, UBSan a MSAN).

Jakožto jeden z hlavních problému je právě instalace samotná, která na slabších zařízení nemusí probíhat jednoduše. Nejčastější problém při instalaci je nedostatečně velká paměť RAM. V základu se totiž všechny potřebné linky provádějí staticky. Tento problém se podle všeho dá vyřešit omezením paralelismu odkazů, snížením spotřeby paměti pomocí jednoho nebo obou linker flags, změna linkerů na Gold nebo lld anebo místo použití statických linků použít sdílené. [24]

LibFuzzer sice bude fungovat bez jakýchkoli počátečních ukázkových vstupů, ale bude méně efektivní, pokud testovaný subjekt přijímá složité strukturované vstupy. Proto je dobré použíti korpusu s ukázkovými vstupy pro testovaný kód. Takový korpus by měl obsahovat pestrou sbírku platných i neplatných vstupů. Například pro testování hudebního přehrávače můžeme tento soubor naplnit MP3, WAV a dalšími hudebními formáty. Fuzzer poté generuje náhodné mutace na základě těchto souborů. Pokud mutace spustí dříve nezjištěnou cestu uloží tuto mutaci do korpusu pro příští použití. [23]

LibFuzzer podporuje paralelní fuzz testování. Přestože každý proces libFuzzeru probíhá v jednom vlákně, pokud testovány objekt nezačne své vlastní vlákno, je stále možné nechat běžet několik procesů libFuzzeru paralelně s tím, že budou sdílet jeden korpusový adresář. To zajistí že jakýkoliv nový vstup nalezený při procesu jedním, bude dostupný ostatním, pokud ovšem nedojde k zakázání pomocí -reload=0. [23]

Vzhledem k tomu, že fuzz target není nějakým způsobem závislá na libFuzzeru, dá se použít společně s dalšími fuzzery. Je tak možné fuzz testovat program s pomocí libFuzzeru a AFL. Protože oba fuzzery očekávají, že je testovací korpus umístěn ve složce, dají se pustit tak že si pokaždé jeden vezme testovací soubor ze složky a po něm druhý. Na druhou stranu je

nelze zatím spustit paralelně tak aby mohli zároveň sdílet stejný adresář. Je také nutné je oba pravidelně restartovat, aby mohli požívat vzájemně své poznatky. [23]

Tabulka 3. Výhody a nevýhody libFuzzeru

Výhody	Nevýhody
Součástí knihovny, odpadá nutnost instalovat další potřebné knihovny pro práci.	Dlouhá a místy problémová instalace.
Spolupracuje s dalšími knihovnami (např. ASAN).	I když jde pustit s dalšími fuzzery, nelze spustit paralelně, vstupy z korpusu si berou na střídačku.
Fuzz target není závislá ne fuzzeru, možnost použít ji při použití jiného fuzzeru.	Při spuštění dvou různých fuzzerů, se stejným korpusem, nutno oba po čase restartovat, aby došlo ke sdílení vzájemných poznatků.
Dá se spustit společně s dalšími fuzzery, které pracují s korpusem.	
Paralelní fuzz testování, může běžet více libFuzzerů se sdíleným korpusem a vzájemně sdílet poznatky, nemusí tedy dojít k jejich restartu.	
Podpora Linux, MacOS, Windows.	
Obsáhla dokumentace.	

3.3 American fuzzy lop++

American fuzzy lop neboli AFL ++ je open source fuzzer vycházející z open source fuzzeru AFL (american fuzzy lop). [31]

Stejně jako AFL i jeho „dcera“ AFL++ používá vlastní kompilátor a genetické algoritmy k objevování čistých a zajímavých testovacích případů, které spouští nové stavy v cílovém systému. [29]

V dnešní době ale není AFL udržovaný, sám autor AFL, Michal Zalewski, na oficiálních stránkách upozorňuje na to, že AFL nebylo aktualizování už několik let, sice by stále mělo fungovat v pořádku, a doporučuje podívat se na AFL++. [29]

V roce 2021 google ve svém oss-fuzz nahradil AFL právě za AFL++. [35]

AFL++ má rychlou a jednoduchou instalaci, jak je ukázáno níže na obrázku 7, s možností nainstalování dalších přidružených knihoven. Jako například UNICORN nebo QEMU, které umožní testovat binární kód programu. [31]

```
$ git clone https://github.com/AFLplusplus/AFLplusplus
$ cd AFLplusplus
$ make distrib
$ sudo make install
```

Obrázek 7. Instalace AFL++ popsána na oficiálních stránkách [30]

Momentálně AFL++ plně podporuje Linuxové systémy, MacOS, Android a Solaris. V aktuální době neexistuje podpora pro Windows.

Níže na obrázku 8 vidíme ukázku konfigurace projektu a spuštění AFL++ při testování knihovny Libxml2. Pod ním na obrázku číslo 9 vidíme status screen s výpisem informací o průběhu fuzzingu v reálném čase. Je z něj možné vyčíst, jak dlouho se testuje, kolik bylo nalezeno chyb, kdy došlo k poslednímu pádu aplikace anebo i kolik cyklů testování bylo již vykonáno.

```
#vypnout sdílené knihovny
$ ./autogen.sh
$ ./configure --enable-shared=no
#povolit ASAN a UBSAN
$ export AFL_USE_UBSAN=1
$ export AFL_USE_ASAN=1
#sestavit knihovnu s použitím clang wrappers
$ make CC=~/AFLplusplus/afl-clang-fast CXX=~/AFLplusplus/afl-clang-fast++
LD=~/AFLplusplus/afl-clang-fast
#vytvoření složky odkud se bude brát vstup pro testování a ukládat výstup z
testování
$ mkdir fuzz
$ cp xmllint fuzz/xmllint_cov
$ mkdir fuzz/in
$ cp test/*.xml fuzz/in/
$ cd fuzz
#konfigurace systému přes skript před spuštěním AFL++
$ sudo ~/AFLplusplus/afl-system-config
#SPUŠTĚNÍ AFL++
#-i odkazuje na složku se vstupy, -o je složka pro ukládání výstupů, -m
znamená omezení paměti pro testování -m none je žádné omezení, toto omezení
je doporučováno při testování s pomocí ASANu. -d povolí FidgetyAFL, ty
přeskočí deterministický úsek a přeskočí rovnou na náhodný.
#Pokud bychom chtěli testovat binárně pomocí QEMU stačí zadat parametr -Q
$ ~/AFLplusplus/afl-fuzz -i in/ -o out -m none -d -- ./xmllint_cov @@
```

Obrázek 8. Ukázka nastavení projektu a spuštění AFL++, při testování knihovny Libxml2[37]

```
      american fuzzy lop ++4.01a {default} (./xmllint_cov) [fast]
 process timing                                   overall results
       run time : 0 days, 0 hrs, 0 min, 7 sec      cycles done : 0
  last new find : 0 days, 0 hrs, 0 min, 0 sec     corpus count : 438
last saved crash : none seen yet                 saved crashes : 0
 last saved hang : none seen yet                   saved hangs : 0
 cycle progress                      map coverage
  now processing : 23.0 (5.3%)          map density : 4.45% / 10.64%
  runs timed out : 0 (0.00%)         count coverage : 2.66 bits/tuple
 stage progress                       findings in depth
  now trying : havoc                 favored items : 33 (7.53%)
 stage execs : 4009/24.6k (16.31%)    new edges on : 227 (51.83%)
 total execs : 7661                  total crashes : 0 (0 saved)
  exec speed : 969.0/sec              total tmouts : 1 (1 saved)
 fuzzing strategy yields                            item geometry
   bit flips : disabled (default, enable with -D)    levels : 2
  byte flips : disabled (default, enable with -D)   pending : 438
 arithmetics : disabled (default, enable with -D)  pend fav : 33
  known ints : disabled (default, enable with -D) own finds : 396
  dictionary : n/a                                 imported : 0
havoc/splice : 0/0, 0/0                           stability : 100.00%
py/custom/rq : unused, unused, unused, unused
    trim/eff : 0.00%/146, disabled                     [cpu000: 33%]
```

Obrázek 9. Stavová obrazovka s informacemi o průběhu fuzz testování v aktuálním čase

3.3.1 Výhody a nevýhody

AFL++ má sice docela rozsáhlou dokumentaci, kde jsou vypsány všechny jeho funkce, na stranu druhou postrádá jednoduché vysvětlení co, jak a kdy používat.

Instalace je, na rozdíl do libFuzzeru, rychlá. Během instalace se dá vybrat, jestli se má nainstalovat celý AFL++ se všemi dostupnými nástroji, nebo jenom s některými. Například pro instalaci čistě nástrojů pro binární fuzzing stačí u sestavění specifikovat: make binary-only. Nejednoduší ale pořád zůstává nainstalovat vše s pomocí příkazu make distrib. Taktéž nabízí i možnost Dockerfile.[30]

Při sestavování používá AFL++ svůj vlastní kompilátor, což by mohl být problém, pokud chceme s jeho pomocí fuzz testovat něco, k čemu nemáme zdrojový kód.

Zvládá testování binárních kódu pomocí podporovaných knihoven. Například QEMU nebo Unicron_mode.[32]

Podporuje paralelní fuzz testování, s tím že každá kopie si vezme jedno jádru procesoru. To znamená, že teoreticky na zařízení, které má n jader procesu jsme schopni spustit n kopií. [33]

Má konzolové uživatelské rozhraní, takzvanou stavovou obrazovku, na které se v reálném čase zobrazují informace o probíhaném fuzz testování. Nachází se tam například jak dlouho je fuzzer spuštěný, kdy byla nalezena poslední nova cesta, kdy došlo k unikátnímu pádu aplikace ale třeba i počet cyklů nebo celkový počet nalezených tras. [34]

Podporuje fuzz testování s pomocí ASANu a UBSanu. U použití ASAN je dobré nenastavovat limit využití paměti, pokud by byl limit nastaven nemuselo by například vůbec dojít ke správném spuštění fuzzeru na cíl, ale také by se během běhu mohla vyskytnout chyba, která by ale nesouvisela s testovaným cílem, ale právě s tím, že ASAN si bere moc paměti. [36]

Tabulka 4. Výhody a nevýhody AFL++

Výhody	Nevýhody
Rychlá a jednoduchá instalace.	Vlastní kompilátor.
Podpora ASAN a UBSan.	Dokumentace postrádá vysvětlení, jak fuzzer správně používat.

Podpora paralelního fuzz testování.	Při paralelním fuzz testování si každá kopie vezme jedno jádro procesoru.
U instalace možnost specifikovat, jestli provést kompletní instalaci nebo jenom část pro specifické testování.	Zatím není podpora pro Windows.
Možnost testování binárních kódu.	
Podpora Android, Solaris, MacOS a Linux zařízení.	

