Implementace fuzz testování pro embedded systém

Bc. Pavlína Kulhavá

Diplomová práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

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

Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Pavlína Kulhavá**
Osobní číslo:	**A22302**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Implementace fuzz testování pro embedded systém**
Téma práce anglicky:	**Implementation of Fuzz Testing for an Embedded System**

Zásady pro vypracování

1. Zpracujte literární rešerši na téma fuzz testování a možnosti jeho využití při vývoji software.
2. Vyberte fuzz framework vhodný pro testování embeddec aplikací.
3. S využitím zvoleného frameworku implementujte fuzz testování pro vybraný projekt z programového balíčku MCUXpresso a zvolenou vývojovou desku.
4. Proveďte fuzz testování pro tři konfigurace: s přímo připojenou vývojovou deskou, na simulátoru a s využitím vzdáleného volání testované funkce.
5. Uvedené tři způsoby testování srovnejte a vyhodnoťte možnosti jejich použití s ohledem na výkonnost a náročnost použití.

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

Seznam doporučené literatury:

1. HARPER, Allen, Daniel REGALADO, Ryan LINN. *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 Hackers Playbook 2: Practical Guide To Penetration Testing*. South Carolina: Secure Planet, 2015. ISBN 978-1512214567.
3. Køien, G.M., Øverlier, L. A Call for Mandatory Input Validation and Fuzz Testing. *Wireless Pers Commun* (2023). https://doi.org/10.1007/s11277-023-10431-2
4. SUTTON, Michael, GREENE Adam a AMINI Pedram. *Fuzzing Brute Force Vulnerability Discovery*. Indiana: Addison Wesley, 2007. ISBN 0-32-144611-9.
5. TAKANEN, Ari, DEMOTT Jared, MILLER Charlie a KETTUNEN Atte. *Fuzzing for Software Security Testing and Quality Assurance. 2nd Edition*. Massachusetts: Artech House, 2018. ISBN 978-1608078509.
6. ZELLER Andreas, GOPINATH Rahul, BÖHME Marcel, FRASER Gordon, HOLLER Christian. *The Fuzzing Book* [online]. CISPA Helmholtz Center for Information Security, 2021. Dostupné z: https://www.fuzzingbook.org. [citováno. 2021-11-29].

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

Datum zadání diplomové práce: **5. listopadu 2023**
Termín odevzdání diplomové práce: **13. května 2024**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN
L.S.

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

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

Ve Zlíně dne 5. ledna 2024

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové 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 diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- 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 – diplomovou 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 diplomové 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 diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 10.5.2024

Pavlína Kulhavá, v.r
podpis studenta

ABSTRAKT

Tato diplomová práce se zabývá implementací fuzz testování pro embedded zařízení. Cílem práce bylo implementovat fuzz testování pro testování přímo na zařízení. K tomu byly použité tři přístupy, a to testování pomocí simulátoru, přímo na zařízení a s použitím vzdáleného volání procedur (eRPC). Teoretická část popisuje dosavadní snahu o implementaci, včetně možného použití fuzz testování v rámci projektu. Praktická část pak poznatky získané v teoretické aplikuje na implementaci výše zmíněných třech přístupů pro AFL++ fuzz framework. K testování je použita vývojová deska od NXP mimxrt1064_evk a ukázkový projekt SHELL volně dostupný z mcuxpresso SDK. Výsledkem této práce je srovnání složitosti implementace, rychlosti testování a použitelnosti jednotlivých přístupu pro testování na reálných zařízeních.

Klíčová slova: Fuzz testování, Fuzzer, Embedded systém, Testování, Embedded Simulátory, AFL++, LibFuzzer, Renode, QEMU

ABSTRACT

This thesis deals with the implementation of fuzz testing for embedded devices. The aim of the thesis was to implement fuzz testing for testing directly on the device. To do this, three approaches were used, namely testing using a simulator, directly on the device and using remote procedure calls (eRPC). The theoretical section describes the implementation efforts to date, including the possible use of fuzz testing within the project. The practical part then applies the knowledge gained in the theoretical part to the implementation of the above three approaches for the AFL++ fuzz framework. The NXP mimxrt1064_evk development board and the SHELL sample project freely available from the mcuxpresso SDK are used for testing. The result of this work is a comparison of the implementation complexity, testing speed and usability of each approach for testing on real devices.

Keywords: Fuzz testing, Fuzzer, Embedded system, Testing, Embedded Simulators, AFL++, LibFuzzer, Renode, QEMU

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 diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

Fuzz testování v poslední době nabývá na oblíbenosti obzvláště v oblasti bezpečnosti a to právě kvůli jeho základní vlastnosti automatizované generace testovacích dat. Díky tomu je mnohem snazší odhalit chyby v kratší době než klasickými způsoby testování. Povětšinou existují již různé implementace frameworků, které obstarávají základní koloběh fuzz testování (generace/mutace dat, předání dat na vstup, vyhodnocení výsledů). Tyto implementace jsou, ale primárně určeny, aby běžely na výkonných počítačích a nejsou uzpůsobeny k práci s méně výkonnými systémy.

Proto vzniká otázka, jak moc je možné a efektivní tyto frameworky využívat při testování embedded systému. Samotná generace/mutace dat je sice na hostitelském systému rychlá a samotné testování jednoho testovacího případu je v nejhorším případě v rámci sekund. Na méně výkonném zařízení můžeme předpokládat zhoršený čas a tady i pokles nějaké výhody právě v užití tohoto způsobu testování. Zároveň je zde také otázka, jestli během v simulátoru jsme schopni zjistit všechny zranitelnosti a chyby, které by se mohli vyskytnout při reálném použití softwaru na příslušném hardwaru s přístupem k existujícím periferiím. Ty jsme totiž během exekuce testů mimo reálný cíl nuceni buď nahrazovat nebo je nebrat v potaz. Tím může dojít ke zkreslení testovacích výsledků.

Existují právě tři možné přístupy k testování embedded systému pomocí fuzz frameworků neboli fuzzerů.

Běh v simulátoru. Některé simulátory mají podporu právě fuzz testování a mnoho fuzz frameworků má naopak již implementaci s použitím některého ze simulátoru. Je zde, ale základní otázka, jak moc dobře jsme schopni nasimulovat zvolený HW do virtuálního prostředí a zachovat přesné chování systému při zvýšeném toku dat. Například je zde šance, že by mohlo dojít k přetečení sériového komunikačního rozhraní a tím pádem i k určitému zkreslení výsledků. Systém se může tvářit, že se nic neděje a že stíhá, zatímco data se vůbec nemusí dostat na vstup programu.

Běh přímo na vývojovém kitu. Některé fuzz frameworky mají pevně nastavený cíl v závislosti, na jakém systému byly sestaveny. Je tedy možné sestavit fuzzer přímo na cíli testování. Dalším možným přístupem je komunikace s fuzzerem přes ladící rozhraní. Ovšem zde můžeme předpokládat, že dojde k poklesu výkonosti oproti první variantě, protože budeme muset počítat ještě s časem, než se vůbec data dostanou na vstup a než se výsledek promítne zpátky do fuzzeru.

Poslední možným přístupem je použít vzdálené volání funkce pomocí Remote Procedure Call. Zde bychom mohli předpokládat vylepšení rychlosti oproti přímému běhu na HW. A to vzhledem k tomu, že nejnáročnější část, samotné generování a mutace dat, probíhá na výkonnějším hostovi. Také můžeme minimalizovat šanci, že bychom přehlédli zranitelnost, kterou je možné vyvolat pouze na reálném hardwaru, a ne v jeho simulaci.

Cílem práce je zjistit, jak moc se výkonnostně liší spuštění vybraného fuzz frameworku přímo na hardwarovém zařízení, v simulátoru a vzdáleného volání funkce s pomocí Remote Procedure Call.

I. TEORETICKÁ ČÁST

1 FUZZ TESTOVÁNÍ

Fuzz testování je čím dál oblíbenější metoda testování softwaru, zejména k nalezení chyb a zranitelností, které by mohly ovlivnit funkcionalitu a bezpečnost systému. K tomu je využívaný pseudonáhodný generátor dat s cílem vyvolat chybu nebo najít novou doposud skrytou zranitelnost. Fuzz testování je dynamická metoda testování, tudíž během běhu je software analyzován. K tomu, aby ale byla chyba odhalena je potřeba spustit software několikrát, protože dynamická analýza umožňuje najít chyby jenom pokud se je během běhu programu povede vyvolat. [7]

1.1 Techniky používané ve fuzz testování

Během vývoje bylo do fuzzerů implementováno mnoho technik, které zlepšují jejich efektivitu (mutace testovacích dat, zpětnou vazbu od pokrytí testovaného kódu, statickou analýzu a tak dále). Jednotlivé techniky nabízejí také různé benefity například mezi rychlostí, přesností nebo i automatizaci samotného fuzz testování. [8]

Jednotlivé techniky můžou být rozděleny do tří skupin v závislosti na cílech jejich primárního zaměření v struktuře fuzzerů a to na 3 níže popsané:

1. techniky zabývající se generací dat
2. techniky dynamické analýzy
3. techniky statické analýzy.

1.1.1 Techniky zabývající se generací dat

Fuzz testování primárně stojí na mutaci a generaci nových vstupních testovacích dat. Proto techniky, které používané k výběru a mutaci náhodného čísla nebo pole čísel (seed), které ovlivňují sekvenci v jakém jsou generované nové testovací vzorky jsou nedílnou a důležitou součástí jakéhokoliv vybraného fuzz frameworku [8].

Ty mohou obsahovat jednu nebo více níže vypsanou techniku. U starších fuzzerů se jedná o náhodnou mutaci a generaci u novějších můžeme předpokládat i zapojení technik umělé inteligence při mutování nebo volení mutujících dat.

Avšak většina fuzz frameworků používá i další analytické techniky a nestačí jim jenom dobře zvolit rodiče mutantů. Důležitá je i například znalost struktury vstupních dat anebo znalost jaké procento kódu je pokryto testy. Proto nelze tvrdit, že jenom dobrou implementací některé z technik pro generování dat vytvoříme nejvíce efektivní fuzzer.

1.1.1.1 Náhodná mutace

Jednoduchá a rychlá technika na implementování, která spočívá v generování nového vstupu, z předem definovaného seedu, náhodnou mutací některých polích. Tyto fuzzery, které mají základ na náhodné mutaci byly velmi používané pro bezpečnostní testování. Kromě jednoduché a rychlé implementace je zde velmi dobrá škálovatelnost, protože není závislá na žádných komplexních výpočtech nebo programových monitorovacích technikách, na druhou stranu ji ale chybí informace o aktuálním stavu programu. [8]

1.1.1.2 Gramatická reprezentace

Gramatika je soubor pravidel, která popisují strukturu vstupních dat.

Technika gramatická reprezentace řeší jeden z problémů techniky náhodné mutace, a to neefektivnost nalezení chyb, které se nacházejí hlouběji v struktuře programu, protože většina vstupů je zamítnuta během prvních několika exekučních kroků. Informace o gramatice vstupních souborů je možno získat automaticky pomocí statické analýzy anebo manuálně napsáním takzvaných gramatických souborů a kódů. [8]

Gramatické soubory obsahují právě ony zmíněná pravidla popisující vstupní strukturu a výše zmíněné kódy. Pokaždé když dojde k rozpoznání nějakého definovaného pravidla je spuštěn odpovídající kód. Kromě těchto dvou věcí gramatické soubory obsahuje také podprogram pro provedení základního vstupu. [9]

Gramatická reprezentace je nejvíce účinná technika při testování aplikací, které mají komplexní strukturu vstupních dat. Avšak narozdíl od náhodné mutace, je zde potřeba mít znalost cíle testování nebo alespoň vstupních dat. Navíc pokud program vyžaduje specifický formát, je tento formát typicky psán manuálně a tím pádem celková implementace grammar based fuzzeru je časově náročná. [8]

1.1.1.3 Algoritmy plánování

Proveditelné metody maximalizace výstupů fuzz testování pomocí optimalizované strategie volby seedu a mutace se souhrnně označují za algoritmy plánování. Tyto algoritmy můžou obsahovat mnoho jiných algoritmů například algoritmus simulovaného žíhání anebo Markov. [8]

1.1.2 Techniky dynamické analýzy

Dynamická analýza je proces, kdy během exekuce programu získáváme dynamicky další informace například o problémech s pamětí nebo i pádu aplikace [10]. V rámci fuzz frameworků se pak tyto získané informace používají při generování dalších vstupních dat [8].

1.1.2.1 Pokrytí kódu testy

Pokrytí kódu testy je jednouchý a zároveň i velmi efektivní způsob, jak získat informace o vhodnosti a úspěšnosti použitých vstupních dat. Framework si zaznamenává, v jakých větvích kódu už byl a pokud se mu povede vstupními daty dostat někam dál zařadí si je jako užitečná, pokud ne testovací data zahodí. [7]

Většina fuzzeru má už implementovanou základní analýzu pokrytí kódu testy v sobě. Proto také dost závisí na takzvaném korpusu [11]. Korpus označuje, v tomto slova smyslu, skupinu vstupních dat, které může framework použít jako odrazový můstek při dalším generování a hledání vstupů. Tyto uživatelem předdefinované vstupy by měly být úspěšné, protože pokud dojde k selhání všech vstupních souborů, nemá se fuzz framework od čeho odrazit, a tedy s čím dál pracovat. Funguje pak jako náhodné lokální prohledávání.

Fuzzer má tak zpětnou vazbu kam a jak daleko se dostal s aktuálními vstupními daty, a tak se během testování fuzzer pohybuje konstantě tak aby došlo ke zlepšení hodnoty pokrytí kódu testy. Právě díky tomu jsou fuzzery řízené pokrytím kódu testy více efektivní než náhodné. Avšak generování komplexních dat a dostání se do komplexnějších větví je pro tento typ fuzzerů stále náročné. [11]

1.1.2.2 Symbolické dynamické spuštění

Dynamické symbolické spuštění je metoda, která je schopná definovat možná vstupní data programu. Princip spočívá v použití takzvaných symbolických hodnot, které jsou programu předány na vstup. Tyto hodnoty jsou následně sbírány na konci cesty načež dochází k inverzi omezení, aby mohla být vygenerována nová cesta programu. Jako poslední krok je generace nových vstupních dat pomocí řešitelů splnitelnosti modulo teorie (SMT). [8]

Splnitelnost modulo teorie je problém, který rozhoduje o splnitelnosti matematického vzorce. Jestli je možné ho řešit v dané rovině nebo není. K tomu, aby se rozhodnul nad splnitelností, se používají řešitelé SMT. [45]

1.1.2.3 Dynamická analýza skvrn

Dynamická analýza skvrn se principálně podobá symbolickému dynamickému spuštění, je možné je také zkombinovat k vylepšení přesnosti fuzz testování. Slouží k odvození vlastností vstupních dat a jaký offset na vstupu ovlivňuje podmínkovou větev. Tyto informace se pak používají k efektivnímu zvolení seedu a jeho mutaci. [8]

Dynamická analýza skvrn narozdíl od symbolické posílá na vstup označené vstupní data tagy. Ty můžou definovat jakým způsobem má program vstupní data zpracovat a které elementy programu jimi byly poskvrněna. [8]

1.1.3 Techniky Statické analýzy

Technika, která se skládá z grafu analýzy průběhu programu a řezy toky dat. Graf analýzy programu je, jak název napovídá, grafické znázornění programu. Uzly jsou znázorněny jako základní bloky s hranami, které reprezentují cestu spuštěného programu k zranitelným místům. Následně přímým nebo zpětným průchodem této analýzy pomocí řezu toku dat vyseknout části, které nějakým způsobem ovlivňují počítanou hodnotu. [8]

1.2 Jak zvolit správný fuzzer

Existuje několik fuzz frameworků, které můžeme použít pro testování softwaru. Povětšinou jejich odlišnost spočívá v použitých technikách řečených v předchozí kapitole ale také i v primárním cíli testování. Některé frameworky, s pomocí simulátorů, podporují plně black box testování (neznámé vnitřní strukturu programu, povětšinou nemáme přístup ke zdrojovému kódu) a tak nám stačí mít jenom sestavený program, který můžeme otestovat. Naopak pro některé frameworky je znalost kódu klíčová, protože používá svoje funkce, které vkládá do zdrojového kódu programu. Tyto kódy pak pomáhají zlepšit a mnohdy i zrychlit testování, protože fuzzer je schopen lépe mapovat testovaný cíl.