3.4 BooFuzz

BooFuzz je fuzzer vycházející ze Sulley fuzzing framework, ten stejně jako u případu s AFL a AFL++ není už delší dobu pravidelně a často udržovaný. Boofuzz kromě opravy několika chyb v originálním Sulley fuzzing frameworku, ze kterého vychází, přidal i další funkce a zaměřuje se na rozšiřitelnost s heslem: „Fuzz everything.“. Stejně jako Sulley je i Boofuzz open source projekt. [38]

Sám autor na oficiální stránce projektu na Githubu píše, že pokud uživatelé hledají aktivně udržovaný fork Sulleyho mají zkusit Boofuzz. [39]

Jedná se o open source fuzzer, který se zaměřuje na testování internetových protokolů a síťových aplikacích. [38]

Stejně jako Sulley i boofuzz zahrnuje všechny kritické prvky fuzzerů: snadné a rychlé generování dat, detekci poruch, restartování cíle po selhání a záznam testovaných dat. Dále došlo k jeho vylepšení a rozšíření funkcí, takže zvládá: libovolná komunikační média, má vestavěnou podporu sériového fuzzingu, ethernetové a IP vrstvy, UDP broadcast, poskytuje lepší záznam testovaných dat (konzistentní, důkladný, jasný), export výsledku testu do CSV, došlo k rozšíření detekce poruch a obsahuje méně chyb. [38]

Vzhledem k tomu, že se boofuzz dá nainstalovat jako Python knihovna, a to pomocí příkazu pip install boofuzz, je instalace rychlá a velmi jednoduchá. Je i možnost instalace přímo ze zdroje, volně dostupného na gitHubu. [40]

Níže na obrázku 10 se nachází ukázkový skript, kterým dochází k testování zranitelnosti v TFTP serveru a následně na obrázku 11 vidíme výpis z konzole po spuštění tohoto scriptu.

```
#!/usr/bin/env python

from boofuzz import *

def main():
    session = Session(
        target=Target(
            connection=SocketConnection("192.168.98.147", 69,
proto='udp')),sleep_time = 3)

    s_initialize("Request")
    s_binary("0002")
    s_string("filename.txt", fuzzable=True)
    s_binary("00")
    s_string("netascii", fuzzable=True)
    s_binary("00")

    session.connect(s_get("Request"))
    session.fuzz()

if __name__ == "__main__":
    main()
```

Obrázek 10. Ukázkový script pro testování zranitelnosti v TFTP serveru ze zdroje 47[47]

```
python tftp_boofuzz.py
/home/daky/.local/lib/python3.9/site-packages/boofuzz/connections
/socket_connection.py:81: FutureWarning: SocketConnection is deprecated and
will be removed in a future version of Boofuzz. Use the classes derived from
BaseSocketConnection instead.
  warnings.warn(
[2022-05-01 14:15:33,126]     Info: Web interface can be found at
http://localhost:26000
[2022-05-01 14:15:33,131] Test Case: 1: Request:[Request.String2:0]
[2022-05-01 14:15:33,131]     Info: Type: String
[2022-05-01 14:15:33,131]     Info: Opening target connection
(192.168.98.147:69)...
[2022-05-01 14:15:33,131]     Info: Connection opened.
[2022-05-01 14:15:33,131]   Test Step: Monitor
CallbackMonitor#140737304432160[pre=[],post=[],restart=[],post_start_target=
[]].pre_send()
[2022-05-01 14:15:33,131]   Test Step: Fuzzing Node 'Request'
[2022-05-01 14:15:33,131]     Info: Sending 37 bytes...
[2022-05-01 14:15:33,131]     Transmitted 37 bytes: 21 40 23 24 25 25 5e 23
24 25 23 24 40 23 24 25 24 24 40 23 24 25 5e 5e 2a 2a 28 28 29 6e 65 74 61
73 63 69 69 b'!@#$%%^#$%#$@#$%$$@#$%^^**(()netascii'

[2022-05-01 14:15:33,131]   Test Step: Contact target monitors
[2022-05-01 14:15:33,131]   Test Step: Cleaning up connections from
callbacks
[2022-05-01 14:15:33,131]       Check OK: No crash detected.
[2022-05-01 14:15:33,131]     Info: Closing target connection...
[2022-05-01 14:15:33,131]     Info: Connection closed.
[2022-05-01 14:15:33,131]   Test Step: Sleep between tests.
[2022-05-01 14:15:33,131]     Info: sleeping for 3.000000 seconds
```

Obrázek 11. Ukázkový výpis při spuštění boofuzzu

3.4.1 Výhody a nevýhody

Boofuzz podporuje plně Linux, sice je možnost nainstalovat ho na Windows, ale nikde není dohledatelné, zda je plně podporován.

Jeho hlavní využití spočívá při testování internetových protokolů. Při testování aplikace nepracující na sítí, nebo neběžící na ní je prakticky nepoužitelný.

Má jednoduchou a rychlou instalaci. K správné instalaci je doporučená verze Pythonu větší, nebo rovna, verzi 3.5. Doporučená instalace také potřebuje pip. Možnost instalace i ze zdroje. [40]

Má obsáhlou dokumentaci všech funkcí, které nabízí, s možností zobrazení zdrojového kódu pro každou funkci.

Umí detekovat jaká unikátní část testu zapříčinila selhání cíle, umí tuto skutečnost zaznamenat a taktéž zvládá kategorizovat chybové stavy. [39]

Tabulka 5. Výhody a nevýhody Boofuzz

Výhody	Nevýhody
Rychlá a jednoduchá instalace.	Čisté využít jako fuzzer pro internetové protokoly (dá se polemizovat, jestli je to jeho nevýhoda).
Obsáhlá dokumentace.	Známá je pouze podpora Linuxu.
Umí detekovat a zaznamenávat poruchy a testovaná data.	
Podporuje sériový fuzzing.	
Zvládá libovolná komunikační média.	
Podporuje ethernetové a IP vrstvy a UPD broadcast.	
Výsledky testů se dají exportovat ve formátu CSV.	

3.5 Code Intelligence Fuzz

Code Intelligence Fuzz je zdarma dostupný fuzzer, který vznikl v Německu. Využívají ho velké společnosti jako například Bosch nebo Deutsche Telekom. [41]

CI Fuzz nabízí chytrou detekci chyb, pokročilý debugging, detailní hlášení chyb a zranitelností, jednoduché nastavení fuzzingu, integraci CI fuzz do vývojového prostředí (IDE, CI/CD). [42]

Plně podporuje Linuxová zařízení. Podpora pro Windows je zatím pouze experimentální.

CI Fuzz obsahuje fuzzing backend, ale také i uživatelské rozhraní. Uživatel může interagovat se systémem skrz příkazový řádek konzole anebo prostřednictvím rozšíření pro Visual Studio Code.

Backend pak spoléhá na docker, který zapouzdřuje různé úkoly například provoz a monitorování komponentů.

CI Fuzz se skládá ze tří hlavních komponentů. Prvním z nich je CI-Daemon, jedná se o Server, který je zodpovědný za úkoly backendu, jako je například kompilace projektu, správa fuzz-targetů a operace týkající se docker infrastruktury a jeho interakce s CI-Client a uživatelským rozhraním. Druhé je samostatné uživatelské rozhraní pro Visual Studio Code, které pomáhá uživateli vytvořit a spravovat fuzz-targety a reprodukovat pády. Poslední komponenta je CI-Client to je rozhraní příkazového řádku pro inicializaci sestavování a spouštění fuzzerů.

Lehce složitější instalace¸ která spočívá v nainstalování dockeru, Visual Studio Code a pro testování pomocí Java API fuzzing je potřeba Java Runtime Enviroment. Na druhou stranu lze stáhnout i obraz pro virtuální počítač pro software VirtualBox, kde je vše pro fuzz testování přednastavené a předinstalované, včetně Visual Studio Codu. [44]

3.5.1 Výhody a nevýhody

Primárně se soustředí na fuzz testování C/C++ nebo Java aplikací.

V dokumentaci je, kromě vysvětlení instalace na různá zařízení, dostupný i ukázkový projekt, s vysvětleným postupem, jak na něm použít fuzzer. Ukázkové projekty jsou jak pro fuzzing C nebo C++ aplikací, tak i pro fuzzing v Java Virutal Machine.

Uživatelské rozhraní, které se jako rozšíření nainstaluje do Visual Studio Code, je přívětivé a umožní i nezkušeným uživatelům jednoduše fuzz testovat svojí aplikaci. Navíc obsahuje

řídící panel, který po zahájení fuzzingu, zobrazí informace v reálném čase o aktuálním probíhaném fuzz testování. Například kolik bylo nalezeno unikátních vstupů nebo jaký je momentální výkon. [45]

Jakmile najde chybu poskytne uživateli i zpětnou vazbu s více informacemi o ní, například se zde nachází, jestli je to zranitelnost nebo bug, kdy byl nalezen, při jakém testu a v jakém bloku kódu došlo k chybě. [43]

Umožňuje kontinuální fuzzing v CI/CD. [46]

Přestože se na oficiálních stránkách nachází návod, jak používat fuzzer ve formě rozšíření pro Visual Studio Code, není toto rozšíření volně dostupné.

Tabulka 6. Výhody a nevýhody Code Intelligence Fuzz

Výhody	Nevýhody
Uživatelské rozhraní jako rozšíření pro Visual Studio Code.	Složitější a delší instalace.
Detailní informace o nalezených chybách.	Podpora Windows pouze experimentální.
Možnost automatického vygenerování funkce fuzz-target.	I když se na stránkách nachází návod, jak fuzz testovat, s pomocí rozšíření pro Visual Studio Code, není toto rozšíření volně dostupné.
Informace o probíhaném fuzzingu v reálném čase.	
V dokumentaci se nachází podrobné vysvětlení, jak s fuzzerem pracovat.	
Umožňuje kontinuální fuzzing.	

II. PRAKTICKÁ ČÁST

4 STRATEGIE FUZZINGU PRO PROJEKT MAESTRO PLAYBACK Z PROSTŘEDÍ MCUXPRESSO SDK

Cílem práce je zjistit, zda je možné fuzz testovat embedded software. Z toho důvodu byl vybrán projekt z prostředí MCUXpresso SDK, na který dojde k implementaci navržených strategiích pro fuzz testování embedded softwaru.

4.1 Maestro Playback

Vybraný projekt Maestro Playback je volně dostupný ukázkový projekt od společnosti NXP. Tento projekt je schopný přehrávat zvukové soubory, které se nachází na vložené SD kartě. Otestovat je potřeba, zda projekt zvládne přijímat vstupní soubory a nedojde k neočekávanému běhu aplikace. Projekt navíc při čtení a ukládání dat pracuje s cyklickou frontou, zde je namístě otestovat, jestli je zásobník dobře ošetřen a nemůže dojít ke ztrátě dat.

4.2 Volba fuzz frameworku

Pro fuzz testování projektu byla zvolena kombinace fuzzerů AFL++ a libfuzzeru.

AFL++ byl zvolen, aby otestovat hlavní funkci projektu, a to práci se soubory.

Libfuzzer byl zvolen, protože je schopný nalézt přetečení zásobníku do několika sekund.

Vzhledem k tomu, že aplikace nepracuje přes síť, není vhodné použít fuzz framework s primárním zaměřením na testování síťových aplikacích a protokolů. Toto by bylo neefektivní a nemuselo by dojít k pokrytí všech možných chyb, které by bylo možné najít s použitím více adekvátního fuzz frameworku.

4.3 Definice strategie pro testování

4.3.1 Strategie č.1: Testování celého binárního projektu

První strategie zvolená pro fuzz testování spočívá v zaměření se na projekt jako celek. Sestavit celý projekt a otestovat ho s pomocí AFL++ a jeho rozšířením QEMU, které právě umožňuje takové projekty testovat.

Pomocí build_all.sh souboru, který je již předpřipravený ve vzorovém projektu již při jeho stažení, dojde k sestavení projektu. Vznikne tak soubor s koncovkou elf, který bude následně fuzz testován.