Nejznámější frameworky jsou například již zmíněné AFL[12], LibFuzzer[11] ale také i AFL++[13], který vychází z AFL[12] a je stále, narozdíl od jeho předka, udržovaný a aktualizovaný.

Existuje, ale nespočet dalších, menších nebo i větších projektů. Nejrozšířenější jsou pak fuzz frameworky, které přebírají základ z výše zmíněných fuzzerů. Ty povětšinou rozšiřují funkcionalitu, implementují nové způsoby, jak zvýšit rychlost a přesnost testování anebo upravují testovací cíl. Většina z nich je pak, stejně jako jejich rodiče, volně dostupná.

Právě v závislosti na cíli testování, vstupních datech ale i použitém jazyku bychom měli volit fuzz framework. Některé fuzery mají totiž i vlastní kompilátory, které umí pracovat s jejich definovanými flagy anebo vkládat mezi náš testovaný kód jejich pomocné funkce, které pak využívá fuzz framework během samotného testování. Jedná se například o LibFuzzer, který je úzce spjatý s LLVM knihovnou anebo i AFL++, který má upravený clang a gcc kompilátor. AFL++ sice umožňuje testovat kód, který nebyl sestaven pomocí jejich modifikovaných kompilátoru, ale nesporná výhoda, kterou tyto modifikace přinášejí je zrychlení testování a lepší generování vstupních dat díky tomu, že modifikace jsou schopny přidat k našemu kódu pomocné funkce, kterými je AFL++ schopno zmapovat více větví kódu. [13]

1.3 Použití

Jak již bylo zmíněno fuzz testování je populární v oblasti bezpečnosti díky možnosti vygenerovat za krátký časový úsek vstupní data, která by byla schopná vyvolat nesprávnou funkcionalitu, nebo třeba i shodit celý systém. O popularitě a aktuálnosti fuzz testování svědčí i fakt, že i Google na testování svých projektů používá fuzzer, konkrétně ClusterFuzz [14]. Ten implementoval také do open-source Continous Integration frameworku pro testování open source projektů, ten k srpnu 2023 eviduje nalezení přes 36 000 chyb, identifikaci a opravení přes 10 000 zranitelností v 1 000 projektech [15].

Obzvláště kritické pak může být, pokud program přímo nebo nepřímo přistupuje k internetu. V tomto případě, zvlášť pokud jednotlivé programy jsou schopny spolu komunikovat, nejsou v ohrožení pouze data jednoho uživatele, ale mnoho jiných, i třeba celé organizační struktury. [3]

1.3.1 Validace vstupní data

Systém by měl vždy kontrolovat přípustnost vstupních dat, právě totiž nesprávné zacházení se vstupními daty je povětšinou hlavní důvod, proč se útočník dostal do částí kam by neměl mít přístup. Jednou z možností, jak narušit funkcionalitu programu špatnými daty, pro které nemáme ošetřené zacházení je přetečení paměti zásobníků anebo vložením SQL příkazů do vstupních textových polích. [3]

Právě přetečení zásobníků bylo historicky jeden z nejčastějších problémů špatného zacházení se vstupními daty. Pokud dojde totiž k přeplnění zásobníku, tyto data, která se již nevejdou do zásobníku budou zapsána do paměti, která je ale rezervovaná pro jiné účely. To může ovlivnit část paměti, která odkazuje na return volané funkce, výsledkem celého útoku

pak může být, že dojde k spuštění kódu vloženého útočníkem místo kódu, který by byl na konci funkce spuštěn za normálních podmínek. [3]

Přestože přetečení zásobníků je jeden z nejstarších a nejjednodušších způsobů, jak narušit funkcionalitu systému stále se jedná, podle statistiky z roku 2021, o jednu z nejčastějších zranitelností ze všech existujících [16].

Jak vyplývá z výše uvedených příkladů hlavní použití fuzz frameworků spočívá v ulehčení a automatizaci vstupních dat. Což má za následek nalezení zranitelností, které bychom při manuálním testování hledali patrně delší časový úsek. Obzvláště pak chyby, které jsou spojené se špatnou správou paměti. Je mnohem jednoduší a efektivnější automatizovat generaci většího množství dat než ručně testovat, jestli je vstup nebo výstup, schopný zvládat manuálně vložená vstupní data.

1.3.2 Funkcionalita programu

Kromě jednotlivých funkcích bychom měli testovat i jejich závislost na ostatních funkcích, v případě embedded systému i na požadovaných periferiích. Může se totiž zdát, že jednotlivé funkce fungují v pořádku, pokud ale spustíme programu může dojít k pádu systému, protože některé funkce nemusí v kontextu jiných fungovat zcela správně nebo jejich logika neodpovídá logice aplikované v systému.

1.3.3 Regresní testování

Jedním z oblíbených částí testování, kdy se fuzz framework používá je regresní testování. To se povětšinou doporučuje implementovat jako součást CI/CD pipeline. Použití fuzzerů, jakou součást pipline, kdy je kód testován při každé změně, je doporučováno hlavně z důvodu krátké zpětné vazby, jestli nová funkcionalita je v pořádku. Problém nastává, že fuzzer je doporučováno nechat běžet delší čas, aby byl schopen se dostat i do hlubších částí programu. Tím může docházet k dlouhé blokaci pipeline, proto je doporučováno, pokud implementujeme fuzz testování, spouštět testování během hodin, kdy máme jistotu, že vytíženost, aktualizace, implementace nových funkcí bude nízká. Například v nočních hodinách. [17]

Regresní testování je pojem, který definuje typ softwarového testování prováděného při změně kódu (přidání funkce, změně funkcionality, opravě chyby atd.) k ověření že nedošlo k implementaci nových nebo již známých chyb [18]. Hlavním požadavkem na regresní testy je pravidelnost jejich spouštění například pokaždé při změně kódu, v určitou denní dobu, v určitou hodinu nebo také i určité dny v týdnu.

Vzhledem k tomu, že většina fuzz frameworků uchovává data, která zapříčinila chybu v programu je možné tyto data použít jako základní vstupní data pro regresní testování, nemusíme tedy začínat pokaždé od začátku s prázdným nebo základním korpusem dat [17]. Jsme tím i lépe schopni zajistit, že chyby, které byly již dříve odhaleny, se nepovedlo znovu implementovat do nové verze programu, protože s daty, které zapříčinili chybu začínáme testování.

2 SIMULÁTORY EMBEDDED ZAŘÍZENÍ

Vzhledem k tomu, že ne vždy máme k dispozici vývojovou platformu, nebo nemáme finance abychom používali pro vývoj reálnou platformu, například pokud chceme implementovat jenom návrh a nevíme, jestli budeme ve vývoji dále pokračovat, existují simulátory embedded systémů [19]. Tyto simulátory jsou schopné emulovat chování systému. Jejich nespornou výhodou, kromě toho že se na nich dát simulovat jakékoliv zařízení (pokud máme dostupnou jeho specifikaci), je i to, že využívá hardwarových součástí hostitele, na kterém běží a tudíž může být výkonnější než skutečné zařízení. To je ale i zároveň nevýhoda, pokud chceme otestovat hraniční hodnoty, které by mohly být kolapsové pro reálné zařízení. Simulátor může tyto hodnoty zpracovávat bez větších problému, reálný HW nemusí tyto hodnoty stíhat zpracovávat vůbec a může dojít i k poškození periférií.

Primárně simulátory můžeme rozdělit do dvou kategorií částečné a úplné. Úplná emulace je lepší než částečná, protože v ní dochází k emulaci celého systému. Pokud používáme jenom částečnou emulaci je potřeba zajistit všechny potřebné vstupy a periférie. [10]

2.1 Proč používat simulátory během vývoje

Embedded systémy jsou vyvíjené pro jasně dané cílové zařízení. To se od hostitelského systému, počítače, liší. Přestože se může zdát, že program je plně funkční na hostitelském systému, na cílovém zařízení tomu tak být nemusí [19]. Čím dřív se podaří odhalit chybovost na cílovém zařízení tím méně cenově náročné je problém vyřešit.

Kromě nesporné výhody začátku vývoje i bez cílového zařízení, je i kontrolované testovací prostředí, které simulace poskytuje. Protože simulace je vlastně další program je možné ho jednoduše restartovat nebo měnit testovací hodnoty téměř okamžitě. Umožní tím vývojáři testovat bez nutnosti restartování nebo změny na fyzickém zařízení. Zároveň, protože simulátor je počítačový model, není ovlivněný chybami prostředí a zároveň v něm lze otestovat, jak se bude program chovat během chyby. [20]

Další výhodou je možnost zjištění, jestli se chyba nachází v softwaru anebo některá z periferií nefunguje správně. Otázkou zde je, jak moc lze věřit implementaci periferií v simulátoru. Periferie se mohou jevit implementovaná správně a funkčně. Například ukázkový program na nich běží bez problému, ale pokud dojde k nahrání a pokusu spustit tentýž program přímo na cílovém zařízení mohou se vyskytnout chyby, které se vlivem špatné implementace periferií v simulaci neprojevili. Pokud však náhle přestane předtím funkční program

fungovat na cílovém zařízení, ale v simulátoru pořád funguje, je možné že jen některá z periferií přestala fungovat, pokud ovšem ale program nefunguje ani na simulátoru nejspíše je za tím implementovaná chyba ve zdrojovém kódu. [20]

2.2 Typy emulací

Existuje několik cest, jak můžeme přistupovat k emulaci [21]:

1. Emulace v uživatelském prostředí pro spustitelné soubory nebo služby
2. Emulace v uživatelském prostředí se simulovanými soubory
3. Emulace systému bez zavaděče
4. Emulace celého systému

Typy emulací jsou důležité pro výběr simulátoru, protože vycházejí z toho, co a jakým způsobem chceme simulovat.

V dalších kapitolách jsou popsané některé z populárních simulátorů.

2.3 QEMU

Asi nejznámější simulátor je QEMU. Právě díky jeho rozšířenosti a povědomí o něm má tento simulátor velkou podporu v různých fuzz frameworcích. Jedná se o generický a volně dostupný simulátor [22].

Disponuje různorodými módy, z nichž nejběžnější je emulace celého systému. V tomto módě poskytne plný virtuální stroj k běhu emulovaného systému. Lze v něm, za předpokladu že emulovaný systém a hostitelský systém jsou stejné, emulovat kompletní systém a také používat pomocníky jako KVM, Hypervisor, Hax nebo Xen. [21]

QEMU nelze jednoduše rozšířit, nedisponuje svým skriptovacím jazykem pro automatizaci testů a přidávání komplexní chování zařízení není jednoduché [19].

Díky tomu, že je open source, má několik dalších forků, které rozšiřují jeho funkcionalitu i seznam podporovaných systémů a architektur. Taktéž kvůli jeho popularitě je, jakožto fork, součástí několika fuzz frameworků.

2.3.1 Podpora ve fuzzerech

QEMU je nejčastěji podporovaný emulátor ve většině volně dostupných fuzz frameworcích. Ovšem to je i jeho hlavní problém. Verze, implementované ve fuzzerech, jsou ve většině případu starší verze.

Například AFL podporuje QEMU ve verzi 2, jeho fork AFL++ podporuje pak QEMU ve verzi 5 [13]. Obě tyto verze jsou, vzhledem k tomu že aktuální verze QEMu je 8 [22], zastaralé a obsahují chyby, které jsou ve vyšších verzích již ošetřené. Například problémy se statickým linkováním knihoven ve verzi 5 [23]. Čímž vzniká problém při spouštění AFL++ v QEMU módě. Binární soubor spadne ihned při vstupu do funkce main a není možné jej tedy fuzz testovat v QEMU módě. Pokud se tento stejně sestavený soubor spustí v AFL například v Renode módě, funguje naprosto bez problému.

QEMU také nabízí v některých fuzzerech (AFL++ a některé jeho forky) podporu persistentního módu.

Persistentní mód je funkcionalita, která umožní volat jenom určitou funkci nacházející se na předem definovaném rozsahu adres. Tím dochází ke zrychlení fuzzování, protože binární soubor se nespouští při každém novém testovacím případu od znova, ale na předem definované adrese. Odpadne tedy nutnost například znovu inicializování periférií a nastavovaní systému. V případě kombinace s AFL++ je zrychlení volání exekucí za sekundu dvoj až pěti násobné. [24]

2.4 UNICORN

Unicorn je simulátor, který, jako mnoho dalších, vychází z výše zmíněného QEMU. Jedná se o CPU emulátor, jenž může emulovat několik architektur procesorů (například ARM, Mips, Sparc atd.). Kromě vlastních podporovaných architektur je schopen také podporovat architektury, které mají podporu v QEMU. [25]

Disponuje vlastní API, která umožňuje snadnější práci s emulátorem. Kromě základních funkcionalit, jako spouštění a zastavení simulace, umožňuje také spravování paměti pomocí takzvaných hooků, čtení z paměti anebo registrů. Umožňuje také dynamické alokování v průběhu simulace. Tím může spravovat nesprávnou nebo chybějící paměť. Hlavní výhodou této API je také to, že není závislá na žádné architektuře, takže se dá použít pro jakýkoliv projekt. [25]

Stejně jako celé Unicorn jádro i API je napsaná čistě v jazyku C, ale má také vazbu a dostupnost pro použití s jazykem Python. Dosahuje vysoké výkonosti s použitím kompilačních just-in-time (JIT) technik. [25]

Just in time kompilace spočívá v optimalizaci výkonu aplikace kompilací nativního strojového kódu za běhu. Využívá se hlavně pro Java programy. [26]

2.4.1 Rozdíly od QEMU

Jeden z hlavních a znatelných rozdílů mezi Unicorn a QEMU je zaměření emulace. Zatímco QEMU se primárně soustředí na emulaci celého systému se všemi jeho periferiemi a procesorem. Unicorn se hlavně soustředí na dobrou emulaci procesorové jednotky a správa ostatních součástí a komponent je zanedbána a není obstarána simulátorem. Tím že emulujeme jenom CPU jsme schopni exekuovat surový kód bez kontextu, který právě nejsme schopni dobře emulovat na emulátorech, které se zaměřují i na emulaci ostatních komponent systému. [27]

Další z výhod oproti QEMU je bezpečnost. QEMU má totiž špatnou historii se spoustou zranitelností, které mohou být využity k proniknutí do hostitelského systému. Sice se chyby většinou týkají BIOSu, firmwaru a žádné se netýká přímo komponenty obstarávající CPU emulaci. Unicorn na svých stránkách tvrdí, že je mnohem bezpečnější právě díky malé ploše, na které mohou být útoky zaměřeny (CPU). [27]

Další výhody oproti QEMU jsou: podpora více vláknových operacích, dynamická instrumentace a správa paměti pomocí hooku za běhu emulace, a protože je Unicorn upravený tak aby obsahoval pouze funkce, které nějakým způsobem zabezpečují emulaci CPU zabírá 10x méně paměti než QEMU. [27]

2.4.2 Podpora ve Fuzz Frameworcích

Vzhledem k povaze emulátoru nemá velkou podporu ve fuzzerech. Jediná dohledatelná instance se nachází v AFL++ [13], kde je doporučen k použití, pokud nelze použít QEMU. Ovšem narozdíl do QEMU, kde stačí mít dostupný binární kód testované aplikace je u Unicornu modu potřeba mít dobrou znalost rozložení paměti a obsahu právě testované aplikace [28].

2.5 RENODE

Renode je simulátor, který se specializuje na emulaci, jak CPU včetně systém on chip, tak i ostatních periferiích a jejich komunikace mezi nimi. Ve zkratce umožňuje komplexní emulaci celého systému. Tím pádem jsme schopni plně odhadnout, jak bude software fungovat na reálném zařízení. [29]

Má již několik předpřipravených implementací vývojových desek, ale oproti QEMU disponuje velkou výhodou v možnosti snadno implementovat jakoukoliv vývojovou desku, za

předpokladu že známe její specifikaci. Tudíž ho můžeme použít k testování firmwaru pro vývojové desky, které ještě nemáme k dispozici, nebo i takové které nejsou prozatím dostupné na trhu, ale známe jejich hardwarovou specifikaci. [30]

2.5.1 Podpora ve Fuzz Frameworcích

Renode dostává větší popularity až v nynější době, a tak jeho podpora ve fuzzer není velká. Respektive, stejně jako u Unicorn, jeho jediná implementace se nachází pro AFL++ [13].

Tato implementace se nenachází přímo v oficiálním repositáři AFL++, ale v repositáři od antmicro. Jedná se o verzi, která vychází z vydané verze 4.07c, tato verze fuzz frameworku není aktuální, ale nejedná se ani o několik let starou verzi. [44]

Antmicro ve své implementaci pro AFL++ používá jiný přístup k testování než ostatní simulátoru. Fuzz testování přes Renode probíhá pomocí paměťových hooků použitých v Python scriptu. Tento python script, již implementován v základní verzi AFL++, obstarává komunikaci mezi AFL a běžící instancí Renode. [31]

Součástí implementace je i druhý python script, ve kterém je popsaná logika práce se vstupními daty získanými od AFL++. Je zde umožněno specifikovat jakým způsobem se budou data, získaná od AFL++, dále zpracována a předána na periferii. [31]

3 VZDÁLENÉ VOLÁNÍ PROCEDUR

Komunikační softwarový protokol, který se používá, pokud chce jeden program využít služby jiného programu bez nutnosti detailní znalosti sítě nazýváme vzdálené volání procedur, zkratkou RPC z anglického Remote Procedure Call. [32]

Využívá principů klient-serveru, kdy klient je program, který žádá program poskytující služby neboli server, o součinnost. Celý proces probíhá synchronně, klient čeká na dokončení požadavku na serveru. [32]

3.1 Embedded Remote Procedure Call

Embedded Remote Procedure Call, zkráceně eRPC, je implementace vzdáleného volání procedur pro embedded zařízení vyvinutá firmou NXP. [33][34]

Cílové zařízení, na kterém se volaná funkce má vykonat, může být jakýkoliv procesor, který je připojen k jednomu z definovaných přenosových kanálů, s kterými umí eRPC pracovat. Jedná se například o UART, TCP/IP nebo IPC kanály přes které je schopno posílat jednotlivé byty. Změna přenosového kanálu není, ale nějak složitá a stejné programy mohou používat jiné přenosové kanály. [33][34]

Nedílnou součástí eRPC je i definice jazykového rozhraní neboli IDL [33]. Interface Definition Language, jak zní anglický originální název, je označení pro takový jazyk, který umožní, že program, který je napsán v jednom jazyce zvládne komunikovat s jiným programem, jenž je napsán v neznámém jazyce [35]. Zatímco tedy eRPC generuje kód pro server, který ví, jak interpretovat požadavek a zavolat požadovanou funkci, IDL se používá k definovaní datových typů a RPC požadavků generátoru [33].

Vzhledem k tomu že eRPC je zaměřen na embedded zařízení je jeho celková velikost nižší než jiné implementace RPC. Což ovšem znamená i to, že nedisponuje velkým výkonem při přenosu bytů. Zároveň mu, ale také nezáleží na použití striktní API struktury a většině uživatelů stačí vytvořit jednoduchý zdrojový kód v jazyku C pro klienta i server. [34]

3.2 Použití společně s fuzz frameworky

Remote Procedure Call umožní zavolat funkci na vzdáleném zařízení. Toho je docílenou povětšinou implementací RPC přímo do zdrojového kódu testovaného programu.

Většina fuzz frameworků má podporu vlastních „main“ funkcí, u některých fuzzerů nazývaných taktéž fuzz target [11] funkce. Jedná se o funkci, která povětšinou ovládá logiku práce s daty, nemusí tomu být ale vždy. Například AFL++ [13] umožňuje implementovat do zdrojového kódu jim používané funkce potřebné k urychlení rychlosti počtu spuštění za sekundu. Tyto funkce AFL++ implementuje do kódu automaticky, pokud dochází k překladu pomocí jím úpravných kompilátorů.

Do těchto funkcí lze definovat cokoliv, od volání jedné funkce, nekonečné smyčky i to jakým způsobem bude s daty zacházeno.

Vzhledem k tomu, že všechny fuzz frameworky fungují tak, že přeberou data ze vstupu a předají je na vstup testovanému programu vybízí se možnost napevno naimplementovat Remote Procedure Call funkcionalitu do zdrojového souboru. S tím že volaná funkce bude přebírat data ze vstupu distribuována AFL++. Hlavní nevýhoda tohoto postupu pro testování embedded systému je nutnost přepsat zdrojový kód, nebo ho jiným způsobem upravit pro možnost fuzz testování. Pokud není k dispozici zdrojový kód tato možnost implementace pro testování na reálném zařízení vypadává.

I přes to, že embedded Remote Procedure Call se jeví jako ideální způsob fuzz testování, protože nejvíce náročná část generace dat se provádí na hostitelském zařízení a na cílové se dostanou už data zpracována v rámci volání funkce, nenachází se nikde zmínka, že by někdo někdy zkoušel testovat reálný embedded cíl tímto způsobem. Reálná použitelnost tohohle přístupu tak zůstává prozatím neznámá.

4 BĚH NA CÍLOVÉM ZAŘÍZENÍ

Testování přímo na zařízení, pro které je software vyvíjen je optimální přístup k testování. Na host zařízení ani v simulátoru nemůžeme zajisti, že funkcionalita programu je v pořádku, protože nemáme reálné periférie, se kterými systém komunikuje. I když tyto periférie máme dostupné v simulátoru nemůžeme s jistotou tvrdit, že jejich reakce na podněty je stejná jako na hardwaru, protože vždy předpokládáme pozitivní reakci. Tu ale na reálném zařízení nemusíme dostat třeba i vnějšími vlivy jako je rušení nebo jenom nízká výkonost testovaného zařízení.

Právě přítomnost periferií je nesporná výhoda pro testování, protože se vyhneme nutnosti nahrazovat periferie, a ne stoprocentní jistoty, že nahrazená periferie funguje správně. Také dostáváme představu o výkonosti programu přímo na cílovém zařízení, pro které bychom měli vyvíjený program optimalizovat.

Ovšem to se stává i nevýhoda při fuzz testech přímo na zařazení, rychlost oproti testování na hostitelském zařízení poklesne. Fuzz frameworky jsou totiž schopny za krátký čas vygenerovat velké množství dat, které se nemusí ani velikostně vejít na komunikační rozhraní používané pro zadávání příkazů pro testovaný program. Může dojít k takzvanému bufferingu neboli přeplnění zásobníků určeného pro UART, kdy program není schopný zpracovat velké množství dat, které byly za krátký čas umístěny na UART. Vzhledem k tomu, že fuzz frameworky nekontrolují, jestli nedošlo k přeplnění a dále posílají data na vstup může se stát, že se zařízení už nikdy z bufferingu nedostane a výsledky testování mohou být proto zkreslené.

4.1 Fuzz framework přímo na zařízení

Některé fuzz frameworky mají pevnou vazbu na Linux knihovny, například AFL++, které ve velkém používá při konfiguraci linuxové systémové knihovny. Naportovat takový fuzz frameworky přímo na zařízení, pokud na něm nejsme schopny dostat linuxový OS, například Debian, je složité.

V mnoha případech se neobejdeme bez úpravy zdrojových kódu fuzz frameworku a nahrazení těchto knihoven vlastní implementací, které by mohli být použité na bare metalovém zařízení, které nemá právě přístup k specificky systémovým knihovnám.

Pokud jsme, ale schopni dostat operační systém s používanými systémovými knihovnami je možné, například přes PuTTY, připojit se k zařízení a fyzicky na něj nainstalovat fuzz

framework, kterým se provede otestování binárního souboru. Tím, že fyzicky sestavíme fuzz framework pro cílovou platformu se vyhneme i nutnosti testovat ho v nějakém simulovanému modu, čímž minimalizujeme nutnost dodefinování systému nebo úprav v binárním souboru. To z toho důvodu, že testovaný soubor jsme schopni, po většinou, sestavit sestavenými upravenými kompilátory, které jsou sestavovány pro hostitelské zařízení.

Vyzkoušené je například sestavení AFL pro zařízení Rasberry Pi s operačním systémem verze 2020-02-13 s dvěma různými cestami implementace, a to buď sestavením ze zdrojového kódu anebo z balíčku dostupného pro unixová zařízení. [36]

Ovšem existuje i úprava AFL++, která je cílená přímo pro ARM zařízení ARM-AFL. Stejně, ale jako u předchozí implementace je potřeba mít na cílovém zařízení dostupný Linux jako operační systém, a navíc ještě zdrojový kód programu. Avšak tento fork přímo pro ARM obsahuje vlastní detektor kolizí na haldě. [7]

4.2 Fuzz framework s použitím ladících zařízení

Oblíbený způsob, co se týká počtu dostupných článků a publikací, k přístupu testování embedded systému přes fuzzery je použití extérního ladících zařízení, jakým je například Lauterbach nebo GNU Debugger.

Ladící zařízení neboli debuggery, umožňují nastavit breakpointy v běžícím, zkompilovaném, programu, získávat aktuální informaci o stavu programu, a i jeho paměti. Proto je jejich využití pro fuzz testování tak oblíbené, protože jsme schopni dostat aktuální stavové informace z cílového zařízení. [37]

4.2.1 BreakPoint

Jedním ze způsobů, jak využít debugger při fuzz testování je nastavením breakpointu v programu na jednotlivých větvích a následné kontrolování jaký breakpoint byl spuštěn jakým vstupem. Tímto způsobem jsme schopni získat informace o pokrytí a tyto informace pak použít při další mutaci vstupních dat. Tento způsob používá GDBFuzz, který využívá právě GNU Debugger [38]. [37]

GDBFuzz je řízen informacemi o pokrytí kódu, tím pádem je schopný se na základě těchto informací rozhodovat, zda vstupní soubor byl užitečný (došlo k zvýšení pokrytí nebo pádu aplikace) nebo ne. Pokud došlo k zvýšení informace o pokrytí, nebo pádu aplikace, tyto vstupní data jsou dále mutovaná, jinak dochází k jejich zahození.[38] [37]

4.2.2 Proxy

Systém, který poskytuje komunikační prostor mezi uživatelem a jiným zařízení, povětšinou internetem, můžeme nazvat Proxy Server.

Zavedení tohoto přístupu testování povětšinou spočívá v definici vlastního Proxy serveru, který zprostředkuje komunikaci mezi cílem testování a fuzz frameworkem. Cíl testování nemusí být čistě hardwarové zařízení, ale může se jednat i o síťový protokol nebo i webovou stránku. Tento přístup je používaný například i AFL++. Kdy je uživatel schopný definovat jakým způsobem se posílají testovací data cílovému zařízení a jak získaný výstup ovlivňuje fuzz framework. Například došlo-li ke získání doposud nezjištěné adresy. [39]

V případě embedded systému dává největší smysl využít k posílání dat hardwarový debugger, ze kterého jsme schopni zjistit i aktuální stav testování a nemusíme tak definovat vlastní logiku získávání stavu od zařízení.

Existuje ale i fuzz framework, který se primárně specializuje na testování přes Proxy, Avatar^2. Fuzzer využívá dynamickou analýzu, a kromě reálných zařízení je schopný analyzovat i emulátory. Tím že je vytvořen jako Proxy fuzzer je nastavení cílových testovaných zařízení jednoduší než v případech, kdy je fuzzer uzpůsoben k testování na hostitelských zařízení nebo při white box testování tj. Kdy máme přístup k testovanému zdrojovému kódu. [40]

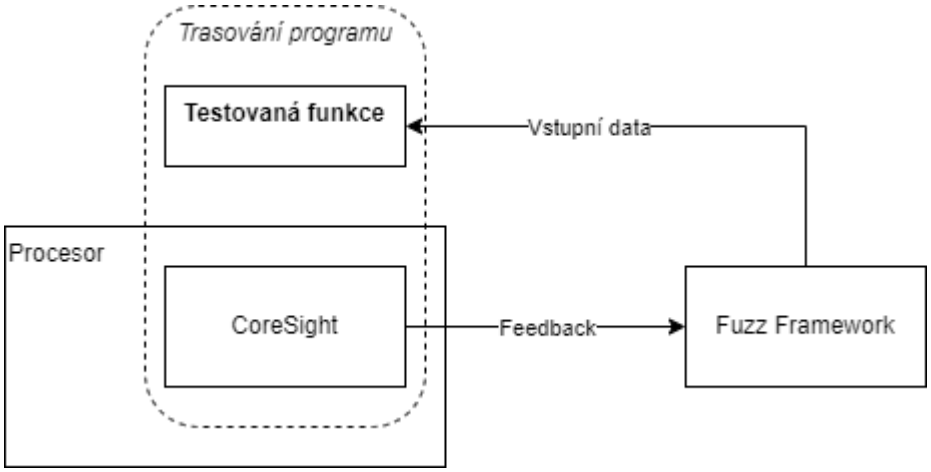
4.2.3 CoreSight

V roce 2021 vznikl i fork pro AFL++, který využívá vlastnosti některých ARM procesorů, CoreSight [41]. CoreSight je, jak už bylo řečeno, vlastnost ARM procesorů, která usnadňuje debuggování přímo pro systémy na čipu zkráceně SoC [42].