Jako vstupní soubory se použijí nepoškozené, krátké, komprimované a nekomprimované zvukové soubory. Jedná se o 3 soubory, každý s jiným formátem. Jeden ve formátu mp3, druhý ve formátu aac a poslední ve formátu wav. Tyto tři vstupní soubory poslouží jako odrazový můstek pro další jejich mutace.

V této strategii dojde k použití pouze AFL++ fuzz frameworku. Vzhledem k tomu, že se k projektu přistupuje jako k celku a nejsme schopni ho nějakým způsobem upravit a předpokládáme neznalost kódu programu a jeho rozložení. Z těchto důvodu by použití libFuzzeru nebylo adekvátní.

Cílem této strategie je dostat rozsáhlou sadu vstupů, které by mohly být použity dále při aplikaci například dalšího fuzz frameworku, který podporuje file format fuzzing. Nebo také při otestování jiného projektu, který pracuje se stejnými formáty dat či jenom otestování, zda jsou dobře ošetřeny vstupy, aby formáty dat, které nepodporuje, nemohly nějakým způsobem ovlivnit běh programu.

Taktéž pokud při tomto stylu fuzz testování dojde k nalezení chyby, můžeme otestovat ošetření této chyby pouze pomocí souboru, který chybu vyvolal. Není tedy potřeba testovat celý program znova.

4.3.2 Strategie č.2: Vyseparování funkce z cílového programu

U druhé strategie je hlavní myšlenka otestovat zvlášť funkci bez toho, aby se testovala celková funkčnost programu. Dojde tedy k separaci jedné funkce, která nějakým způsobem buď pracuje s daty od uživatele nebo od něj přímo přijímá vstup.

Dojde k upravení a vyseparování funkce tak aby výsledná oddělená samostatná funkce byla nadále sestavitelná, spustitelná a aby hlavně nedošlo k změně funkcionality.

Prvotní testování se provede pomocí AFL++. Během této fáze dojde k otestování vyseparované funkce pomocí file format fuzzingu. Cílem této fáze je otestovat, zda jakýkoliv poškozený soubor nezapříčiní neočekávané a nesprávné chování aplikace.

Druhá fáze testování se provede s pomocí libFuzzeru. U tohoto testování se nepoužije korpus, ale postačí se se základními vygenerovanými daty.

U testování s pomocí fuzz frameworku libFuzzer dojde k napsání testovací funkce Fuzz Traget, která bude pracovat se vstupními daty. Následně dojde ke spuštění pomocí libFuzzeru, pro základní otestování postačí testování bez korpusu. Tyto vstupní data bude posílat do funkce fuzz target a ta s nimi bude dále pracovat.

Cílem této strategie je získat sadu vstupů, s kterými by se dalo dále pracovat, ale taktéž otestovat správnou funkcionalitu jedné funkce bez nutnosti testovat správnost celého programu.

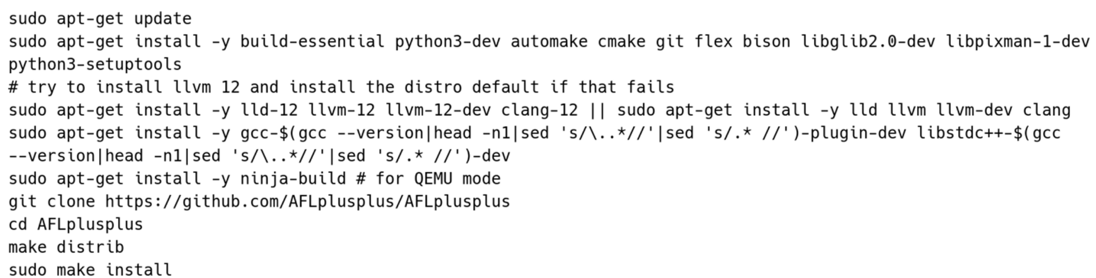
5 IMPLEMENTACE STRATEGIÍ V REÁLNÉM PROSTŘEDÍ

5.1 Příprava k implementaci strategií

Vzhledem k faktu, že implementace všech strategií probíhá na operačním systému Linux Kali nebylo potřeba nainstalovat libFuzzer. LibFuzzer je již součástí základní instalace operačního systému. Bylo tedy potřeba doinstalovat pouze AFL++, Visual Studio Code pro linuxová zařízení s podporou jazyka C++, GCC ARM Embedded toolchain a jako poslední stáhnout, z oficiálních stránek MCUXpresso SDK, testovaný projekt.

5.1.1 Instalace AFL++

Instalace AFL++ proběhla bez problému. Došlo k naklonování frameworku z jeho oficiálních github stránek. Instalaci potřebných programů a knihoven pro správnou funkci například python 3 nebo ninja-build (více viz obrázek č.13). S pomocí příkazu make distrib a poté sudo make install došlo k sestavení a nainstalování kompletního fuzz frameworku se všemi podporovanými rozšířeními.



Obrázek 12. Instalace AFL++

5.1.2 Instalace Visual Studio Code

Další krok bylo nainstalovat Visual Studio Code. Na oficiálních stránkách Microsoftu pro Visual studio Code došlo k vybrání operačního systému Linux a programovacího jazyku C++. Následně byla stažena verze Visual Studio Code pro Linux, soubor Linux x64 .deb. Oficiální celkový název staženého souboru byl: code_1.66.2-1649664567_amd64.deb. Tomto souboru bylo v příkazovém řádku pomocí příkazu sudo chmod +x code_1.66.2-1649664567_amd64.deb, přidáno oprávnění k spuštění. Samotná instalace proběhla pomocí dpkg neboli debian package manageru. Příkaz pro instalaci byl: sudo dpkg -i code_1.66.2-

1649664567_amd64.deb, kde možnost -i znamená instalační příkaz. Všechny příkazy použity pro instalace jsou zobrazeny na obrázku č.13.

```
sudo chmod +x code_1.66.2-1649664567_amd64.deb
sudo dpkg -i code_1.66.2-1649664567_amd64.deb
```

Obrázek 13. Instalace Visual Studio Code

5.1.3 Instalace GCC ARM Embedded toolchain

Jako poslední bylo potřeba stáhnout a sprovoznit GCC ARM Embedded toolchain. Z oficiálních stránek ARM Developer byl stáhnut gcc arm verze 11.2 pro Aarch32 bare-metal target (arm-none-eabi).

V domovském adresáři byla vytvořena složka opt do které byl přesunut stažený toolchain. Následně byl pomocí příkazu tar s parametry xjf, soubor rozbalen a byly mu změněny oprávnění na pouze pro čtení.

Ke kontrole funkčnosti kompilátoru byl použit příkaz: ~/opt/gcc-arm-none-eabi-8-2018-q4-major/bin/arm-none-eabi-gcc –version. Ukázkový výpis z konzole zobrazen na obrázku č.14.

```
$ ~/opt/gcc-arm-11.2-2022.02-x86_64-arm-none-eabi/bin/arm-none-eabi-gcc --version
arm-none-eabi-gcc (GNU Toolchain for the Arm Architecture 11.2-2022.02 (arm-11.14))
11.2.1 20220111
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Obrázek 14. Kontrola funkčnosti GCC ARM kompilátoru

5.1.4 Stažení projektu ze stránek MCUXpresso SDK

Z oficiálních stránek MCUXpresso SDK bylo sestaveno SDK pro desku MIMXRT1170-EVK. Toto SDK bylo sestaveno pro hostovaný operační systém Linux se zvoleným toolchainem GCC ARM Embedded.

Po sestavení balíčku byl balíček stáhnut a rozbalen do předem vytvoření složky s názvem MCUXpresso.

Cesta k ukázkovému projektu, na kterém bude probíhat fuzz testování je: ~/MCUXpresso/Board/MIMXRT1170/Audio_Example/Maestro_playback.

5.2 Implementace Strategie č. 1

Cílem této implementace bylo zjistit, zda se dá fuzz testovat embedded software, od kterého máme dostupný pouze sestavený binární soubor. AFL++ v tomto testování na vstup stdin posílá zvukové soubory. Níže je popsán celý postup testování i s jeho výsledky.

Prvním krokem u implementace strategie číslo 1, bylo sestavení projektu pomocí přiložených build souborů. Vzhledem k tomu, že CMake potřebuje definovat proměnou k nalezení kompilátoru byl prvotní krok ve složce, kde se nacházeli build soubory, toto definovat. Pomocí příkazu: export ARMGCC_DIR=~/opt/gcc-arm-11.2-2022.02-x86_64-arm-none-eabi, bylo pro CMake definováno umístění ARM GCC kompilátoru.

Před sestavením projektu je potřeba nastavit oprávnění pro spuštění a následně je již možné sestavení programu s příkazem ./build_all.sh.

```
$ export ARMGCC_DIR=~/opt/gcc-arm-11.2-2022.02-x86_64-arm-none-eabi
$ chmod +r build_all.sh
$ ./build_all.sh
```

Obrázek 15. Sestavení projektu maestro_playback

Došlo k automatickému vytvoření dvou složek: flexspi_nor_debug a flexspi_nor_realese. Obě složky obsahují soubory s koncovkou .elf, ale u souboru nacházejícího se ve složce flexspi_nor_debug je navíc povoleno zobrazení informacích o ladění.

Vzhledem k faktu, že testovaný soubor je sestavený pro procesor ARM, který nelze spustit na počítači s procesorem Intel. Proto jsem využila možnost AFL++ frameworku, a to testovat s pomocí QEMU, které je schopno emulovat ARM procesor.

Aby došlo k oddělení fuzz testování od samotného projektu vytvořila jsem novou složku s názvem fuzz. Do této složky jsem vytvořila 2 podsložky. Jednu s názvem in, do které se vloží základní vstupní data a druhou s názvem out do které se budou ukládat výstupy z testování.

Do složky in jsem, ze zdroje č.48, stáhla tři krátké, šestnácti sekundové, jedno kanálové, soubory. Jednalo se o soubory: gs-16b-1c-44100hz.mp3, gs-16b-1c-44100hz.aac, gs-16b-1c-44100hz.wav. Tyto soubory jsem následně pro zjednodušení a přehlednost přejmenovala na: SAMPLE1.mp3, SAMPLE2.wav a SAMPLE3.aac tak aby byla koncovka zachována a nedošlo k jejímu přepsání a třeba i poškození souboru.

Do složky fuzz jsem přesunula soubor maestro_playback.elf ze složky flexspi_nor_debug- pomocí příkazu cp flexspi_nor_debug/maestro_playback.elf fuzz.

Po přesunu binárního souboru na testování do složky fuzz jsem se v příkazovém řádku pomocí cd fuzz přesunula do téže složky, do které byl soubor přemístěn. V této složce jsem nastavila a spustila všechny důležité scripty a konfigurace potřebné před zahájením testování. Povolila jsem ASANu, který detekuje chyby v paměti C a C++ aplikace, a UBSANu, který na rozdíl do ASAN detekuje nedefinované chování programu. Obě možnosti jsem povolila pomocí pomocí příkazů: export AFL_USE_UBSAN=1 a export AFL_USE_ASAN=1. Pro jistotu jsem ještě spustila konfigurační soubor AFL++ k nakonfigurování systému pro testování a následně přešla k pokusu o spuštění fuzzeru na maestro_playback.efl. Všechny použité příkazy zobrazeny na obrázku č.16.