Specifikace této funkcionality se liší k architektuře od architektury, ale primárně se skládá ze tří částí a to: trasující zdroj, trasující jímka, a nakonec trasující linkování. Trasující zdroj je, jak už název vypovídá zdroj, který má za úkol generování trasování, tyto průchozí trasy se pak ukládají do trasovací jímky. Poslední část, linkování, jak název napovídá zprostředkovává připojení trasujícího zdroje k trasující jímce. Z těchto trasujících dat pak pomocí dekodéru můžeme dostat přesné znění chování programu. [41]

Celý fork spočívá ve využití CoreSight pro instrumentaci binárního souboru, kdy jsme v reálném čase schopni dostat zpětnou vazbu pro potřeby dalšího postupu v testování. Podle výstupních dat obsažených ve zdroji, se oproti jiným instrumentačním nebo trasujícím způsobům se dostáváme k velmi slušné rychlosti testování, která je pro účinný fuzzing klíčová.

Čím více dat jsme schopni v krátké době poslat na cíl tím máme větší šanci, že v krátkém čase narazíme chybu. [41]



Obrázek 1 Schéma funkcionality CoreSight

5 POUŽITÍ PŘÍSTUPŮ S FUZZ FRAMEWORKY

Neexistuje volně dostupný fuzzer, který by byl schopen testovat software pro embedded zařízení přímo na cíli. Existují již záznamy o úspěšných portech existujících frameworků a přístupu k testování těchto struktur, ale žádné z těchto řešení není volně dostupný. [7]

Přidat podporu pro testování embedded systému skrz fuzzing je náročný a dlouhý proces, protože jednotlivá zařízení se mnohdy od sebe liší ať už například v umístění jednotlivých portů nebo používání různých procesorů, které se neustále vyvíjejí. Jsme také mnohdy limitováni prostorem anebo také i použitými specifickými flagy pro kompilaci, které jsou sice podporovány použitým kompilátorem, ale při přechodu na upravené kompilátory distribuované s fuzz frameworkem není kompatibilita vždy zaručena. Například LibFuzz od LLVM nemá podporu nastavení flagu – fsanitize pro ARM procesory.

Největší problém je ale vysoké množství architektur. Nejsme totiž schopni zajistit jejich stoprocentní podporu. I přestože některé fuzzery sice slibují podporu ARM architektur nakonec z toho plyne podpora povětšinou architektur typu ARMv, ale už ne Cortex. Primární problém spočívá i ve slibované podpoře binárních souborů, kdy se převážně myslí soubory pro mobilní zařízení nebo kernely, ale ne programy určené pro bare metal embedded zařízení běžící bez operačního systému, nebo využívající některé extérní periferie.

Výše zmíněné problémy by měli představovat hlavní parametry pro vybrání vhodného fuzz frameworky. Výběr fuzzeru se, ale může lišit i v závislosti na tom, jak chceme k testování přistupovat. Avšak fuzz testování se v posledních letech stává velkým tématem i pro tyto zařízení, zvlášť v oblasti bezpečnosti, a je nejspíše jenom otázka času, než se veřejně dostupné řešení tohoto problému objeví na trhu.

5.1 Instrumentace binárního souboru

Instrumentace binárního souboru, staticky nebo dynamicky, je způsob, jakým lze upravit binární soubor vložením vlastních bloků definovaných pro zvolený fuzzer. Můžeme tak do již zkompilovaného souboru, který nejde sestavit s pomocí překladačů definovaných přímo pro fuzz framework, vložit funkce fuzz frameworku a tím zlepšit pokrytí a hledání nových užitečných vstupních dat. Nepoužitím speciálních funkcích fuzz frameworku se ochuzujeme i o lepší výkon vzhledem k tomu, že tyto funkce pomáhají lepšímu generování vstupních dat.

Hlavní problém instrumentace je časová náročnost. Celý zkompilovaný binární soubor musíme projít adresu po adrese a zjisti kdy se vyplatí vložit jednotlivé funkce a také posunout adresní prostor pro funkce poté.

Tenhle problém se dá vyřešit automatizací, existuje pár automatizovaných řešení instrumentace pro AFL++ a to jmenovitě: ZAFL, RetroWrite, Dynist, Mscema. Problém všech zmíněných implementací je ten, že většina z nich nepodporuje architektury typu ARM a primárně jsou cílená na x86_64 architektury. Dysninst sice slibuje podporu alespoň arm7 a arm8, ale při pokusu rozběhnout instrumentační nástroj na Cortex-M binárním souboru program hlásí neplatnou architekturu z čehož můžeme soudit, že podpora cortex procesorů není, v této době, dostupná. [43]

S instrumentací pracuje i výše zmíněná implementace CoreSightu pro AFL++, pro fuzz testování ARM architektur.

Tento typ přístupu k testování nalezneme povětšinou u fuzz frameworků, které pro testování používají vlastní upravené kompilátory, které jsou schopné přidat do testovaného kódu při kompilaci své pomocné funkce k lepší účinnosti fuzzování.

5.2 Běh v simulatoru

Simulátor je jeden z nejjednodušších přístupu k testování embedded systému, a to právě díky velké podpoře pro binární fuzz testování ve fuzzerech. Problém, avšak nastává, pokud chceme testovat nejenom jádro, ale i ostatní periferie. Mnoho fuzz frameworku tomuto není uzpůsobená, a tak jsme povětšinou nucení implementovat vlastní proxy, abychom mohli například v AFL++ testovat v QEMU módě celý definovaný systém.

I když je dost simulátoru implementovaných v těchto frameworcích, největší podporu má pořád AFL++. Pokud chceme testovat vlastní systém, který není obvyklý, nevyhneme se definici takzvaného harnessu, který fuzzer potřebuje pro to, aby věděl, jakým způsobem dochází ke komunikaci se zařízením.

Jako v předchozí možnosti se střetáváme s problémem podpory architektur. Ve většině případů při definování podpory binárního fuzz testování, nebo embedded systému, se primárně myslí binární soubory běžící na jádře anebo mobilní technologie.

Oproti běhu na cílovém zařízení, ale můžeme dosáhnout vyšší výkonosti v závislosti na hostovi na kterém simulace probíhá. Na druhou stranu dochází ke zkreslení výkonosti vůči reálnému zařízení.

5.3 Běh na cílovém zařízení

Pokud se chceme vyhnout definování vlastního simulátoru a zavedení případně chyby je testování přímo na cílovém zařízení dobrá volba zvlášť co se týká množství dostupných studií a dat.

Nejedná se o neprozkoumané území a mnoho studií se právě zaměřuje na tento styl testování embedded systému, protože nemůžeme říct, že je program v pořádku, pokud není otestován na zařízení, pro které je vyvíjen.

Nejčastější přístup spočívá v testování přes proxy a hardwarového či softwarového debuggeru, ze kterého jsme schopni dostat aktuální stav programu. Tento přístup se dá aplikovat pro některé fuzz frameworky, pokud podporují definování vlastní proxy, která by zabezpečovala posílání dat a získávání zpětné vazby přímo od zařízení. Tímto způsobem lze testovat i komunikační protokoly, pokud zařízení komunikuje s jiným zařízením.

5.4 Běh s použitím vzdáleného volání procedur

Doposud neprozkoumané užití v rámci testování embedded zařízení. Podobné užití jako při použití vlastní definice proxy serveru, s tím rozdílem, že na zařízení je volaná pouze testována funkce, tím pádem nedochází k takovému zatížení cílového zařízení při nutnosti spouštět znova celý binární soubor s každým vstupem.

II. PRAKTICKÁ ČÁST

6 POPIS ZVOLENÉHO HARDWARE A SOFTWARE

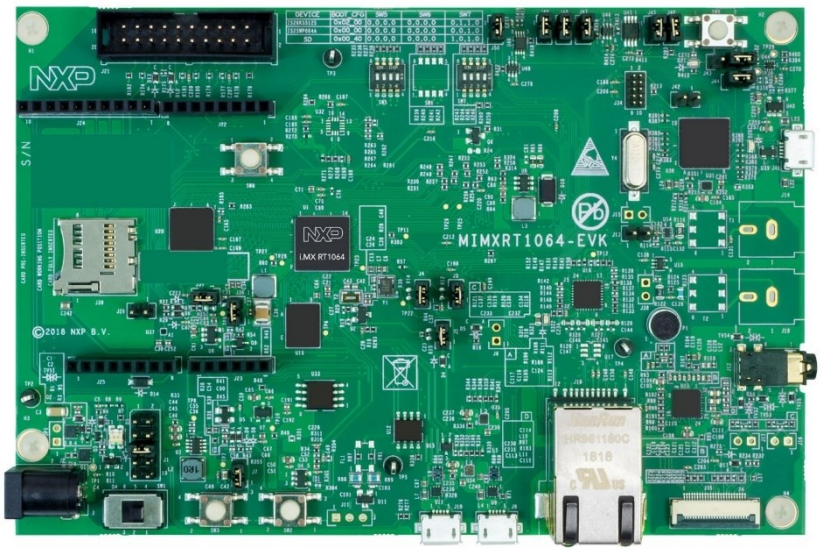
Vzhledem k tomu, aby výsledné poznatky z implementace fuzz testování pro reálné zařízení, měli vypovídající hodnotu byli fixně zvoleny použité testovací nástroje.

Vzhledem k větší podpoře fuzzerů, a i simulátoru pro systémy Linux, je celé testování prováděno ve virtuálním stroji s operačním systémem Ubuntu.

Všechny možné implementace byly prováděny pro vývojovou desku mimx1064rt-evk s volně dostupným ukázkovým programem pro práci se shellem. Dále byl zvolen i fuzz framework, který se používal pro všechny tři přístupy k testování, AFL++. Veškeré důvody a podrobnější popis softwaru a hardwaru se nachází v dalších kapitolách.

6.1 Mimx1064rt_evk

Mimx1064 je vývojová deska od společnosti NXP s procesorem i.MX RT1064, který obsahuje pokročilou implementaci ARM Cortex-M7 jádra.



Obrázek 2. Vývojová deska i.mx RT1064 EVK [46]

Primární důvod k zvolení této vývojové desky byl ten, že v simulátoru Renode, se nachází již její před konfigurace. Tudíž použitím se dalo minimalizovat chyby a nestandartní chování simulace při testování, které mohli být způsobeny špatným nastavením testovacího prostředí.

Kromě standartních portů se na desce nachází také možnost připojení jak externího ladícího zařízení, tak využití i ladícího zařízení implementovaného přímo na vývojové desce. Toto debugovací zařízení, takzvaně on board, bylo nakonec v této práci využito.

Vývojová deska obsahuje taky tři možné nastavení firmwaru lišící se v použití ladícího zařízení a způsobu nahrání binárního souboru na chip. V základní distribuované verzi od výrobce je poveleno takzvané drag and drop programování. Což znamená, že lze binární soubor přetáhnout do příslušné složky, zobrazené po připojení zařízení. V tomto módě je možné také vývojovou desku napájet stejným kabelem jakým je zajištěna sériová komunikace.

Pro potřeby testování byl firmware na desce nahrazen odpovídajícím nastavením pro použití JLink debbugeru nacházející se na desce. V tomto nastavení není již možné napájet desku stejným kabelem jakým je připojena k počítači, ale je potřeba změnit napájecí mód přesunutím jumpru J1 na pozici 3-4. Více podrobně o změně firmwaru lze najít v kapitole 8.1.1 Firmware.

6.2 Shell

Ukázkový program Shell je jeden ze základních ukázkových programů, které se dají na vývojovou desku nahrát. Je to již před vytvořený open source program, jehož cílem je při zadaní příkazu: led index on/off, rozsvítit nebo zhasnout příslušnou LED diodu. Kromě tohoto příkazu je program schopný zpracovat ještě dva následující příkazy, a to jmenovitě help a exit.

Příkaz help vypíše v připojené konzoly nápovědu a druhý jmenovaný příkaz exit, ukončí program. Jakýkoliv další text nebo jiný vstup, který by byl poslán na vstup vývojového kitu vyvolá hlášku o neznalosti tohoto příkazu a doporučí použít nápovědu.

Posílání dat na vstup probíhá přes sériový port. Vstupem je v tomto příkladě nízko napěťové uart, lpuart. Toto je ale pro potřeby testování problém, protože lpuart, jak bylo zjištěno při testování v simulátoru, není schopno v krátké době zpracovat větší množství požadavků a může snadno dojít k přetečení.

Při sestavování binárního elf souboru byl, v cmake souboru, který je používat pro sestavení programu definován parametr DEBUG_CONSOLE_TRANSFER_NON_BLOCKING, který povoluje použití neblokovacího módu, tím pádem dochází k zvýšení rychlosti a možnosti poslat na vstup více dat.

Vzhledem k tomu, že pro testování na reálném zařízení potřebujeme povolit některé debugovací funkce programu, dostupné pouze jenom při sestavení s nastavením debug, není pevně stanovena jediná možná buildovací konfigurace.

6.3 AFL++

Fuzz framework AFL++ byl zvolen hlavně kvůli jeho možnosti implementovat ho tak aby testoval skoro jakýkoliv program na jakékoliv platformě, díky AFL++ proxy. Taktéž je celý jeho kód psaný v jazyce C, byla zde tedy možnost zkusit sestavit AFL++ přímo pro bare metal embedded zařízení. Tato myšlenka, ale není proveditelná kvůli systémovým knihovnám, na které má AFL++ pevnou vazbu.

Také ale disponuje velkou podporou, co se týče emulací a předně podporuje Renode, které je jako jediné z většiny podporovaných simulátoru primárně zaměřeno na emulaci celého systému, a ne jenom určité části jako v případě QEMU. Rozhodující byla ale i výkonost, kterou má AFL++ velmi dobrou oproti jiným fuzz frameworkům.

7 TESTOVÁNÍ NA SIMULÁTORU

Jako první probíhalo zkušební testování s pomocí simulátoru, a to z několika důvodů. Nedošlo k předem definování, na jakém cílovém zařízení se nakonec celé testování uskuteční, počítalo se většími komplikacemi právě kvůli definování vlastního systému. Volba fuzz frameworku byla pevně závislá na podporovaných simulátorech.

Konečný zvolený simulátor byl Renode, primárně kvůli lepší implementaci vlastního systému než jiné systémy. Kromě toho Renode disponovalo několika vlastními definovanými systémy i pro zařízení distribuované společností NXP.

7.1 Nastavení simulátoru

V prvotní fázi došlo k samotnému nastavení simulátoru a zjištění, zda je vůbec vybraný testovaný binární soubor schopný běžet v simulátoru.

Z oficiálního gitHubu Renode projektu byla naklonovaná portable verze Renode. Po přidání všech potřebných cest do systému byl spuštěn příkazem: renode, Renode simulátor. Následně bylo použity další příkazy v tomto pořadí:

1. mach create => vytvoření nového stroje, ve kterém bude probíhat testování
2. machine LoadPlatformDescription @platforms/cpus/imxrt1064_evk.repl => načtení definovaného chipsetu v tomto případě desky MIMX1064-evk
3. LoadElf pathToElf => načtení do paměti testovaný binární soubor
4. Sysbus.ShowAnalyzer lpuart1 => otevření low power uartu zařízení pro možnost zadávaní vstupů k otestování funkčnosti binárního souboru
5. Start => spuštění simulátoru

Po pátém kroku došlo k spuštění simulátoru, ale nedošlo k spuštění binárního souboru z důvodu chybějících adres a problému s předdefinovanými zařízeními.