```
$ export AFL_USE_UBSAN=1
$ export AFL_USE_ASAN=1
$ sudo ~/AFLplusplus/afl-system-config
$ ~/AFLplusplus/afl-fuzz -i in/ -o out -m none -d -Q -- ./maestro_playback.elf @@
```

Obrázek 16. Konfigurace před spuštěním a samotné spuštění AFL++

Nedojde ke spuštění fuzzeru. Všechny ukázkové vstupní soubory zapříčiní okamžitý pád programu. K tomu, aby fuzzer mohl být spuštěn, je totiž potřeba minimálně jeden soubor, který neshodí program. Ale vzhledem k tomu, že ve složce in jsou 3 různé zvukové soubory, které jsou přehratelné a nějak poškozené, problém s největší pravděpodobností nebude v nich. Přesto jsem stáhla ze zdroje č.48 další vstupní souboru a vyzkoušela běh fuzzeru znova, avšak se stejným výsledkem. Došla jsem k závěru, že program nejspíš není schopný přijmout vstupní data.

Vzhledem k tomu, že chyba přetrvává i při změně vstupních dat nejspíše se nemůže fuzzer s programem domluvit na způsobu práce s daty. I když AFL++ komunikuje s programem přes zacyklit AFL++ pouze na jedné funkci. K tomu je možné využít persistentní mód, který QEMU umožňuje.

QEMU persistentní mód umožňuje spustit Fuzzer na určité funkci v binárním souboru, bez nutnosti tuto funkci vykopírovat do vlastního souboru. I kdyby na začátku běhu programu docházelo k inicializaci periférií, možné spuštění AFL++ pouze na jedné funkci, která s těmito periferiemi nepracuje, by mohlo být úspěšné.

Ve složce fuzz jsem spustila příkaz nm maestro_playback.elf, který vypíše všechny možné informace o binárním souboru, přesněji názvy všech funkcí, které binární soubor obsahuje, avšak nevypíše, s jakými daty funkce pracují. Pro rychlejší hledání by dopomohla znalost kódu, ale první strategie předpokládá onu neznalost. Vybrala jsem nakonec funkci STREAMER_file_Create, která soudě podle jejího pojmenování, by mohla pracovat s daty (viz obrázek č.17). Jeden z problému je ten že nm vypíše binární adresu, která je pozičně nezávislá, ale QEMU persistent potřebuje skutečnou adresu, ne offset. Naštěstí QEMU načítá spustitelné pozičně nezávislé soubory na fixní adrese 0x4000000000 pro x86_64, můžeme tedy říct, že adresa funkce, pokud je projekt sestaven jako spustitelně pozičně nezávislý, je 0x40300065b1.

```
$ nm maestro_playback.elf | grep STREAMER
300065b1 T STREAMER_file_Create
300069a9 T STREAMER_Init
3000639d t STREAMER_MessageTask
```

Obrázek 17. Nalezení vhodné funkce pomocí kombinace příkazu nm a grep

Hlavní problém nastává v tom, že projekt maestro_playback nebyl sestaven jako spustitelně pozičně nezávislý. To se dá zjistit pomocí příkazu file maestro_playback.elf. Na výpisu zobrazeném na obrázku 18 vidíme, kromě toho, že se jedná o 32 bitový soubor i to že je LSB executable. Pokud by byl pozičně spustitelně nezávislý bylo by zde místo toho napásáno LSB pie executable

```
$ file maestro_playback.elf
maestro_playback.elf: ELF 32-bit LSB executable, ARM, EABI5 version 1
(SYSV), statically linked, with debug_info, not stripped
```

Obrázek 18. Výpis informací o souboru maestro_playback.elf

Přesto jsem se rozhodla zjistit, jestli se na dané adrese funkce nenachází. Nastavila jsem QEMU persistent adresu příkazem AFL_QEMU_PERSISTENT_ADDR=0x40300065b1. Vzhledem k tomu, že se parametry předávají do registrů, jakmile dojde na konci funkce k vrácení se na počáteční adresu, jsou registry zablokované. To znamená že nadále nemáme ukazatel na název souboru. Abychom tomu zabránili uložíme a obnovíme stav obecných registrů při každém nastavení iterace pomocí AFL_QEMU_PERSISTENT_GPR=1.

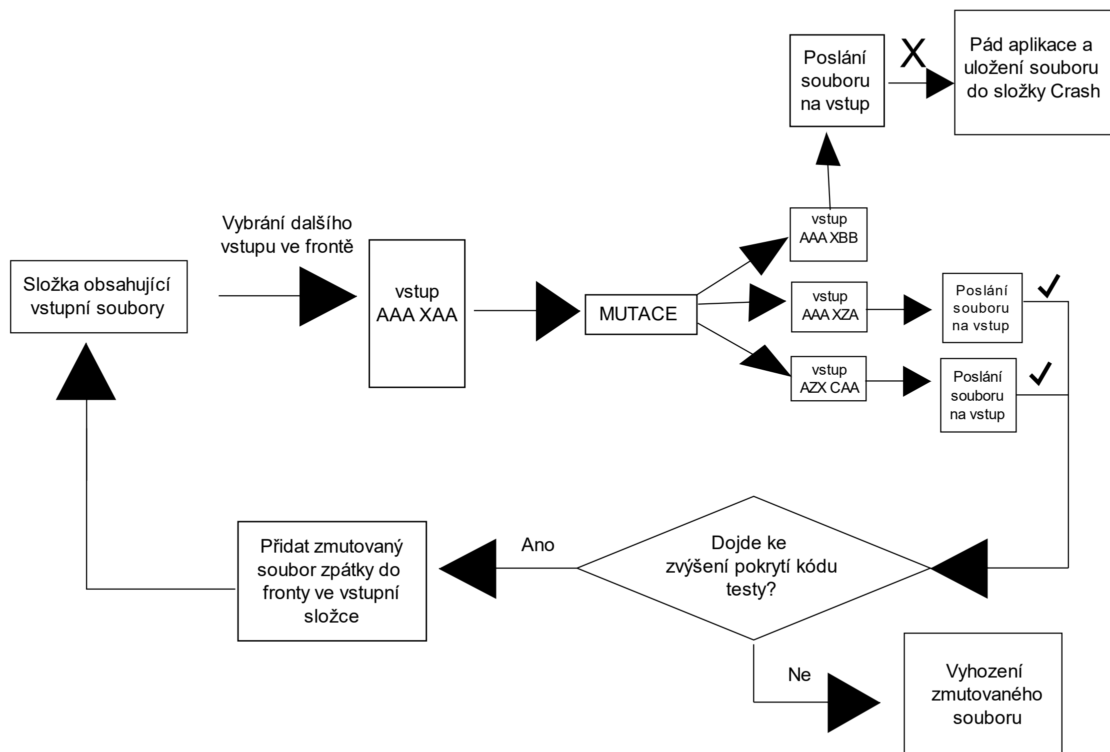
Stejným příkazem jako při prvním pokusu jsem spustila AFL++ fuzzeru, avšak opět s negativním výsledkem. Tentokrát nedošlo ani k navázání spojení s fork serverem. S největší pravděpodobností nesprávně zvolená adresa při použití persistentního QEMU módu.

Vzhledem k chybě s adresu jsem se rozhodla k pokusu zkusit změnit adresu čistě na offset. Zde, ale opět bezúspěšně a se stejnou chybou. Ani poslední možnost, nabízená v navrhovaných řešeních, a to zakázat fork server nebyla úspěšná, nedošlo ani ke změně vyvolané chyby.

Jak bylo později zjištěno AFL++ komunikuje s programem přes stdin vstup kam posílá vstupní soubory. Problém, ale je, že volně dostupný projekt Maestro_Playback nepřijímá jako vstup přímo zvukový soubor, ale název souboru, který následně vyhledává v souborovém systému.

AFL++ vybere další vstupní soubor nacházející se ve frontě ve vstupní složce, která je definovaná při spuštění AFL++. Tento soubor následně několikrát zmutuje a vzniklé mutace pošle přes stdin do programu. Pokud určitá mutace zapříčiní neočekávaný běh programu, uloží jí do podsložky s názvem Crash a na statusové obrazovce informuje o nalezení chyby. Jestliže ale mutace nezapříčiní neočekávaný běh programu a zvýší hodnotu pokrytí kódu testy, je uložena do zpátky do vstupního korpusu a zařazena zpět do fronty. Pokud avšak mutace nezapříčiní neočekávaný běh programu, ale nedojde k zvýšení hodnoty pokrytí kódu testy je taková mutace zahozena. Celý postup běhu AFL++ je zobrazen na obrázku č.19.

Hlavní problém u implementace této strategie spočívá v nedostupnosti vývojové desky. Existuje totiž možnost spustit AFL++ přímo na vývojové desce. Vyhnuli bychom se tak potřebě řešit například emulaci souborového systému SD karty, nebo dalších periferiích, které by program na začátku inicializoval a které by byly dostupné na vývojové desce. Tento problém by šel obejít, pokud by se povedlo spustit AFL++ pouze na určité funkci, které s periferiemi nepracuje. Tehdy by bylo možné fuzz testovat desku, bez nutnosti ji mít k dispozici.



Obrázek 19. Grafické znázornění funkce AFL++

5.2.1 Zhodnocení výsledků implementace strategie č.1

Vzhledem k tomu, že selhaly všechnu známé možnosti, jak otestovat binární kód pomocí AFL++ je implementace první strategie neúspěšná. AFL++, ani zatím, žádný jiný volně dostupný fuzzer, není uzpůsoben na fuzz testování embedded softwaru byla možnost neúspěchu, při testování programu jako celku, pravděpodobná.

QEMU umožňuje spuštění AFL++ na adrese funkce, avšak při projektu, který nebyl sestaven spustitelně pozičně nezávislí bychom mohli přesnou adresu funkce zjistit jedině krokováním programu. Toho ale vzhledem k nedostupnosti vývojového hardwarového kitu této desky nejsem schopná dosáhnout.

Ale pořád se jedná jenom o spekulaci, jestli se pak povede rozběhnout testovaný program společně s fuzzerem, aby došlo k zdárnému výsledku.

Zvolená strategie byla neúspěšná, protože testovaný program svou povahou a komplexností není vhodný pro black box testování. Je u něj nutné testovat jednolitvé funkce zvlášť.

```
[-] Unable to communicate with fork server. Some possible reasons:

    - You've run out of memory. Use -m to increase the the memory limit
      to something higher than 0.
    - The binary or one of the libraries it uses manages to create
      threads before the forkserver initializes.
    - The binary, at least in some circumstances, exits in a way that
      also kills the parent process - raise() could be the culprit.
    - If using persistent mode with QEMU, AFL_QEMU_PERSISTENT_ADDR is
      probably not valid (hint: add the base address in case of PIE)

If all else fails you can disable the fork server via AFL_NO_FORKSRV=1.