7.1.1 Troubleshooting

Vzhledem k nespuštění binárního souboru bylo při rozběhnutí Renode v debuggovacím módě zjištěno, že dochází okamžitě k přetečení zásobníků a nemožnosti Renode nalézt některé komponenty na příslušných adresách v simulátoru.

Při detailnějším zkoumání bylo zjištěno, že adresy, na kterých nastává problém, jsou adresy ihned zpočátku main funkce, a to nastavení memory protect unit (MPU) pro toto vývojové zařízení. Vzhledem k tomu, že simulátor používá vlastní definici MPU pro simulací desky

bylo ze zdrojového kódu shell projektu, odstraněno volání funkce pro nastavení a definování této komponenty.

Po znovu sestavení a prolinkování binárního souboru, znovu nahrání pro potřeby simulátoru byl další zkušební běh úspěšný a program se choval v simulátoru standartním způsobem.

7.2 Nastavení pro fuzz testování

Renode má již implementovanou podporu AFL++. Nemusí tedy dojít k úplnému vymýšlení možnosti propojení simulátoru a fuzz frameworku.

Součástí podpory jsou dva python soubory, které obsahují celou logiku komunikace mezi frameworkem a emulátorem. Ke komunikaci dochází pomocí takzvaných hooku, které posílají AFL++ aktuální adresu a stav celého testovaného programu.

K tomu, aby mohl být ukázkový program použit jako cíl testování stačí implementovat vlastní repl a resc soubory. Což jsou speciální soubory pro Renode, které automatizují celý proces nastavování simulátoru a následně přepsat python skript, který definuje práci se vstupními daty.

Vzhledem k tomu, že použitý soubor s koncovkou .repl, který obsahuje popis celého systému, na kterém probíhá testování, je pouze překopírován do složky ze souboru cpus. Tato složka obsahuje některé před definice možných systému, které Renode umí simulovat bez nutnosti vlastní definice. Nedochází v něm k žádné úpravě a není tedy předmětem z žádných následujících kapitol zabývající se přímo úpravou ukázkových předdefinovaných souborů pro potřeby našeho testování.

7.2.1 Imxrt1064_evk.resc a example-uart-imxrt1064.resc

Soubory s koncovkou .repl jsou speciální soubory, které automatizují základní nastavení pro Renode simulátor. Tyto dva soubory vychází z ukázkového skriptu s názvy example-uart.resc a efr32mg_sltb004a.resc, které jsou dostupné v renode módě jako ukázkové příklady použití.

Základní struktura obou souborů byla ponechána a došlo pouze k změnám definující platformu testování, testovaný program a nastavení proměnné globálního kvanta. Taktéž kromě originálního python skriptu, který byl načten ihned ze začátku dochází k načtení úpravné kopie, která upravuje právě logiku práce s daty, viz kapitola 7.2.2 úprava afl_uart.py.

Změny, které se projevily v souboru example-uart-imxrt1064.resc, tento soubor se volá jako první v příkaze AFL++ pro start testování. Byl změněn include, na importování právě druhého resc souboru mimxrt1064.resc.

Poslední změna v tomto souboru je nastavení globálního kvanta použitého při testování, které je 0.0000025. Správné nastavení této proměnné je klíčové, protože při špatném nastavení se může stát, že UART zařízení přestane správně reagovat nebo dojde i k přetečení zásobníků a data získaná během testování přestanou korespondovat s realitou. Konečná použitá hodnota byla zjištěna metodou pokus-omyl, kdy při tomto nastavení je AFL++ schopno dosáhnout nejlepšího výkonu bez toho, aniž by cílové zařízení dostal do stavu bufferingu ihned po začátku testování.

Pro druhý resc soubor došlo k úpravě většiny nastavení, protože tento soubor, imxrt1064_evk.resc, obsahuje skript, který automatizuje základní kroky k nastavení testovacího stroje. Od vytvoření takového stroje, nahrání platformy, binárního souboru až po zobrazení simulovaného UARTU zařízení. V tomto souboru došlo, tedy ke změně nahrání popisu testované platformy na právě imxrt1064_evk.repl, nahrání testovaného elf souboru a definování použitého uartu z originálního usart0 na lpuart s indexem 1.

7.2.2 Úprava afl_uart.py

Kromě repl a resc souboru použitých pro nastavení Renode, byl upraven i python script, který definoval, jakým způsobem se data získaná z AFL++ dostanou do simulátoru a jak je s nimi dále pracováno.

Základní práce s daty je definována v jediné funkci quantum_hook, která jako parametr přebírá informace o vytvořeném stroji. Prvotně byla odstraněna logika definující předchozí testovány program, který měl, pokud na vstup dostal znak ‚a‘ zapříčinit pád aplikace. Zůstal tedy holý základní kód, ve kterém došlo k úpravě použitého vstupu na zařízení z usart0 na lpuart1. Logika, jakou se data posílají na vstup, a to jako jednotlivé charaktery získané z bytového pole zůstává stejná. Celá funkce je pak vidět na obrázku níže.

```
16 def quantum_hook(mach):
17
18     if len(visited) < IDLE_COUNT:
19         n = read(INFD, data, DATA_SIZE)
20         for byte in bytearray(data.raw[:n]):
21             mach["sysbus.lpuart1"].WriteChar(byte)
22
23         if n == 0:
24             if len(afl_renode.visited) == 1:
25                 afl_renode.status = STATUS_ABORT
26             else:
27                 afl_renode.status = STATUS_SUCCESS
28
29 afl_renode.quantum_hook = quantum_hook
```

Obrázek 3. Upravená funkce quantum_hook

7.2.3 Vstupní korpus dat

Nelze začít AFL++ bez předem definování minimálně jednoho vstupu. Tyto vstupní data, ale nesmí zapříčinit pád aplikace a musí skončit úspěšně. Proto byla vytvořena složka s názvem INPUTS, ve které byly vytvořeny tři vstupní textové soubory obsahující následující fráze: led 1 on, led 2 on a help. Každá fráze v jiném textovém souboru.

Tyto tři rozdílné vstupy dostaly program pokaždé do jiné koncové adresy a vyvolaly jinou reakci. Vzhledem k tomu, že ani jedna výše uvedená fráze by neměla zapříčinit pád aplikace dochází při prvotnímu běhu k pokrytí většiny základního chování programu.

7.3 Spuštění AFL++

Abychom si ověřili správnost implementace logiky testování a funkčnost upravených souborů, byl před samotným testováním, AFL++ spuštěn na pár minut, aby došlo k zmapování a validaci funkčnosti. Po ověření funkčnosti byl AFL++ spuštěn na delší časový úsek, do doby, než výkonost poklesla na stabilní hodnotu.

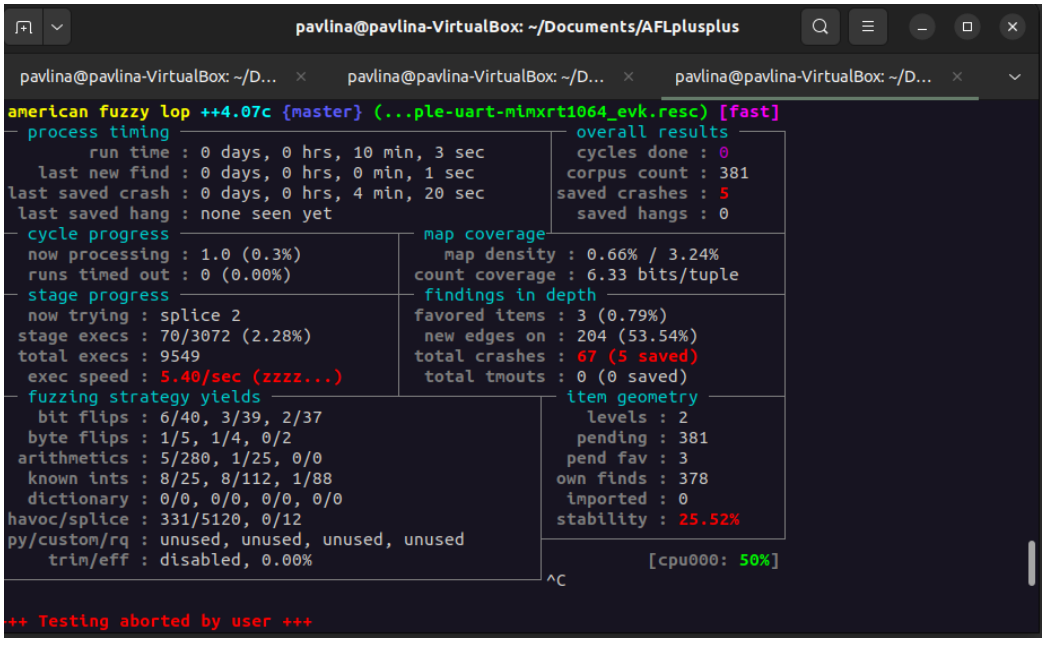
Příkaz, kterým byl spuštěn AFL++, i následně pro již ostré testování je následující: AFL_SKIP_CPUFREQ=1 AFL_SKIP_BIN_CHECK=1 ./afl-fuzz -M testingDir -D -t 15000 -i INPUTS -o OUTPUTS -R -- renode_mode/example-uart-mimxrt1064_evk.resc

Kromě volání resc souboru, ve kterém se nachází i volání repl souboru, zde dochází k definici dvou flagů pro fuzz framework speciálně. Jedná se o AFL_SKIP_CPUFREQ=1 a o AFL_SKIP_BIN_CHECK=1. První definovaný flag říká, aby framework nekontroloval

taktovací frekvenci jádra, se kterou je někdy problém a druhý říká, že není potřeba kontrolovat vstupní data a rovnou začni s testováním. Jedná se o doporučené spuštění AFL++ přímo z oficiálních Renode stránek.

Flag – R nám pak definuje, že chceme AFL++ spustit v Renode módě a jedná se tedy o testování v simulátoru. Velmi užitečný, a i zde použitý je parametr – t, který nám říká že jakmile testovací případ přesáhne tento definovaný časový interval a není do té doby řádně ukončen, ukonči ho a vyhodnoť to jako selhání. Tím implementujeme jednoduché ošetření proti zacyklení během testování.

Při tomto nastavení a úpravě se nakonec při zkušebním běhu dostáváme na počátečně ucházející rychlost 60 exekucí za sekundu. Tato rychlost, ale není konstantní a časem dochází k poklesu. Obrázek číslo 2 pochází již z ostrého testování a je na něm vidět, kromě nalezení 5 vstupů zapříčiňující pád aplikace i nízkou procentuální hodnotu stabilizace. Fuzz framework totiž vstupní data testuje několikrát a pokud se pokaždé dostane k jiné hodnotě, nebo projde při vykonávání programu jinou cestou, vyhodnotí to jako nestandartní chování a stabilita poklesne. Nízká stabilita je, ale s největší pravděpodobností zapříčiněna simulátorem.



Obrázek 4. Běh AFL++ s Renode

7.4 Možnosti zrychlení

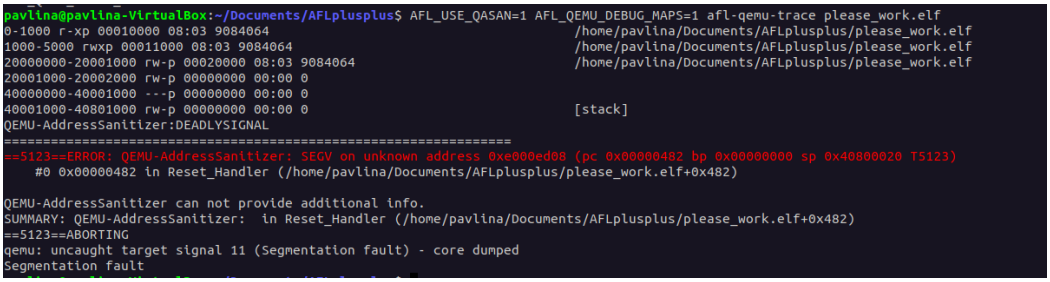
Vzhledem k neoptimálnímu počtu exekucí za sekundu byly zjišťovány možnosti zrychlení. Ideálně by rychlost exekucí měla být přes 100, aby došlo k rychlejšímu nalezení zajímavých

vstupních dat a program mohl být v kratším čase prozkoumán více do hloubky. Pokud je totiž počet exekucí menší než 100, musí se počítat s delším časem testování.

7.4.1 Přechod na QEMU

Ideální rychlost slibuje QEMU, které má u AFL++ největší podporu a nabízí mnoho dalších funkcionalit. Například perzistentní mód. Tento mód umožní zacyklit testování kolem jedné funkce, v našem případě okolo ledControl.

Avšak při pokusu spustit AFL++ v perzistentním módu dochází k podivnému pádu binárního souboru ihned při začátku na adrese Reset_handler. Tato informace byla vytažena pomocí použití funkcionality AFL++ k trasování binárních souborů a nalezení požadovaných adres pro lepší přístup k testování afl-qemu-trace. Na této adrese se definuje funkce, která restartuje celé zařízení, aby bylo zajištěno, že při opětovném spuštění programu není ovlivněn předchozím stavem.



Obrázek 5. Reset_Handler chyba

Prvotní teorie byla, že se nachází problém v linkeru, použitém pro linkování celého projektu. Avšak ani při změně vstupní adresy anebo i při použitím jiného linkeru se tato teorie zdála být zcestná.

Při další a bližší analýze podpory QEMU vyšlo najevo, že QEMU v této fázi je schopno emulovat pouze CPU, ale není schopno bez předchozí definice emulovat celý systém. Navíc primárně se, v případě testování v QEMU módě, jedná o podporu Android zařízeních, ale už ne ostatních zařízení s jiným CPU.