[-] PROGRAM ABORT : Unable to communicate with fork server

         Location : afl_fsrv_run_target(), src/afl-forkserver.c:1573
```

Obrázek 20. Chybová hláška při běhu AFL++ na QEMU persistentní adrese 0x40300065b1

5.3 Implementace strategie č.2

Cílem této strategie je otestovat, jestli je možné fuzz testovat část zdrojového kódu embedded softwaru, bez nutnosti sestavit celý projekt. Pokud by tímto způsobem měla být otestovaná funkce, které pracuje s některou z periferií a pokud lze nějakým způsobem ji emulovat, je vhodné zkusit hardware emulovat. Jestliže by, ale nemohlo dojít k vytvoření emulace, AFL++ lze spustit na vývojové desce, tudíž by bylo možné vyseparovat funkci pracující s hardware a s pomocí AFL++ ji otestovat přímo na vývojové desce. LibFuzzer sice umí emulovat ARM procesor, ale otázka je, jestli je schopný běžet na vývojové desce. Jediná možnost u něj by byl běh na počítači, kde by mohl pomocí agenta na desce, komunikovat s běžícím programem.

Pro tento účel, ale jsem ale vybrala dvě funkce, které s hardwarem nekomunikují a je tedy možnost je jednoduše otestovat. Celkový podrobnější postup testování, výsledky testování a popis vyseparovaných funkcích je popsán níže v této kapitole.

Vyseparovala jsem dvě funkce nacházející se v souboru app_streamer.c. Jednalo se o funkce s názvem STREAMER_Read a STREAMER_Write. Separace dvou funkcí se odklání od prvotního záměru vykopírovat ze souboru pouze jednu funkci, ale vzhledem k tomu, že funkce spolu souvisí, rozhodla jsem se je otestovat spolu.

První vstup funkce STREAMER_Write jsou data, která se zapíší do cyklické fronty a druhý vstup velikost vstupních dat. Naopak funkce STREAMER_Read čte a ukládá data z cyklické

fronty do prvního ze vstupů, druhý vstup je požadovaná velikost přečtených dat. Obě funkce vrací počet přečtených nebo zapsaných dat.

Vytvořila jsem soubor s názvem MaestroTest.c, který jsem uložila do složky testFile. Do tohoto souboru jsem překopírovala ze souboru app_streamer.c dvě, již výše zmíněné funkce: STREAMER_Read a STREAMER_Write. Překopírovala jsem i potřebné knihovny, hlavně pak hlavičkový soubor osa_common.h. Tento hlavičkový soubor k tomu, aby byl přeložitelný, potřeboval několik dalších knihoven, které se nacházeli v různých složkách v originálním projektu. Tyto soubory jsem vyhledala a importovala do projektu. Pro tento účel jsem vytvořila ve složce testFile podsložku s názvem OsaInclude. Do této složky jsem vložila všechny potřebné hlavičkové soubory, které se nacházely v originálním projektu, jenž byly potřeba pro správnou funkci aplikace.

Hlavní problém při vykopírování funkcí byl ten, že obě funkce pracují s dalšími funkcemi, které jsou sice deklarované v hlavičkovém souboru, ale jejich definice se nachází v objektových souborech uložených v archivních knihovnách. Z toho vyplývá že uživatel nemá k definicím přístup a nemůže si je z objektových souborů vykopírovat.

Jedna z funkcích, ke které nemá běžný uživatel přístup, je funkce pracující s mutexem. Tyto funkce, ovládající mutex, jsem z kódu nakonec odstranila, vzhledem k tomu, že v programu se nenachází žádné další funkce, které by mohli nějakým způsobem ovlivnit zápis nebo výpis do cyklické fronty. Dále se jednalo o funkci, která zapisuje do cyklické fronty a o funkci, které z cyklické fronty data čte.

Po konzultaci problému s odborníkem z firmy NXP došlo k navrhnutí třech možných řešení. Použít definici funkci přímo od NXP, ale nikde jí nepublikovat, vytvořit mock, nebo použít volně dostupnou definici. Rozhodla jsem se k výběru třetí možnosti. I když by použití definice funkce přímo od firmy NXP bylo rozumnější, nemožnost publikovat ji by mohla znehodnotit celou práci.

Nakonec jsem použila volně dostupnou základní implementaci cyklické fronty ze zdroje 49. Tuto implementace jsem poupravila tak aby splňovala deklaraci funkce v hlavičkovém souboru. K definicím funkcí jsem přidala proměnnou s názvem Size, která značí velikost zapsaných nebo čtených dat. Ta sice není nikde ve volně dostupné implementaci funkce používaná, ale vzhledem k tomu, že funkce deklarovaná v hlavičkovém souboru tuto proměnou požaduje a také pro potřeby například budoucího testování, kde by se mohla místo volně dostupné implementace funkce použít oficiální definice funkce, bude lepší vyhýbat se

jakýmkoliv úpravám v deklaracích funkcích a místo toho se jim uzpůsobit. Také došlo ke změně vstupních dat z uint8_t data na ukazatel uint8_t *data.

Finální úpravy obou funkcí, STREAMER_Read a STREAMER_Write, spočívaly v odstranění Mutex funkcí. A vzhledem k tomu, že upravené funkce, které nahradily oficiální definici funkcí, pracující se čtením a zápisem do cyklické fronty, vrací při podařeném provedení nulu, při jakékoliv chybě mínus jedničku, upravila jsem tyto funkce tak, aby hlídaly, zda je zápis nebo čtení provedené v pořádku. Změnila jsem podmínku, která prvně hlídla, zda se vrácená hodnota rovná velikosti dat. To vzhledem k tomu, že originální implementace podle všeho vracela velikost zapsaných nebo přečtených dat. Podmínku jsem upravila pro obě funkce jak pro STREAMER_Read tak i STREAMER_Write.

Aby bylo ale možné otestovat pomocí AFL++, zda funkce nemá problém se soubory upravila jsem hlavní funkci main tak aby s těmi soubory aplikace pracovala. Pro tento účel jsem vzala ze zdroje 49 Makro cyklické fronty s pevnou velikostí 32 (viz obrázek 21.).

```
#define CIRC_BBUF_DEF(x,y)
 \  uint8_t x##_data_space[y];
 \  circ_bbuf_t x = {
\       .buffer = x##_data_space
\       .head = 0,
 \      .tail = 0,
 \      .maxlen = y
\   }
CIRC_BBUF_DEF(my_circ_buf, 32);
```

```
#define CIRC_BBUF_DEF(x,y)
 \  uint8_t x##_data_space[y];
 \  ringbuf_t x = {                    \
        .buf = x##_data_space,        \
        .head = 0,
 \      .tail = 0,
 \      .size = y                      \
    }

CIRC_BBUF_DEF(my_circ_buf, 32);
```

Obrázek 21. Vlevo originální Makro, vpravo upravené Makro

V hlavní funkci main jsem následně přidala proměnné typu FILE, uint8_t a char. Proměnná typu FILE pracovala se vstupním parametrem při spuštění programu. Soubor, v zadaném umístění, se otevírá pomocí funkce fopen, která je součástí knihovny stdio.h. Počítalo se s tím, že soubor bude vždy umístěn v jedné podsložce. Došlo ke kontrole, zda ukazatel na soubor není prázdný a pomocí funkce fgets k přečtení celého souboru. Vzhledem k tomu, že se pracovalo pouze s jedním zadaným parametrem ze souboru rozdělila jsem pomocí funkce strtok získaný vstup. Separátorem se staly mezery mezi jednotlivými slovy. Data, které byla tedy zapsána se nacházela hned ze začátku souboru.

Následně došlo k zavolání funkce STREAMER_Write, kde vstupní parametry byla proměnná in_data, která obsahovala data k zápisu a hodnota 32, která značila velikost zásobníku. Zapsání dat bylo definováno podmínku, která pokud nedošlo k správnému zapsání dat

ukončila celý běh programu. Pokud ovšem došlo k bezproblémovému zapsání dat mohly se data přečíst, přečtená data se uložili do proměnné out_data. Celá hlavní funkce main je vidět na obrázku 22.

```
int main(int argc, char **argv)
{
    FILE* ptr;
    uint8_t out_data=0;
    uint8_t *in_data;
    char str[32];

    if(argc > 1){
    ptr = fopen(argv[1], "a+");
     if (NULL == ptr) {
        printf("file can't be opened \n");
    }

    while (fgets(str, 32, ptr) != NULL) {
            //printf("%s\n", str);
    }
    fclose(ptr);
    char * token = strtok(str, " ");
    in_data = str;


    if (STREAMER_Write(in_data, 32)) {
        printf("Out of space in CB\n");
        return -1;
    }

    STREAMER_Read(&out_data, 32);
    // here in_data = in_data = 0x55;

    printf("Push: 0x%x\n", in_data);
    printf("Pop:  0x%x\n", out_data);
    }
    return 0;
}
```

Obrázek 22. Kompletní funkce Main

Sestavila jsem projekt pomocí AFL verze clangu, s povoleným ASAN. Pro jistotu jsem ASAN a UBSAN povolila ještě jednou s příkazem export. Celková sada příkazu provedena před spuštěním testování je vidět níže na obrázku 23.

```
$ afl-clang-fast -fsanitize=address,undefined -ggdb -O0 MaestroTest.c -o mae_AFL
$ export AFL_USE_UBSAN=1
$ export AFL_USE_ASAN=1
$ sudo ~/AFLplusplus/afl-system-config
```

Obrázek 23. Konfigurace systému a sestavení projektu

Vytvořila jsem podsložku s názvem in, do které jsem přidala 4 vstupní soubory. Jednalo se o soubory ve formátu: dva .txt, .odt a .doc. Tyto vstupní soubory jsem naplnila náhodným textem, aby nebyly prázdné. Spustila jsem AFL++ pomocí příkazu: ~/AFLplusplus/afl-fuzz -i in/ -o out -m none -d -- ./mae_AFL @@.

Nedošlo k nalezení žádné chyby, celkový běh fuzzeru byl okolo 40 minut viz obrázek 24. AFL++ na stdin vstup posílá vstupní soubory, to samé prováděl i u první strategie. Žádný ze vstupních souborů ani jejich mutací nezapříčinil neočekávaný běh programu.

```
american fuzzy lop ++4.01a {default} (./mae_AFL) [fast]
process timing                                overall results
       run time : 0 days, 0 hrs, 59 min, 32 sec    cycles done : 424
  last new find : 0 days, 0 hrs, 58 min, 42 sec   corpus count : 9
last saved crash : none seen yet                saved crashes : 0
 last saved hang : none seen yet                  saved hangs : 0
cycle progress                        map coverage
 now processing : 3*334 (33.3%)          map density : 23.73% / 23.73%
 runs timed out : 0 (0.00%)           count coverage : 3.79 bits/tuple
stage progress                        findings in depth
 now trying : splice 12               favored items : 1 (11.11%)
stage execs : 0/12 (0.00%)             new edges on : 1 (11.11%)
total execs : 9.97M                   total crashes : 0 (0 saved)
 exec speed : 2758/sec                 total tmouts : 0 (0 saved)
fuzzing strategy yields                             item geometry
  bit flips : disabled (default, enable with -D)      levels : 2
 byte flips : disabled (default, enable with -D)     pending : 0
arithmetics : disabled (default, enable with -D)    pend fav : 0
 known ints : disabled (default, enable with -D)   own finds : 7
 dictionary : n/a                                   imported : 0
havoc/splice : 3/3.49M, 4/6.48M                    stability : 100.00%
py/custom/rq : unused, unused, unused, unused
   trim/eff : 13.87%/176, disabled                      [cpu000: 25%]
```

Obrázek 24. Běh AFL++ fuzz frameworku

Dále jsem se rozhodla upravit soubor MaestroTest.c tak aby s funkcemi mohl pracovat libFuzzer. Vytvořila jsem kopii souboru MaestroTest.c s názvem CpyForLibFuzzer.c. V této kopii jsem odstranila hlavní funkci Main, vzhledem k faktu že si libFuzzer poradí bez této funkce, a přidala funkci fuzz target, ve které jsem definovala, co se mí stát s vygenerovanými daty.

Vstupní data pro fuzz target funkci s názvem LLVMFuzzerTestOneInput byl ukazatel na vstupní data datového typu uint8_t a velikost oněch dat datového typu size_t. Ve funkci byla vytvořena proměnná outdata, která opět poslouží k uložení přečtených dat. Vstupní proměnná Size byla použita jako velikost zásobníků a vygenerovaná data jako vstup. Celková fuzz target funkce je vidět na obrázku 25. Fuzz target funkce je důležitá součást libFuzzeru, protože přes ní dochází k posílání dat do testovaných funkcí. LibFuzzer totiž funguje tak, že se spojí s testovanou knihovnou a vstupy pro její otestování posílá právě přes tuto funkci,

následně sleduje, jakých úrovní kódu bylo dosaženo a za účelem maximalizování možného pokrytí kódu testy, vstupní data zmutuje.

V těle funkce Target jsem následně přidala proměnou outdata datového typu uint8_t, pro uložení přečtených dat. Vstupní data byla použita jako vstup pro funkci STREAMER_Write společně s proměnnou Size. Vstupní proměnná Size v těle funkci přepsala aktuální velikost cyklické fronty, na velikost aktuálních dat na vstupu. Po zapsání dat do cyklické fronty dojde k jejich přečtení pomocí funkce STREAMER_Read.

```
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    uint8_t outdata = 0;

    my_circ_buf.size = Size;

  STREAMER_Write(Data, Size);
  STREAMER_Read(&outdata, Size);

  return 0;
}
```

Obrázek 25. Fuzz target funkce

Jakmile jsem definovala fuzz target, sestavila jsem projekt pomocí Clang kompilátoru. A to příkazem clang -fsanitize=address,fuzzer CpyForLibFuzzer.c. Z příkazu vyplývá že došlo k povolení použití ASAN, to definuje parametr address. Parametr fuzzer říká, že dojde k binárnímu sestavení. Tento příkaz provede nezbytnou instrumentaci a propojení s libFuzzer knihovnou.

Následně jsem spustila výstupní soubor, vzniklý po sestavení projektu, a.out. Nebyl použit žádný vzorový korpus obsahující vzorové vstupní data. Skoro okamžitě došlo k přetečení zásobníků a nalezení chyby. Celá chybová hláška se nachází na obrázku 26. Kromě chybové hlášky, ale můžeme vyčíst i to, že libFuzzer zkusil alespoň 148 vstupů, to značí ono číslo za hashtagem a dosáhl 16 bodů celkového pokrytí kódu testy. Na jednom z těchto vstupů detekoval ASAN přetečení zásobníků. Z chybové hlášky lze vyčíst, že k přetečení došlo ve funkci ringbuf_write, co za data přetekla zásobník a do jakého souboru byla tato informace uložena.

```
└─$ ./a.out
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 3239528156
INFO: Loaded 1 modules   (16 inline 8-bit counters): 16 [0x548fc0, 0x548fd0),
INFO: Loaded 1 PC tables (16 PCs): 16 [0x521e98,0x521f98),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 10 ft: 11 corp: 1/1b exec/s: 0 rss: 30Mb
#3      NEW    cov: 15 ft: 17 corp: 2/3b lim: 4 exec/s: 0 rss: 30Mb L: 2/2 MS: 1 InsertByte-
#4      NEW    cov: 16 ft: 18 corp: 3/5b lim: 4 exec/s: 0 rss: 30Mb L: 2/2 MS: 1 InsertByte-
#148    REDUCE cov: 16 ft: 18 corp: 3/4b lim: 4 exec/s: 0 rss: 30Mb L: 1/2 MS: 4 ChangeByte-ChangeByte-ShuffleBytes-
EraseBytes-
=================================================================
==3207474==ERROR: AddressSanitizer: global-buffer-overflow on address 0x000000e30f00 at pc 0x00000050a834 bp 0x7fffffffd510
sp 0x7fffffffd508
WRITE of size 1 at 0x000000e30f00 thread T0

    #0 0x50a833 in ringbuf_write (/home/daky/testFile/a.out+0x50a833)
    #1 0x50a93a in STREAMER_Write (/home/daky/testFile/a.out+0x50a93a)
    #2 0x50abb6 in LLVMFuzzerTestOneInput (/home/daky/testFile/a.out+0x50abb6)
    #3 0x440a03 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) (/home/daky/testFile/a.out+0x440a03)
    #4 0x43fe1d in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned long, bool, fuzzer::InputInfo*, bool, bool*)
(/home/daky/testFile/a.out+0x43fe1d)
    #5 0x441abb in fuzzer::Fuzzer::MutateAndTestOne() (/home/daky/testFile/a.out+0x441abb)
    #6 0x4426e5 in fuzzer::Fuzzer::Loop(std::vector<fuzzer::SizedFile, fuzzer::fuzzer_allocator<fuzzer::SizedFile> >&)
(/home/daky/testFile/a.out+0x4426e5)
    #7 0x42f97d in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long))
(/home/daky/testFile/a.out+0x42f97d)
    #8 0x45b6f2 in main (/home/daky/testFile/a.out+0x45b6f2)
    #9 0x7ffff7a3b7fc in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x277fc)
    #10 0x423929 in _start (/home/daky/testFile/a.out+0x423929)

0x000000e30f00 is located 0 bytes to the right of global variable 'my_circ_buf_data_space' defined in
'CpyForLibFuzzer.c:17:1' (0xe30ee0) of size 32
SUMMARY: AddressSanitizer: global-buffer-overflow (/home/daky/testFile/a.out+0x50a833) in ringbuf_write