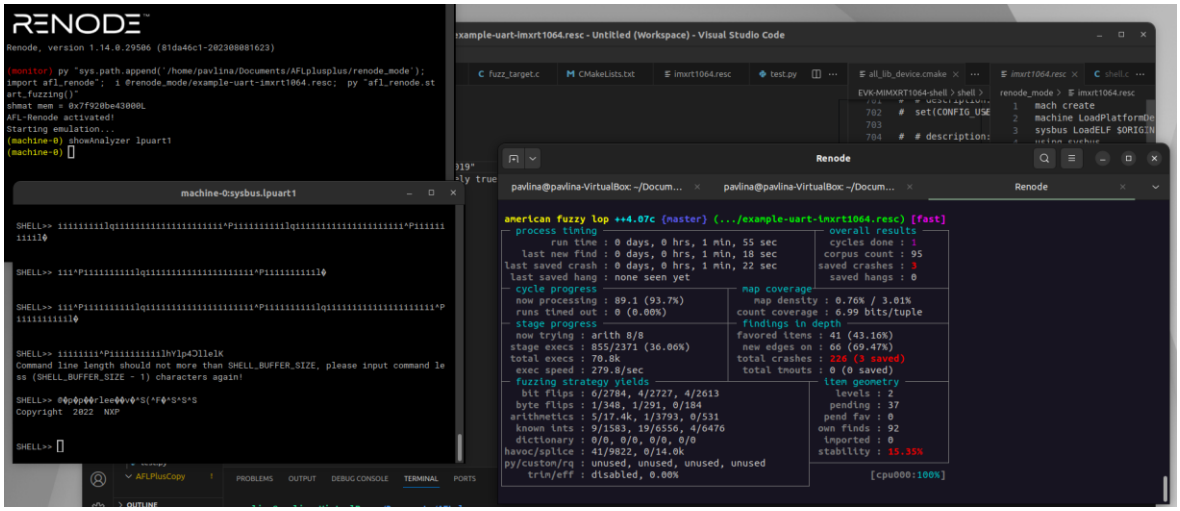
Aby mohlo být QEMU použito pro tento styl testování bylo by potřeba navrhnout vlastní definici celého simulovaného prostředí a toto prostředí pomocí AFL proxy propojit s AFL++. Avšak toto není primárním objektem zkoumání této práce a po konzultaci s konzultantem se opustilo od implementace QEMU podpory pro toto testování.

7.4.2 Instrumentace binárního souboru

Dalším možným řešení bylo zkusit instrumentaci binárního souboru. Neboli přidat funkce do již sestaveného souboru, K tomu existují již nějaké nástroje přímo pro AFL++. Avšak i zde je základní problém, že sice hlásají podporu ARM procesoru, ale procesoru typu Cortex ARM, už ne. Avšak v moment, kdy by se do těchto nástrojů přidala podpora i jiných procesorů, jedná se o nejlepší možné řešení. Tím že bychom byli schopni přidat funkce, které používá AFL++ pro lepší mapování paměti a stavu, bychom mohli dosáhnout lepšího pokrytí, ale hlavně i nalezení lepších testovacích dat a dosáhnutí i lepších výsledků.

7.4.3 Úprava proměnné QuantumTime

Poslední možností bylo tedy najít ideální hodnotu pro QuantumTime. To se sice povedlo, ale pokud se hodnota jednom lehce změní nebo se v určitý čas pošle moc dlouhá vstupní zpráva, dojde k bufferingu. Tento jev, ale neskončí vyhozením chyby, ale překvapivě podezřelým zvýšením výkonosti. Na vstupu se sice žádná data neprojeví, ale vzhledem k tomu, že jsou na vstup poslána a jenom dochází k jejich ukládání k pozdějšímu zpracování a všechno dále v pořádku funguje, AFL++ nepřestává testovat. Tento problém se, ale objevuje buď v pozdější fázi testování nebo při špatném nastavení Quantum time, kdy je hodnota příliš nízká.



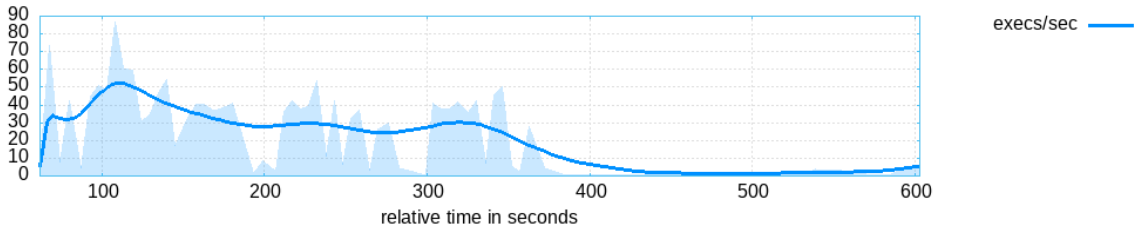
Obrázek 6. Ukázka vyvolaného Bufferingu

Jak můžeme vidět na obrázku výše, hodnota exekucí za sekundu se pohybuje až v absurdní výšce v závislosti, na čem testování provádíme, na lpuartu pak nejsou vidět žádné příchozí vstupní data.

7.5 Výsledky testování

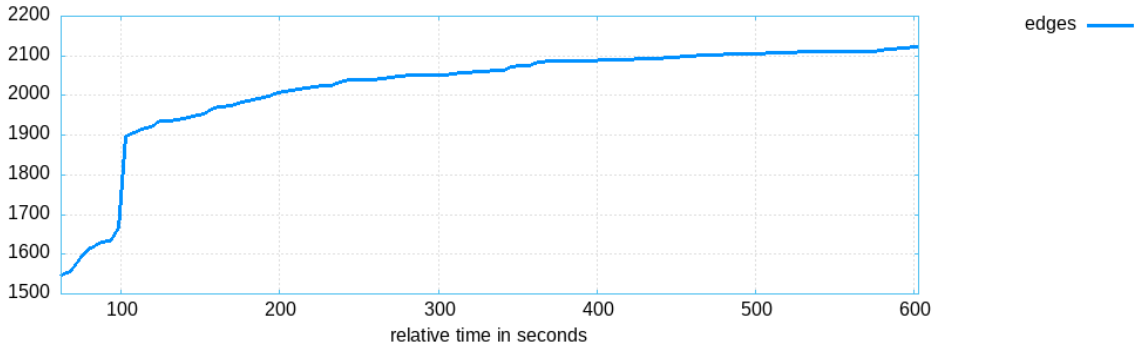
Při správném a hraničním nastavení jsme schopni dostat se na rychlost 60 exekucí za sekundu. To znamená, že za jednu sekundu jsem schopni udělat 60 testovacích případů. Ovšem toto číslo je sice pěkné, ale jakmile AFL++ projde všechny základní větve, rychlost poklesne až ke 3 exekucím za sekundu.

Toto je vidět na obrázku číslo 5. Kdy v čase klesá výkonost. Ze začátku testování se dostáváme do nejvyšší získané hodnoty okolo 60, a následně dochází k pomalému poklesu až téměř k tomu, že počet exekucí za sekundu není ani jedna.



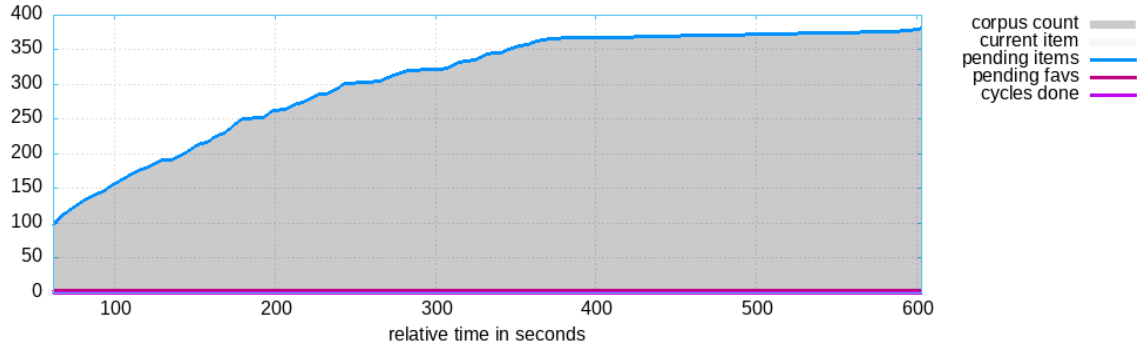
Obrázek 7. Výkonost běhu pro testování v simulátoru

Tento výsledek je, ale ovlivněn hlavně počtem nalezených hraničních hodnot. Toto je vidět na následujícím grafu číslo 6. V moment, co framework nalezne většinu hlavních větví a snaží se dostat k méně zásadním a hůře nalezeným dochází k poklesu a počet nějakých hraničních případů stagnuje.



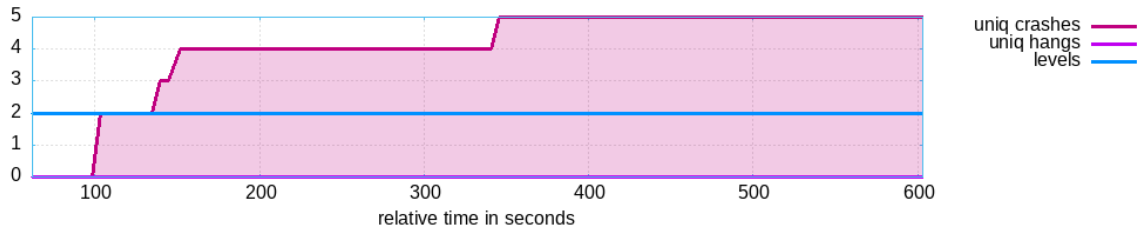
Obrázek 8. Počet nalezených hranic pro testování v simulátoru

Co je ovšem ovlivněno i rychlosti je generace vstupních dat. Kdy ze začátku jsme schopni mutovat rychle a hledat nové cesty, ale vzhledem k tomu že nám ke konci trvá jeden testovací případ skoro i tu sekundu nejsme schopni reagovat a generovat nová zajímavá vstupní data a tím pádem se nám zpomaluje i možnost rozšíření testovací sady.



Obrázek 9. Počet generování vstupních dat pro testování v simulátoru

Avšak testování můžeme označit za úspěšné, protože i když neprobíhalo příliš dlouho tak během prvních pár minut se povedlo najít 5 vstupních dat, které vyvolaly nestandartní chování anebo chybu. Otázkou, ale zůstává, kolik těchto vstupních dat je zapříčiněno chybou ve zdrojovém kódu anebo v simulátoru.



Obrázek 10. Počet nalezených chyb pro testování v simulátoru

Celkově testování pro simulátor nedopadlo až tak zle jak byl předpoklad. Rychlost sice není úplně adekvátní, ale samotná implementace nepředstavuje žádný problém a jedná se o otázku možná tak hodiny. Ovšem jeden z hlavních a nestandartních výsledků je celková stabilita programu, která je nízká. Což značí že se program chová ve velké míře jinak se stejnými testovacími daty. Avšak i to může být zapříčiněno pouze tím, že se testování neprovádí na reálném zařízení.

8 BĚH NA ZAŘÍZENÍ

Jako druhý testovací přístup byl zvolen běh na zařízení. Obzvláště z toho důvodu, že jakmile dojde k vyřešení komunikace mezi zařízením a fuzz frameworkem, implementace eRPC by již nemusela představovat problém. Jediný rozdíl mezi implementacemi by byl totiž jenom v tom, že eRPC volá jenom určitou funkci a nedochází k znovu nahrávání programu na zařízení po každém testovacím případu.

8.1 Nastavení zařízení

Prvně byla otestována základní funkcionalita zařízení bez použití fuzz frameworku a ověření, jestli je vůbec testovaný program a vývojová deska kompatibilní. Byl tedy sestaven program v release nastavení. Vývojová deska byla připojena pomocí USB kabelu k PC. Tento USB port byl propojen s virtuálním strojem, na kterém běželo Ubuntu, které bylo použito pro všechny přístupy k testování. Přes možnost nahrání binárního elf souboru přímo přes USB kabel, umožněno aktuálním firmwarem desky, byl elf soubor přesunut na zařízení.

Následně pomocí puTTy došlo k připojení na sériový port AMC0 s nastavením komunikační rychlosti 115200. Přes tuto komunikaci bylo následně na vstup zařízení poslán příkaz: „led 1 on“ a tím došlo k rozsvícení LED na zařízení.

Základní funkcionalita byla ověřena a následně se přistoupilo k definování a implementaci potřebných funkcí k fuzz testování.

8.1.1 Firmware

Vzhledem k tomu, že v základní podobě je sice dostupný debugger umístěný přímo na vývojové desce, ale vzhledem k rychlosti a dalším možnostem došlo k změně firmwaru desky na podporování SEGGER debuggeru.

Došlo ke stažení nástroje umožňujícího snadné provedení této změny, distribuovaného oficiálně NXP, LPCScript. Tato složka byla otevřena v terminálu, na vývojové desce byl propojen jumper J42 a přes příkazový řádek byl spuštěn skript nacházející se ve složce script: program_JLINK, který celou desku přeprogramoval k používání Jlink SEGGER debbugeru. Bylo odstraněno propojení jumper J42 a vývojová deska byla znovu připojena ke zdroji.

Vzhledem k tomu, že v této změně, již není možno vývojovou desku napájet stejným kabelem jakým je připojena na sériový port, musí dojít ke změně napájecího módu a připojením jiného napájecího zdroje. Pro tyto účely bylo zvoleno napájení přes USB port.

8.2 Soubor COM.c

Pro potřeby komunikace mezi počítačem, na kterém běží AFL++ a vývojovou deskou, na kterou se pokaždé přes ladící zařízení nahraje program byla zvolená komunikace přes sériový port. Vzhledem k nemožnosti automatizace programu PuTTY, byla v jazyce C navrhnuta vlastní definice.

Jazyk C byl zvolen z toho důvodu, že AFL proxy je psán taktéž v tomto jazyce, a tak není problém volat pouze funkce z toho programu než řešit volání podprogramu nebo skriptu. Pro tyto účely byla použita systémová knihovna Termios, která obsahuje všechny důležité funkce pro definování a návrh vlastního komunikačního programu. Proměnná fd, která obsahuje všechny informace a pro které je připojení navrženo, je pak definovaná jako globální proměná, aby se s ní dalo pracovat v rámci každé funkce bez nutnosti přeposílat ji jako parametr.

V dalších kapitolách jsou popsané funkce obsaženy v tomto souboru.

8.2.1 definePortSettings

Struktura termios typu, která definuje základní nastavení sériového portu. Implementované nastavení portu je základní a doporučeno v readme souboru obsaženém v ukázkovém programu pro nastavení puTTy.

Jedná se tedy o nastavení těchto parametrů: Rychlost komunikace, počet datových bitů, parity, stop bit a kontrola propuštění. Hodnoty byly nastaveny fixně a není možné je tedy bez zásahu do zdrojového kódu měnit. K nastavení byly použity funkce právě z knihovny termios pro nastavení sériových portů. Reálné hodnoty nastavení jednotlivých parametrů jsou vypsány v tabulce číslo 1 níže.

Tabulka 1. Hodnoty nastavené pro sériový port

Rychlost komunikace	115200
Data bit	8
Parita	Ne
Stop bit	1
Kontrola propuštění	Ne

8.2.2 openCommunication

Tato funkce otevře a naváže spojení se sériovým portem s předem definovaným nastavení právě ve výše zmíněné funkci. Přebírá dva parametry, a to konstantní char, který definuje název portu kam se má připojit a termios strukturu, která obsahuje právě nastavení portu.

Komunikace se otevírá jen pro zápis, a to z toho důvodu, že nebyla nalezena možnost přesměrování JLink SEGGER debuggeru na sériový port, aby z něj mohlo docházet i ke čtení. Kromě tohoto nastavení se také pokusí otevřít komunikaci v ne blokujícím módě, aby pokud by se nějaký subproces snažil také připojit musel počkat a neshodil celé testování.

Předtím než dojde k načtení nastavení je aktuální obsah fd vyflushován, aby nedošlo ke konfliktu a nahrazen nastavením v předaném parametru.