Shadow bytes around the buggy address:
  0x0000801be190: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0000801be1a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0000801be1b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0000801be1c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0000801be1d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0000801be1e0:[f9]f9 f9 f9 00 00 00 00 00 00 00 00 00 00 00 00
  0x0000801be1f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0000801be200: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0000801be210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0000801be220: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0000801be230: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==3207474==ABORTING
MS: 5 InsertRepeatedBytes-CrossOver-InsertByte-CrossOver-InsertRepeatedBytes-; base unit:
adc83b19e793491b1c6ea0fd8b46cd9f32e592fc
0xa,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0x27,0xff,0xff,0xff,0xff,0xff,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0
x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x
0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0xf
f,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xa,0xff,0xff,0xa,0xff,0xff,0xff,0xff,0xff,0xff,
0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
\x0a\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff'\xff\xff\xff\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xff\xff\xff\xff\xff\xf
f\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\x0a\xff\xff\x0a\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
\xff\xff
artifact_prefix='./'; Test unit written to ./crash-44b170fb471455a0d85ab775d59417004071a1ca
Base64:
Cv//////////////J///////AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA/////////////////////wr//wr/////////////////////////
```

Obrázek 26. Chyba nalezená pomocí libFuzzeru

5.3.1 Zhodnocení výsledků implementace strategie č.2

Druhá strategie byla na implementaci o trochu pracnější než první. To hlavně proto, že kromě vykopírování funkce muselo dojít i k přepsání používaných funkcí oné funkce a importování všech potřebných knihoven a hlavičkových souborů, které byly mnohdy různě umístěné v originálním souboru.

Ačkoliv testování pomocí fuzz frameworku AFL++ nenašlo žádnou chybu a zaměřovalo se pouze na zkoušení souborů, bylo zde přesto dokázané, že lze použít této framework při testování embedded softwaru.

LibFuzzer byl úspěšnější a nalezl chybu v definici funkce ringbuf_write. Tato chyba vznikla při nahrazení oficiální definice ringbuf_write volně dostupnou implementací. Při přepisu funkce došlo ke změně vstupní proměnné z uint8_t Data na ukazatel uint8_t *Data. Primárně byla funkce určena pro zapisování jednoho bytu dat, ale při přepsání na ukazatel dochází k zapisování většího množství dat. Tuto chybu jsem zkusila vyřešit smyčkou, která by vykonala zápis po jednotlivých bytech dat, avšak libFuzzer stále detekuje přetečení ve funkci ringbuf_write. Vzhledem k tomu, že ale nebyly použity oficiální definice není zaručeno, ale ani vyvráceno, že v originálním projektu se tato chyba nenachází.

Cílem práce je pouze ověřit možnosti použití fuzz testování a toho se dá docílit i bez použití originálních funkcí, ke kterým nemá obyčejný uživatel přístup. Aby se povedlo vyextrahovat z testované aplikace všechny příslušné funkce není náplní této práce, stačilo pouze zprovoznit funkce tak aby na něj mohla být použita metoda fuzz testování.

Přesto nalezení chyby bylo v řádu sekund a došlo k ní téměř okamžitě po spuštění fuzz testování. Stejně jako u AFL++ i zde bylo dokázáno, že je tedy možné fuzz testovat tímto způsobem embedded software, a i napasování na testování pomocí libFuzzeru je o něco rychlejší a jednoduší než při použití AFL++.

5.4 Návrh dalšího postupu pro fuzz testování

První strategie měla za úkol zjistit, zda je možné otestovat embedded software jako celý složený binární program. Pro tyto účely byl vybrán AFL++, protože došlo k chybné úvaze, že program přijímá jako vstupní data soubory ve zvukovém formátu. Pravda je takové že program pracuje s názvem souboru, který vyhledává v souborovém systému. Strategii se nakonec nepovedlo úspěšně implementovat a je tedy otázka, jestli by návrh dalšího fuzz testování touto strategií na jiném embedded projektu mohl být úspěšný.

Obecně by, ale mohlo dojít k její úspěšné implementaci na projektu, který přijímá vstupní data z příkazového řádku ve formě souborů, a ne jenom jejich názvu, které dále vyhledává. Důvod vybrání AFL++ byl i taký, že s pomocí QEMU persistentního módu je možné zacyklit fuzzer pouze na jedné funkci bez nutnosti funkci separovat. Dochází tak k běhu AFL++ jenom na jedné funkci programu, ale bez nutnosti mít přístup ke zdrojovým souborům a funkci vyseparovat. Zde je ovšem potřeba znát přesnou adresu funkce, která by tímto způsobem mohla být otestovaná.

Adresu můžeme získat krokováním programu při běhu na vývojovém hardwarovém kitu. Ten bohužel nebyl pro účely psaní práce k dispozici, a tak tento postup pro fuzz testovaní pomocí QEMU persistentního módu je pouze návrh dalšího možného postupu pro testování.

Kromě krokování programu by bylo dobré zjistit, jestli by nebylo možné sestavovat projekty jako spustitelně pozičně nezávislé. V tomto případě by nutnost krokování k zjištění adresy funkce opadla a mohly bychom počítat se získaným offsetem s příkazu nm. Vzhledem k faktu, že AFL++ pouští všechny pozičně nezávislé programy na fixní adrese 0x4.

Druhá strategie spočívala v separaci funkce z programu. Podle odborníka z firmy NXP tuto strategii používají některé firmy a v mnoha případech bývá testovací kód mnohem složitější a větší než samostatný funkční kód. Pro tuto strategii byl zvolen AFL++ aby došlo k otestování, zda žádný vstupní soubor není schopný zapříčinit nesprávnou funkcí programu. LibFuzzer byl použit k otestování práce přímo s daty po obsahové stránce.

Tato strategie se dá aplikovat na jakékoliv další projekty u kterých máme přístup ke kompletnímu zdrojovému kódu a můžeme tak vyseparovat všechny potřebné funkce, které potřebujeme pro správnou funkci.

AFL++ a libFuzzer umí pracovat se stejným korpusem. Což znamená, že pokud by existoval projekt, který byl v minulosti testován pomocí libFuzerru, můžeme vzít vstupní data, která libFuzzer získal a použít je jako vstupní data pro testování pomocí AFL++ a ušetřit si tak tvorbu počáteční sady vstupních dat. Práci si můžeme ušetřit i naopak, pokud bychom měli vstupní data vygenerována po testování pomocí AFL++, můžeme je použít jako vstupní data pro libFuzzer. AFL++ a libFuzzer také umí běžet současně na jednom vstupním korpusu s daty. Můžeme tak k rozběhlému AFL++ testování projektu přidat i testování tohoto samého projektu pomocí libFuzzeru, jediný problém zde nastává že oba fuzz frameworky nevidí vygenerované vstupní soubory druhého a bylo by tak potřeba po nějaké době oba fuzz frameworky restartovat, aby mohly vidět své vzájemné poznatky.

Optimální, pro obě strategie, by bylo zjistit, jestli je opravdu možné a účinné, rozběhnutí AFL++, nebo jiné fuzz frameworky, přímo na vývojové desce. Sice jsme bez vývojové desky, schopni otestovat funkce, které nepracující s hardwarem. Avšak pokud by funkce pracovala s nějakou periférií je potřeba ji emulovat. Spolupráce fuzz frameworku a vývojové desky, by tento problém vyřešila rychleji a jednodušeji než vyřešení pomocí výše zmíněných emulací. Otestovat bez vývojové desky, nebo přístupu k periferii lze v tomto případě velmi málo, protože embedded programy pracují většinou s nějakým hardwarem.

ZÁVĚR

Cílem práce bylo zjistit, zda je možné použít zatím volně dostupné fuzz frameworky na testování embedded softwaru. A to z důvodu, že žádný takto volně dostupný fuzzer není plně specializován na testování tohoto typu softwaru. Dalším z cílů práce bylo analyzovat tyto volně dostupné frameworky.

Analýza volně dostupných fuzz frameworků probíhala pouze po stránce teoretické. Jinak řečeno byly analyzovány možnosti toho, co je možné s fuzzer testovat, a jestli třeba nepodporuje další dostupné knihovny, které jiné fuzz frameworky podporují. Pro analýzu byly vybrány čtyři nejvíce známé fuzz frameworky. Fuzz frameworky byly porovnány separátně a nezávisle na sobě. Jejich výhody a nevýhody byly následně, po dokončení celé analýzy, zaznamenány do tabulek pro větší přehlednost.

Fuzz testování bylo provedeno na volně dostupném programu audio_playback od firmy NXP. Tento projekt, který byl součástí desky MIMXRT1170-EVK, posloužil jako testovaný subjekt pro implementaci obou vzniklých strategií.

Navrženy byly dvě strategie, z nichž každá přistupuje k projektu jinak.

První strategie přistupuje k celému testování jako black box testování a následně se více přiklání ke gray box testování. To hlavně z toho důvodu, že po neúspěšném běhu AFL++ na čistě celém čistě sestaveném programu, dochází ke snaze spustit AFL++ jenom na určitou funkci v tomto programu obsaženou. Neznáme obsah zdrojových kódu a nemáme k nim přístup, ale máme sestavený spustitelný binární program, který můžeme otestovat a můžeme do něj i nahlížet. Pro tuto strategii byl zvolen fuzz framework AFL++. Bylo tak učiněno, protože AFL++ nepotřebuje žádnou fuzz target funkci, aby mohl správně fungovat. Pro správnou funkci mu stačí jenom ukázková sada vstupních dat, kterou posílá přes stdin do aplikace.

Ukázalo se však že volba AFL++ byla založena na mylném úsudku, že projekt audio_playback pracuje přímo se zvukovými soubory, které jsou mu posílány na vstup. Avšak tento projekt místo souborů v hudebním formátu pracuje s jejich názvy, podle nich následně soubory vyhledává v souborovém systému.

Implementace první strategie nebyla úspěšná. Nedošlo ke spuštění AFL++, protože jakýkoliv vstupní soubor zvukového formátu zapříčinil okamžitý pád programu. Vzhledem k úvaze, že nejspíše program nepřijímá vstup z konzole, bylo využito persistentního módu

v QEMU, který umožňuje zacyklit AFL++ pouze na určité funkci bez nutnosti vyseparovat onu funkci ze zdrojového kódu projektu.

Persistentní QEMU mód, ale k správnému běhu potřebuje přesnou adresu funkce, okolo které má AFL++ zacyklit. Přesnou adresu vzhledem k podmínkám nebylo možné získat.

Implementace první strategie nebyla úspěšná. Je možné, že na jiném embedded projektu, nebo pokud bychom mohli získat přesnou adresu funkce i u tohoto projektu, by mohla být úspěšně implementovaná. Avšak to může být cíl jiné práce.

Druhá strategie přistupuje k celému projektu jako white box testování. Pro tuto strategii byly zvoleny dva fuzzery. LibFuzzer a AFL++. AFL++ byl zvolen, aby otestoval, zda nedojde k nesprávnému běhu programu při poslání špatného souboru na vstup. LibFuzzer byl naopak zvolen, aby otestoval práci s daty, jejich zápis a čtení. To bylo z důvodu, že vyseparované funkce pracovaly se společnou cyklickou frontou, do kterého jedna zapisovala a druhá z něj data četla. Názvy obou vyseparovaných funkcí byly STREAMER_Write a STREAMER_Read. Tyto funkce se nacházely v souboru app_streamer.c

Cílem práce nebylo otestovat celý projekt se všemi originálními funkcemi, ale zjistit, jestli je možné fuzz testovat embedded software. Proto, když při separaci nebylo možné získat funkce, s kterými hlavní vyseparovaná funkce pracuje, byla funkce nahrazena volně dostupnou implementací.

Při testování projektu s pomocí AFL++ fuzz frameworku nedošlo k nalezení žádných chyb. Při testování s pomocí libFuzzeru, pro který byla definovaná funkce fuzz target, která pracovala se vstupními daty získanými od fuzzeru, byla nalezena chyba ve funkci ringbuf_write. Tato chyba spočívala v přetečení zásobníku při zapisování většího množství dat. Chyba byla zapříčena špatnou úpravou volně dostupné implementace. Ve volně dostupné implementaci se počítalo s možností zapsat jeden byte najednou, zatímco v upravené verzi se počítalo s možností zapsat více bytu naráz.

Závěrem je možno říci, že fuzz testování může pomoc s nalezením nedostatků a chyb v dřívější fázi vývoje embedded softwaru. Čím dříve se na defekt narazí tím jednodušeji ho půjde odstranit. Vzhledem k tomu, že většina embedded softwaru je používaná v odvětvích, které mohou mít vliv na lidské životy, je důležité dbát na správnou funkčnost. Zabezpečit takové systému nám může pomoc zavedení fuzz testování. Fuzzingem jsme schopni najít defekty mnohdy i rychleji než ručním testováním nebo jsme schopni dosáhnout většího procenta pokrytí kódu testy. Hlavní výzva a překážka pro testování embedded softwaru spočívá