8.2.3 CloseCommunication

Jednoduchá funkce, která zavolá funkci close na fd a tím se odpojí z komunikace. Funguje jako čisté ukončení testovacího případu a pokud se i v pořádku odpojí můžeme začít pracovat s logem získaný z testování.

8.2.4 writeToPort

Funkce přebírá jako parametry data k zápisu na port a jejich délku. Pokud na konci vstupních dat nenajde znak pro nový řádek, přidá ho tam. Bez toho by nedošlo k odeslání fráze na vstupu a nebyl by poslán ke zpracování na zařízení.

Zápis je prováděn pomocí funkce write, na definovaný fd port. Po dokončení zápisu je port vyčištěn, aby byla zachována konzistentnost. Funkce také vrací počet bytu, které zapsal, pokud tato hodnota není vyšší, než nula nedošlo k zapsání žádných dat a pokus byl neúspěšný.

8.3 Ladění

Zvolení ladícího zařízení je klíčové, protože je naši jedinou možnou komunikací mezi cílem testování a fuzz frameworkem. Primárně je potřeba dostat ze zařízení informace jaké adresy během vykonání testovacího případu navštívil a jestli se během toho nedostal na nestandartní místo v paměti nebo nedošlo k nestandartnímu chování.

8.3.1 Výběr ladícího zařízení

Vzhledem k tomu, že potřebujeme vidět navštívené adresy bylo by optimální použít ladící kabel, který umožnuje real time posílání dat o stavu testování. Pro tyto účely by byl vhodný použít trace32 s debuggerem Lauterbach. Avšak vzhledem k tomu, že vývojová deska v základu není nastavená pro toto použití, přestože disponuje ladícím JTag portem. Ale v základu je deska nastavená pro komunikaci přes SWD interface a ne Jtag. K tomu, aby mohla být deska pro toto nastavení použitá by bylo potřeba odpálit specifické rezistory na desce, a to vzhledem k dalšímu možném používání vývojového kitu nebylo možné, byl použit on board debugger Segger.

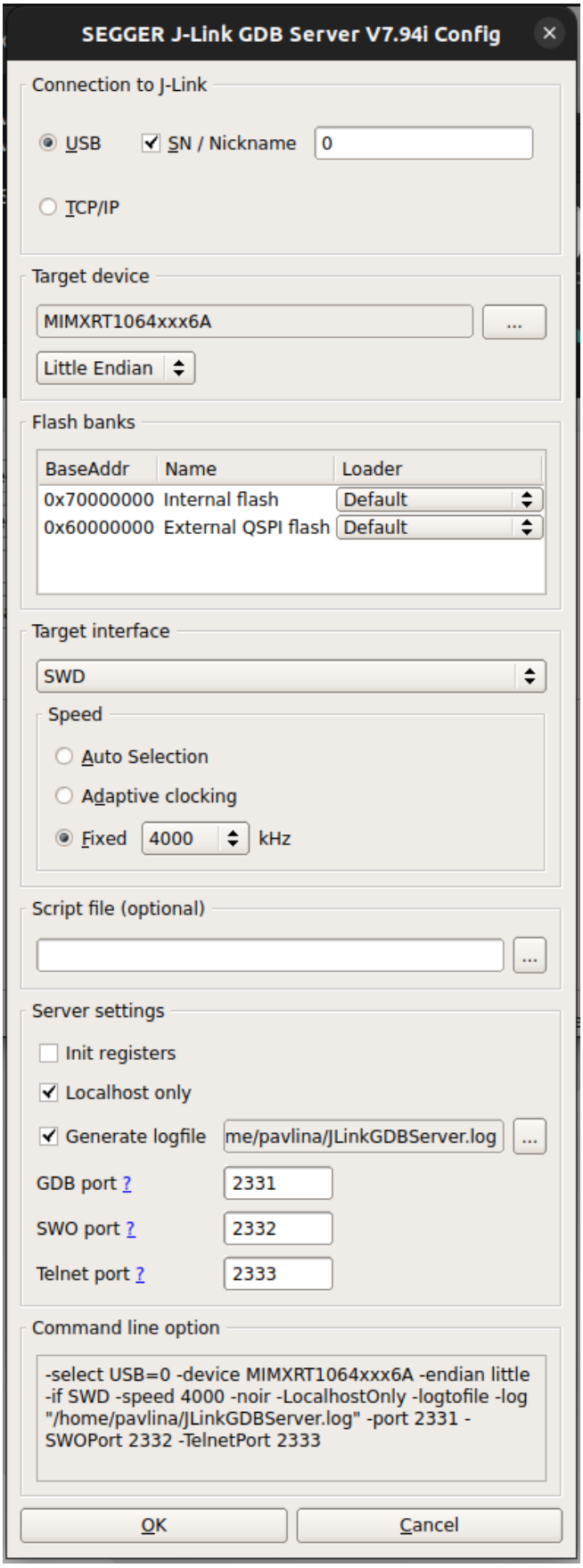
8.3.2 JLinkGDBServer

K navázání komunikace s vývojovou deskou pro potřeby debuggování byl zvolen SEGGER JLink GDB Server, který se manuálně spouští před začátkem testování. Připojení k vývojové desce je přes USB port, s fixní rychlostí SWD interface. Porty jsou taktéž nastaveny fixně a pokud by během testování došlo k pádu serveru může být opět spuštěn se stejným nastavením a není potřeba ho dále pře nastavovat.

GDB Server umožnuje také generovat logovací soubor obsahující všechny potřebné informace během běhu programu. Tento logovací soubor je, ale pro to abychom mohli získat informace o navštívených adresách nepoužitelný. A to primárně z důvodu že během průběhu programu nedochází po jeho úspěšném vykonání a čekání na další příkaz, k jeho pročištění. Tedy se v něm zapsaná data hromadí a vzniká tak obří soubor, ze kterého, pokud bychom chtěli automaticky něco číst, budeme tyto informace získávat těžko.

Pokusy o automatické vymazání tohoto souboru po přečtení dat získaných z proběhlého testovacího případu nebyli úspěšné. Primární důvod neúspěchu je ten že pokud je GDB Server stále aktivní, má svůj Logovací soubor neustále otevřený. A tak jakýkoliv zápis končí jenom poškozením tohoto souboru a nutností restartovat GDB Server, kterého načtení také trvá nějakou dobu a pokud bychom k tomu přičetli nutnost čekat na správné nahrání binárního souboru přes arm-gdb, zvyšuje se nám doba, kdy při testování jenom na něco čekáme a nic neprovádíme a efektivita tudíž klesá.

Počáteční nastavení včetně hodnot, je vidět na obrázku číslo 9. Není nutno nastavovat parametr USB, pokud se jedná o jediné připojené zařízení gdb Server je schopný si je nastavit sám.



Obrázek 11. Použité nastavení pro GDB server

Pomocí tohoto serveru se pak nahraje na vývojovou desku požadovaný testovaný program. Proces nahrání je automatizovaný skriptem, popsaným níže.

8.3.2.1 Start_debug_sesssion.sh

Skript použitý k automatizaci nahrání binárního souboru. Tento skript kromě nahrání, definuje i vytvoření logovacího souboru, který byl následně použit pro získávání dat o testování, ale také i nastavení breakpointu použitých pro debuggování.

Primární myšlenkou použití AFL++ pro testování na zařízení je ta, že použijeme způsob pomocí breakpointu. Tyto breakpointy jsou nastaveny na dvě vstupní funkce a pět výstupních. Předpoklad je takový, že pokud se při konci testovacího případu neocitneme ani na jednom z pěti výstupních předem definovaných breakpointu, program se chová nestandartně.

Kromě toho je zde také definován pod skript gdbLedControl, ve kterém je pouze automatizováno, že pokud se dostane GDB na některé ze vstupních breakpointu má pokračovat, pokud se jedná o breakpointy výstupní má ukončit ladící proces.

8.3.3 DebugConsole.c

Soubor DebugConsole.c je soubor, který automatizuje čtení logovacího souboru získaného z logu arm-gdb. Jeho primární práce je parsovat informace v něm obsažené podle klíčového slova breakpoint. Pokud na takové slovo narazí uloží jeho adresu do pole breakpointu.

Takto prvně uloží prvních 8 nalezených adres, protože prvotně se v logovacím souboru definují všechny nastavené breakpointy.

Tyto nalezené adresy pak dále použije k tomu, aby je nemusel znova parsovat z logovacího souboru a v závislosti na jaké pozici se nacházejí je při nalezení čísla breakpoitnu uloží do adresové struktury. Takto projde celý soubor a všechny breakpointy kterými program prošel uloží do struktury kterou přebírá jako ukazatel na již definovanou strukturu v proxy AFL.

8.4 AFL ++ Proxy

Implementace AFL proxy vychází z dostupné šablony. K tomu, aby mohl být tento vzor použit pro naše účely je potřeba změnit pouze jednu funkci, a to hlavní smyčku while, ve které se definuje, jakým způsobem jsou data posílaná na cílové zařízení a jak se zvyšuje pokrytí.

Předtím než došlo k implementaci změn v hlavní while smyčce, byla do kódu přidaná funkce na hashování hash32, která je součástí knihovny používané v AFL++. Tato hashovací funkce se používá pro generování hash dat z navštívených adres, aby reprezentace byla vždy stejná a nedošlo k pomýlení při zvyšování hodnoty pokrytí.

Předtím než program vstoupí do while smyčky, je mu řečena cesta k logovacímu souboru, který je neměnný po celou dobu testování.

V nekonečné smyčce následně dochází k spuštění skriptu ke startu ladícího procesu. Zde se následně program na krátkou dobu uspí, protože pokud bychom začali rovnou před nahráním posílat data, získané výsledky nebudou přesné.

Jakmile se úspěšně připojí na server, dochází k volání funkcím ze souboru COM.c a připojením k seriovému portu. Pokud vše proběhne v pořádku jsou s pomocí funkce writeToPort poslány data získaná z AFL++ na vstup zařízení. I po tomto kroku dochází k uspání procesu, tentokrát na kratší dobu, jak z důvodu vyčkání na správné vykonání programu, ale hlavně aby došlo k zapsání všech logovacích dat do souboru.

V posledním kroku jsou získány adresy z navštívených breakpointu a vyřešen výpočet pokrytí. Ten se počítá ve for cyklu, pro každou adresu, kterou navštívil. Pokud totiž během procesu například navštívil adresy v jiném pořadí jedná se o jinou cestu a je potřeba zvýšit pokrytí na jiných adresách. K tomu, aby byla zachována konzistentnost pokrytí je použita již výše zmíněná hashovací funkce.

Pokud vše proběhlo v pořádku AFL++ ukončí po dokončení while smyčky testovací případ.

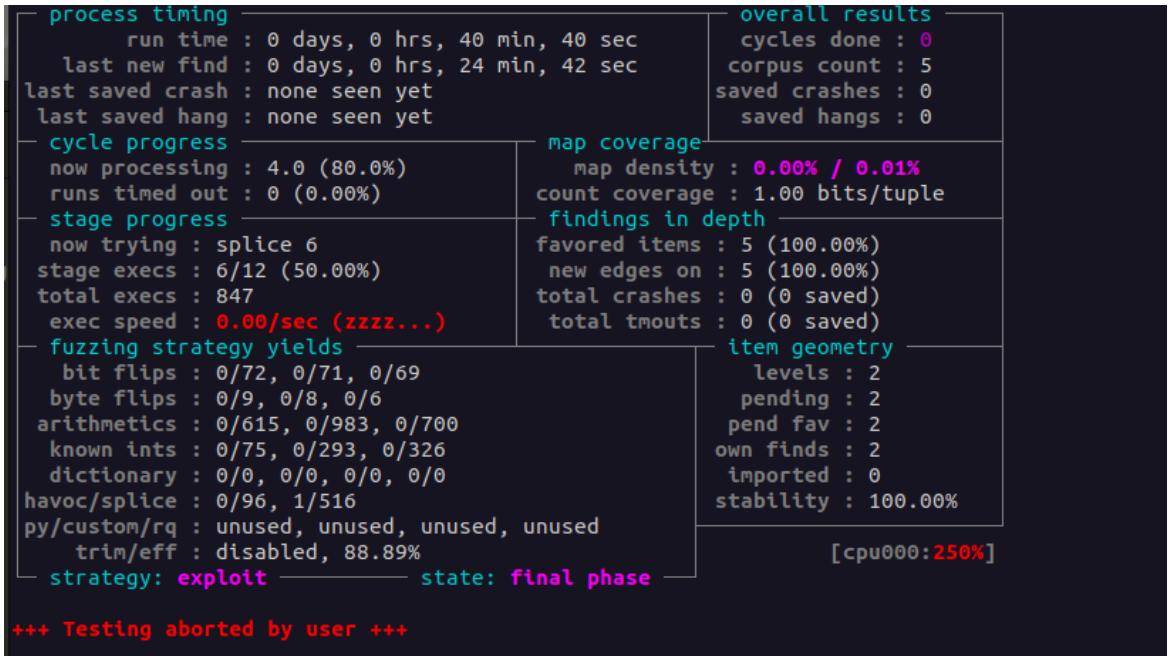
8.5 Spuštění AFL++

Předtím, než je vůbec možné spustit AFL++ s cílem testování afl-proxy je nutné překopírovat podpůrné skripty do hlavní složky AFL++. Přesněji řečeno skript start_debug_session. Pro potřeby testování byl program sestaven s předdefinovanou konfigurací flexi nor debug.

K sestavení afl-proxy byl použit makefile, který definoval všechny potřebné knihovny a include, potřebné pro správné sestavení proxy serveru. Jako poslední krok před začátkem testování je nutné spustit GDB server z příkazové řádky a zkontrolovat že vývojová deska je dobře připojena k počítači a funkční.

Jakmile byli všechny náležitosti vyřešeny může být AFL++ spuštěno následujícím příkazem. AFL_SKIP_CPUFREQ=1 AFL_SKIP_BIN_CHECK=1 afl-fuzz -M onBoard -i INPUTS/ -o OUTPUTS/ -t 150000 -m 200 -D -- ../../ProxyAFL/afl-proxy

Příkaz se liší od příkazu použitého k nastartování testování pro Simulátor minimálně. Kromě, překvapivě, změněného cíle testování z resc souboru na cestu k afl proxy. Je zde i omezení paměti, které se u simulátoru nemuselo použít. Jedná se o maximální velikost dat, se kterými jsme schopni pracovat. Toto číslo je u běhu na zařízení klíčové a dost citlivé. Protože při jakékoliv změně nedojde ani ke spuštění a proběhnutí testovacích případů dostupných v počátečním vstupní složce. Vstupní soubory jsou neměnné od předchozího testování.



Obrázek 12. Běh AFL++ na reálném zařízení

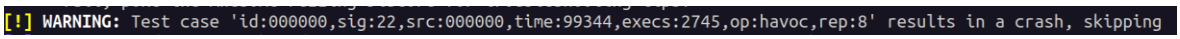
8.6 Použití chybových dat ze simulátoru

Vzhledem k tomu, že během běhu AFL++ na reálném zařízení nedošlo k nalezení ani jediné chyby nalezené při běhu s použitím simulátoru, byli tyto soubory nacházející se ve složce crash, přesunuty do nově vytvořené složky: Crashed_Inputs, jakožto vstupní data pro novou instanci běhu AFL++.