v izolování softwaru a hardwaru, při právě onom testování. Jednou z možných metod je white box testování (viz strategie č.2). Sice izolací jsem schopni následně pomocí fuzzingu otestovat tyto funkce důkladněji, než by bylo možné při black box testování na hardwaru, ale samotná separace je pracná.

SEZNAM POUŽITÉ LITERATURY

[1] ***OWASP*** [online]. [cit. 2022-02-06]. Dostupné z: https://owasp.org/www-community/Fuzzing

[2] ***ClusterFuzz*** [online]. [cit. 2022-02-06]. Dostupné z: https://google.github.io/clusterfuzz/

[3] TAKANEN, Ari, Jared DEMOTT, Charlie MILLER a Atte KETTUNEN. ***Fuzzing for Software Security Testing and Quality Assurance. 2nd Edition***. Massachusetts: Artech House, 2018. ISBN 978-1608078509.

[4] ***Fuzzinginfo*** [online]. [cit. 2022-02-10]. Dostupné z: https://fuzzinginfo.wordpress.com/history/

[5] ***FOLKLORE*** [online]. [cit. 2022-02-06]. Dostupné z: https://www.folklore.org/StoryView.py?story=Monkey_Lives.txt

[6] REIMER, Jonathan. What is fuzz testing?. ***GitLab*** [online]. San Francisco, CA: GitLab, c2022 [cit. 2022-02-12]. Dostupné z: https://about.gitlab.com/topics/devsecops/what-is-fuzz-testing/

[7] ZELLER Andreas, GOPINATH Rahul, BOHME Marcel, FRASER Gordon, HOLLER Christian. ***The Fuzzing Book*** [online]. CISPA Helmholtz Center for Information Security, 2021 [cit. 2021-11-29]. Dostupné z: https://www.fuzzingbook.org/

[8] The Morris Worm: 30 Years Since First Major Attack on the Internet. ***FBI*** [online]. Washington: FBI, 2018, November 2 2018 [cit. 2022-02-12]. Dostupné z: https://www.fbi.gov/news/stories/morris-worm-30-years-since-first-major-attack-on-internet-110218

[9] Fuzz Testing of Application Reliability. ***The University of Wisconsin Madison*** [online]. Wisconsin, 2021 [cit. 2022-02-12]. Dostupné z: https://pages.cs.wisc.edu/~bart/fuzz/

[10] SUTTON, Michael, Adam GREENE a Pedram AMINI. ***Fuzzing Brute Force Vulnerability Discovery***. Indiana: Addison Wesley, 2007. ISBN 0-32-144611-9.

[11] ***OSS-Fuzz*** [online]. [cit. 2022-02-10]. Dostupné z: https://google.github.io/oss-fuzz/

[12] ***AFL*** [online]. [cit. 2022-02-10]. Dostupné z: https://github.com/google/AFL

[13] ***Protocol-fuzzer-ce*** [online]. [cit. 2022-02-10]. Dostupné z: https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce

[14] LIM, Eugene. Life's a Peach (Fuzzer): How to Build and Use GitLab's Open-Source Protocol Fuzzer. ***Medium*** [online]. San Francisco: Medium, 2021, May 21 2021 [cit. 2022-02-12]. Dostupné z: https://medium.com/csg-govtech/lifes-a-peach-fuzzer-how-to-build-and-use-gitlab-s-open-source-protocol-fuzzer-fd78c9caf05e

[15] REIMER, Jonathan. What Is Fuzzing? [Infographic]. ***Code Intelligence*** [online]. Germany: Code Inteligence, 2020, 27 May 2020 [cit. 2022-02-12]. Dostupné z: https://www.code-intelligence.com/blog/what-is-fuzzing-infographic

[16] P. MILLER, Barton, David KOSKI, Cjin PHEOW LEE, et al. Fuzz Revisited: A Re-Examination of the Reliability of UNIX Utilities and Services. ***ResearchGate*** [online]. January 1998 [cit. 2022-02-12]. Dostupné z: https://www.researchgate.net/publication/239668581_Fuzz_Revisited_A_Re-Examination_of_the_Reliability_of_UNIX_Utilities_and_Services

[17] RAMON PALANCO, Jose. The Amazing World of File Fuzzing. ***Jpalanco*** [online]. Madrid: jpalanco, 2021 [cit. 2022-03-02]. Dostupné z: https://www.jpalanco.com/the-amazing-world-of-file-fuzzing/

[18] Fuzzing: what's behind the automated testing technique. ***IONOS: Digital Guide*** [online]. Pennsylvania: IONOS, 2020 [cit. 2022-03-02]. Dostupné z: https://www.ionos.com/digitalguide/websites/web-development/what-is-fuzzing/

[19] MUTHEE, Arthur. The Basics of Genetic Algorithms in Machine Learning. ***Section*** [online]. Colorado: Section, 2021 [cit. 2022-03-02]. Dostupné z: https://www.section.io/engineering-education/the-basics-of-genetic-algorithms-in-ml/

[20] PEDAMKAR, Priya. What is Genetic Algorithm?. ***Educba*** [online]. Mumbai: Corporate Bridge Consultancy Pvt, 2020 [cit. 2022-03-02]. Dostupné z: https://www.educba.com/what-is-genetic-algorithm/

[21] SPARKS, Sherri, Shawn EMBLETON, Ryan CUNNINGHAM a Cliff ZOU. ***Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting***. In: Twenty-Third Annual Computer Security Applications

Conference (ACSAC 2007) [online]. IEEE, 2007, 2007, s. 477-486 [cit. 2022-03-28]. ISBN 0-7695-3060-5. Dostupné z: doi:10.1109/ACSAC.2007.27

[22] HARPER, Allen, Daniel REGALADO, Ryan LINN, et al. ***Gray Hat Hacking: The Ethical Hacker's Handbook***. 5th Edition. New York: McGraw-Hill Education, 2018. ISBN 978-1-26-010842-2.

[23] LibFuzzer: a library for coverage-guided fuzz testing. ***LLVM: COMPILER INFRASTRUCTURE*** [online]. LLVM Project, c2003-2022 [cit. 2022-03-28]. Dostupné z: https://llvm.org/docs/LibFuzzer.html

[24] Not able to build LLVM from its source code. ***Stackoverflow*** [online]. New York: Stack Exchange, c2022 [cit. 2022-03-28]. Dostupné z: https://stackoverflow.com/questions/65633304/not-able-to-build-llvm-from-its-source-code

[25] Getting Started: Building and Running Clang. ***LLVM*** [online]. LLVM Project, c2022 [cit. 2022-03-28]. Dostupné z: https://clang.llvm.org/get_started.html

[26] AddressSanitizer. ***Clang 15.0.0git documentation: ADDRESSSANITIZER*** [online]. The Clang Team, c2007-2022 [cit. 2022-03-28]. Dostupné z: https://clang.llvm.org/docs/AddressSanitizer.html

[27] UndefinedBehaviorSanitizer. ***Clang 15.0.0git documentation: UNDEFINEDBEHAVIORSANITIZER*** [online]. The Clang Team, c2007-2022 [cit. 2022-03-28]. Dostupné z: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[28] MemorySanitizer. ***Clang 15.0.0git documentation: MEMORYSANITIZER*** [online]. The Clang Team, c2007-2022 [cit. 2022-03-28]. Dostupné z: https://clang.llvm.org/docs/MemorySanitizer.html

[29] ***American fuzzy lop*** [online]. Michał Zalewski [cit. 2022-02-10]. Dostupné z: https://lcamtuf.coredump.cx/afl/

[30] Build and install AFL++. ***AFLplusplus*** [online]. AFLplusplus [cit. 2022-03-28]. Dostupné z: https://aflplus.plus/building/

[31] AFL++ Features. ***AFLplusplus*** [online]. AFLplusplus [cit. 2022-03-28]. Dostupné z: https://aflplus.plus/features/

[32] Fuzzing binary-only programs with AFL++. ***AFLplusplus*** [online]. AFLplusplus [cit. 2022-03-28]. Dostupné z: https://aflplus.plus/docs/binaryonly_fuzzing/

[33] Tips for parallel fuzzing. ***AFLplusplus*** [online]. AFLplusplus [cit. 2022-03-28]. Dostupné z: https://aflplus.plus/docs/parallel_fuzzing/

[34] Understanding the status screen. ***AFLplusplus*** [online]. AFLplusplus [cit. 2022-03-28]. Dostupné z: https://aflplus.plus/docs/status_screen/

[35] [afl++] Use AFL++ instead of AFL for fuzzing.: #5046. ***Github*** [online]. San Francisco: GitHub, c2022 [cit. 2022-03-28]. Dostupné z: https://github.com/google/oss-fuzz/pull/5046

[36] Notes for using ASAN with afl-fuzz. AFLplusplus [online]. ***AFLplusplus*** [cit. 2022-03-28]. Dostupné z: https://aflplus.plus/docs/notes_for_asan/

[37] Fuzzing libxml2 with AFL++. ***AFLplusplus*** [online]. AFLplusplus [cit. 2022-03-28]. Dostupné z: https://aflplus.plus/docs/tutorials/libxml2_tutorial/

[38] Boofuzz: Network Protocol Fuzzing for Humans. ***Boofuzz*** [online]. Joshua Pereyda, c2022 [cit. 2022-03-28]. Dostupné z: https://boofuzz.readthedocs.io/en/stable/index.html

[39] Sulley: Network Protocol Fuzzing for Humans. ***Github*** [online]. San Francisco: pedramamini, c2022 [cit. 2022-03-28]. Dostupné z: https://github.com/OpenRCE/sulley#readme

[40] Installing boofuzz. ***Boofuzz*** [online]. Joshua Pereyda, c2022 [cit. 2022-03-28]. Dostupné z: https://boofuzz.readthedocs.io/en/stable/user/install.html

[41] Code Intelligence. ***Crunchbase*** [online]. San Francisco: Crunchbase, c2022 [cit. 2022-03-28]. Dostupné z: https://www.crunchbase.com/organization/code-intelligence

[42] CI Fuzz Product Tour. ***Code Intelligence*** [online]. Bonn: Code Intelligence [cit. 2022-03-28]. Dostupné z: https://www.code-intelligence.com/product-tour

[43] Findings for C/C++. ***Code Intelligence*** [online]. Bonn: Code Intelligence [cit. 2022-03-28]. Dostupné z: https://help.code-intelligence.com/findings-for-c

[44] Local Installation. ***Code Intelligence*** [online]. Bonn: Code Intelligence [cit. 2022-03-28]. Dostupné z: https://help.code-intelligence.com/local-installation

[45] Reporting for C++. ***Code Intelligence*** [online]. Bonn: Code Intelligence [cit. 2022-03-28]. Dostupné z: https://help.code-intelligence.com/reporting-for-c

[46] Continuous Fuzzing. ***Code Intelligence*** [online]. Bonn: Code Intelligence [cit. 2022-03-28]. Dostupné z: https://help.code-intelligence.com/continuous-fuzzing

[47] DOYLE, Ray. Boofuzz Introduction: Installation and Basic Usage. ***Doyler*** [online]. Raleigh: doyler.net, c2022 [cit. 2022-03-28]. Dostupné z: https://www.doyler.net/security-not-included/boofuzz-introduction

[48] Audio Samples. ***ESPRESSIF*** [online]. Shanghai: Espressif Systems (Shanghai) CO., c2016-2019 [cit. 2022-05-01]. Dostupné z: https://espressif-docs.readthedocs-hosted.com/projects/esp-adf/en/latest/design-guide/audio-samples.html

[49] Implementing Circular Buffer in C. ***Embed Journal*** [online]. India: EmbedJournal, c2022 [cit. 2022-05-05]. Dostupné z: https://embedjournal.com/implementing-circular-buffer-embedded-c/

[50] KIM, Peter. ***THE HACKER PLAYBOOK 2: Partical Guide To Penetration Testing***. South Carolina: Secure Planet, 2015. ISBN 978-1512214567

[51] Learn Docker. ***MICROSOFT*** [online]. Washington: Microsoft, c2022 [cit. 2022-05-11]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/architecture/containerized-lifecycle/what-is-docker

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

AFL	american fuzzy lop
AFL++	american fuzzy lop ++
EOF	end-of-file
ASAN	AddressSanitizer
UBSan	UndefinedBehaviorSanitizer
MSAN	MemorySanitizer
CI Fuzz	Code Intelligence Fuzz
QEMU	Quick Emulator

SEZNAM OBRÁZKŮ

Obrázek 1. Vývoj Fuzz testování 12
Obrázek 2. Fáze Fuzz testování 13
Obrázek 3. Cmake pro build LLVM z dokumentace Clang [25] 23
Obrázek 4. Ukázkový příklad použití LibFuzzeru z oficiální dokumentace [23] 24
Obrázek 5. Výpis terminálu při detekci pádu aplikace 24
Obrázek 6. Obsah souboru crash vytvořeného po pádu aplikace 25
Obrázek 7. Instalace AFL++ popsána na oficiálních stránkách [30] 27
Obrázek 8. Ukázka nastavení projektu a spuštění AFL++, při testování knihovny Libxml2[37] 28
Obrázek 9. Stavová obrazovka s informacemi o průběhu fuzz testování v aktuálním čase 28
Obrázek 10. Ukázkový script pro testování zranitelnosti v TFTP serveru ze zdroje 47[47] 31
Obrázek 11. Ukázkový výpis při spuštění boofuzzu 31
Obrázek 12. Instalace AFL++ 39
Obrázek 13. Instalace Visual Studio Code 40
Obrázek 14. Kontrola funkčnosti GCC ARM kompilátoru 40
Obrázek 15. Sestavení projektu maestro_playback 41
Obrázek 16. Konfigurace před spuštěním a samotné spuštění AFL++ 42
Obrázek 17. Nalezení vhodné funkce pomocí kombinace příkazu nm a grep 43
Obrázek 18. Výpis informací o souboru maestro_playback.elf 43
Obrázek 19. Grafické znázornění funkce AFL++ 45
Obrázek 20. Chybová hláška při běhu AFL++ na QEMU persistentní adrese 0x40300065b1 46
Obrázek 21. Vlevo originální Makro, vpravo upravené Makro 48
Obrázek 22. Kompletní funkce Main 49
Obrázek 23. Konfigurace systému a sestavení projektu 49
Obrázek 24. Běh AFL++ fuzz frameworku 50
Obrázek 25. Fuzz target funkce 51
Obrázek 26. Chyba nalezená pomocí libFuzzeru 52

SEZNAM TABULEK

Tabulka 1. Nejčastější chyby nalezené s použitím Fuzz testování v roce 1995[16] ..14

Tabulka 2. Výhody a nevýhody fuzz testování [15] [18] 15

Tabulka 3. Výhody a nevýhody libFuzzeru..26

Tabulka 4. Výhody a nevýhody AFL++...29

Tabulka 5. Výhody a nevýhody Boofuzz 32

Tabulka 6. Výhody a nevýhody Code Intelligence Fuzz...34

SEZNAM PŘÍLOH

Příloha P I: Obsah CD

PŘÍLOHA P I: OBSAH CD

Obsah CD:

- Izolovane_Funkce – obsahuje zdrojový kód použitý k implementaci Strategie č. 2