Tato instance AFL++ byla spuštěna se stejnými parametry jako předchozí až na dvě změny. Výstupní soubor, do kterého se ukládají data získaná během testování se z -M onBoard na -

M CrashTest a vstupní soubor definovaný parameterem -i byl ze složky INPUTS změněn na Crashed_Inputs.

Tyto vstupní data zapříčinily sice vyvolání chyby v AFL++, ale to jenom a pouze z toho důvodu že během jejich vykonávání neprošli standartní cestou, která je definována break-Pointy. Protože pokud se vstupní data z těchto souboru pošlou na vývojovou desku přes standartní rozhraní, bez běhu fuzz frameworku, program se chová standartním způsobem.



Obrázek 13. Upozornění na pád testovacího případu

8.7 Analýza nalezeného vstupu č. 5

Vzhledem k navržení logiky Proxy serveru, by mělo dojít k nalezení čtyř základních vstupů. Tři základní se nachází již ve vstupním souboru, čtvrtý téměř okamžitě nalezený vstup je neplatný příkaz, pátý vstup by měl být příkaz exit. Avšak k nalezení pátého vstupu došlo v krátkém intervalu. Vzhledem k běhu aplikace byla minimální šance, že by se jednalo právě o příkaz „exit“.

Po prozkoumání souboru nacházejícího se ve složce qeue, která ukládá všechny soubory přivedené na vstup aplikace, bylo zjištěno, že příkaz v tomto souboru obsažený není žádný z povolených příkazů. V tomto souboru se nacházel text sestaven z náhodných znaků a symbolů.

Pokud dojde k nahrání testovaného programu na desku a poslání tohoto příkazu přes sériový port, aplikace se zachová nestandartně. Ve standartním běhu aplikace by mělo dojít k vypsání informace uživateli, že tento příkaz nezná a pokud potřebuje pomoc ať zavolá příkaz help. Avšak tento příkaz vyvolá právě onu nápovědu a pak již napíše výše vypsanou hlášku. Dojde tedy k tomu, že program během svého vykonávání projde breakpointy jak pro help tak i pro vypsání neexistujícího příkazu.

To vše je zapříčiněno tabulátorem, který se nachází na začátku poslaného textu. Tabulátor je podle všeho alias na nápovědu, zároveň je možné rovnou poslat i příkaz, bez nutnosti opustit tuto nápovědu. Ale ve zdrojovém kódu se nepovedlo dohledat, kde se tento alias definuje a ani nikde v projektu není specifikováno, že by se jednalo o standartní a očekávané chování programu.

Obrázek 14. Analýza nestandartního chování při poslání nalezeného příkazu

8.8 Možnosti zrychlení

Výsledná rychlost je 0.36 exekucí za sekundu. Toto číslo je zapříčiněno nutností volat funkce sleep a pozastavovat exekuci programu, aby mohl být binární soubor správně nahrát a také aby na vstup byly data řádně předány. Tyto pozastavovací funkce nelze z kódu odstranit ani je snížit. Již v tomto momentě jsou na pomyslném minimum, kdy každá změna vyvolá problémy, které ovlivní výsledek testování (data jsou posílaná před kompletním nahráním programu, čteno je ze staré verze logovacího souboru, data jsou neposlána všechna...).

Možné řešení je tedy implementovat celou logiku s možností získávat informace o testování v reálném čase. Tedy pokud bychom mohli získat přesnou adresu už v momentě kdy se na ní nacházíme a pouze si její část uložit. Tehdy by nebylo nutné ukládat a parsovat logovací soubor. Taktéž bychom zvýšili hodnotu pokrytí, protože v tomto momentě jsme schopni zjišťovat pokrytí jenom na základních bodech, vstup a výstup, programu.

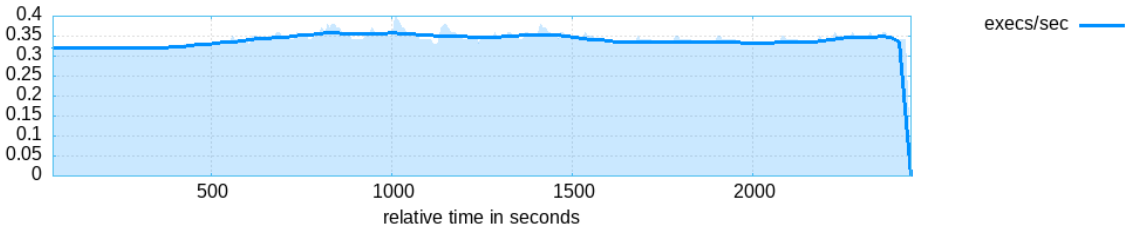
Nahrávání programu do paměti vývojové desky probíhá momentálně s každým začátkem testovacího případu. Pokud bychom, ale automatizovali lépe získávání dat z vývojové desky a měli lepší přístup k ladícím nástrojům mohli bychom nahrávání omezit pouze pokud testovací případ předtím skončil chybou nebo nestandartním chováním.

8.9 Výsledky testování

Výsledky testování běhu přímo na zařízení jsou dosti překvapivé. Primárně tím, že z výkonosti dosažené na simulátoru se dostáváme k rychlosti asi čtvrt testovacího případu za sekundu. To je zapříčiněno z hlavně tím, že během vykonání testovacího případu dochází

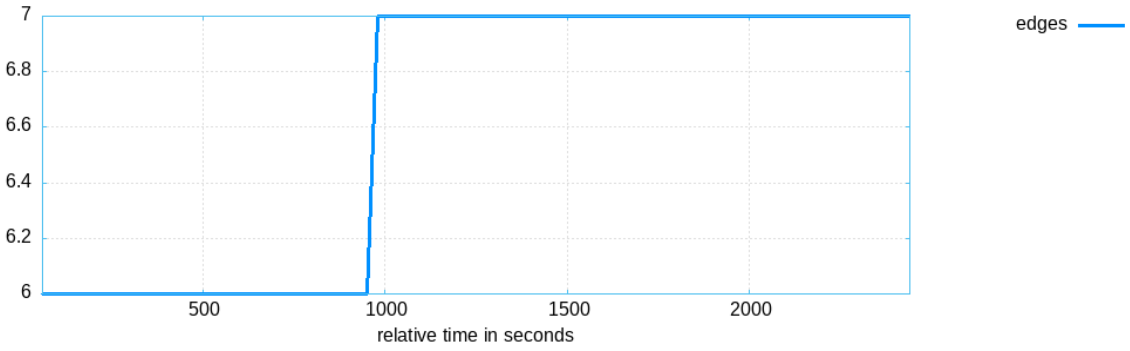
k čekání v celkové délce 3 sekundy. Zde je, ale problém že pokud dojde jenom k malému snížení této délky výsledky jsou nekonzistentní, dochází ke špatnému nahrání binárního souboru anebo čtení z neúplného logovacího souboru. Ovšem tato rychlost je během testování konstantní, jak je vidět na grafu níže. Taktéž je zde vidět chyba, která se stane v momentě je-li testování ukončeno uprostřed vykonávání testovacího případu, stále běží podprogram arm-none-eabi-gdb.

V tomto případě nedojde k validnímu ukončení arm-gdb podprogramu, který zajišťuje komunikaci se serverem a je nutno ho zabít.



Obrázek 15. Výkonost pro běh přímo na reálném zařízení

Počet nalezených hranic je konstantní, a to vzhledem k tomu, že máme pevný počet adres, o kterých můžeme vědět, jestli jsme je během testování narazily. V půlce testování dochází k nalezení pátého korpusu použitého ke generování nových vstupů. Tento korpus zvyšuje počet nalezených hran z hodnoty šest na sedm. Poté nedochází k nalezení žádných dalších zajímavých vstupních souborů, které by rozšířili pokrytí.

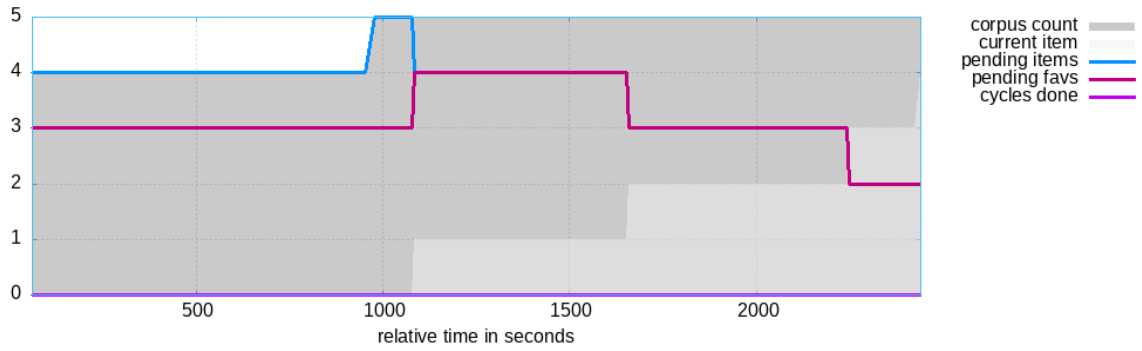


Obrázek 16. Počet nalezených hranic pro testování na reálném zařízení

Vzhledem k počtu exekucí za sekundu je jasné, že nějaká generace a progres během testování není úplně ideální. Obzvláště během toho, kdy nejsem schopni fuzz frameworku říct, jestli je testovací případ, který prošel stejnými vstupními i výstupními adresami lepší než ten předchozí. Během celého běhu testování byl tedy nalezeny dva nové vstupní korpusy. A to

první ihned ze začátku, toto nalezení nelze vidět na grafu níže, protože k němu dochází v prvních několika minutách po začátku testování.

Druhý nový vstup byl nalezen v půlce testování a přidán do fronty. Krátce po jeho nalezení, ale dochází k protestování všech podob jednoho korpusu ve frontě, a tak snížení modré části grafu, která značí počet položek ve fortně. To je vidět na grafu číslo 17. Také si zde můžeme povšimnout vzrůstu oblíbených vstupů ve frontě, také pak i pokles v rámci pokračujícího testování, kdy se dostaneme na hodnotu 2 tj. Zbývá nám otestovat poslední dva vstupy, abychom dokončili cyklus. Oblíbené vstupy jsou takové vstupy, které odhalují nové větve programu a slouží k dalšímu generování vstupních dat.



Obrázek 17. Počet generování vstupních dat pro testování na reálném zařízení

I když běželo testování na vývojové desce déle než samotné testování v simulátoru, ani s delším časem se nepovedlo nalézt jedinou chybu nebo dostat zařízení do nedefinovaného stavu.

Celkově je implementace běhu pro reálné zařízení po vytvoření podpůrných nástrojů a přístupu o něco složitější jak u simulátoru. Je zde potřeba automatizovat více věcí, které AFL++ není schopný sám udělat. Avšak poté dostáváme mocný nástroj, který je schopný neustále posílat data na vstup. Co se týče rychlosti jedná se o velký propad, který je ale způsoben hlavně v úmyslném implementovaném čekání do zdrojového kódu. Samotné poslání dat a provedení programu na vývojové desce není vůbec pomalé a reakce je po poslání i delší zprávy téměř okamžitá. Rychlost zpracování požadavku je zde menší než jedna sekunda.

Pokud by se povedlo vyřešit nutnost uspávat zařízení, je toto řešení i v praxi více použitelné než v tomto stavu, kdy jeho primární použití by bylo během nočních kampaní, kdy si můžeme dovolit blokovat vývojovou desku delší dobu.

9 ERPC

Embedded Remote Procedure call je, jak již bylo řečeno open source rozhraní pro vytvoření klientské a serverové části pro potřeby testování. V našem případě je serverovou částí vývojová deska s nahraným binárním souborem programu. Klientem, který posílá na server požadavky přes sériový port, je počítač s běžící instancí AFL++.

K běhu se použije již vytvořený AFL Proxy, ve kterém se upraví hlavní While smyčka, aby došlo k volání čistě klientského programu, bez neustálého flashování binárního souboru.

9.1 Úprava zdrojových souborů pro potřeby eRPC

Aby mohlo být eRPC použito v Shell projektu je potřeba naimportovat určité soubory a vazby, které eRPC potřebuje.

Jedná se primárně o erpc_c knihovnu, která obsahuje všechny hlavičkové a zdrojové soubory pro úspěšnou implementaci eRPC do projektu. Tato složka byla vložena do __repo__ adresáře, ve kterém se nachází všechny knihovny, které je možné použít při vývoji programu. Přesná cesta, do které byla erpc_c knihovna vložena je: __repo__/middleware/multicore/erpc. V této složce se kromě knihovních funkcí nachází i další zdrojové soubory, a i před sestavený erpcgen program, který slouží ke generování zdrojových souborů z IDL souboru.

Ve složce multicore byla vytvořena i složka example do které byl vytvořen IDL soubor IDL.erpc, který obsahuje informace a definice potřebné pro vygenerování podpůrných souborů pro klientskou i serverovou část aplikace. Toto generování se spustí pomocí příkazu: erpcgen IDL.erpc a erpcgen -g py IDL.erpc.

9.2 IDL soubor

IDL soubor, jak již je psáno výše obsahuje popis hlavní funkce, kterou chceme testovat. Z toho souboru se následně vygenerují soubory definující server a klienta, s kterými budeme dále pracovat.

V IDL souboru byl definován pomocí program erpc_led_control, fixní název pro pojmenování výstupních souborů. Příkaz program umožní konzistentní pojmenování výstupních souboru ve formátu *erpc_led_control*.

Během definování IDL souboru došlo k několika problémům, a to hlavně v tom, že LedControl, který má být cílem testování, protože to je funkce, která nás nejvíce zajímá, pracuje s daty získanými jako parametry z příkazové řádky s tím, že poslední parametr obsahující text získaný z příkazové řádky je dvojitý ukazatel. IDL je schopno definovat pouze string, který pro potřeby jazyka C automaticky převádí na char*. Ale není schopno vytvořit argument funkce, který by měl být zároveň i ukazatel, pokud se nejedná o výstupní argument. Kromě toho je taktéž jeden z argumentů vlastní datový typ, který je definovaný jako prázdná void hodnota. Tu ale také nejsem schopni nadefinovat.

K tomu, aby bylo testování úspěšné musí dojít k úpravě jak ledControl funkce, v rámci definování pro server, ale také v rámci použití v IDL.

9.2.1 Problémy s proměnou shell_handle_t

Jsme zde dost limitováni použitelnými datovými typy, a i když můžeme definovat vlastní nejsme schopni definovat je jako prázdné ukazatele na void, jako v případě hlavičkového souboru, ve kterém jsou definovány originálně. V hlavičkovém souboru fsl_shell.h je totiž tato proměnná definována takto: typedef void *shell_handle_t.

Je potřeba tento datový typ tedy nahradit nebo jako poslední možné řešení, odstranit ho z funkce úplně.

Proměnou shell_handle_t se ani po několika různých přístupech k problému nepovedlo nahradit. Jedno z možných řešeních bylo definovat ji jako prázdnou strukturu, ale to jazyk IDL neumožňuje, umožňuje sice definovat prázdný enum, ale pouze pokud je definovaný externě.

Toto řešení by bylo sice použitelné, ale vzhledem k tomu že by extérní definice byla jiná než v IDL tak značně nepraktické. I když je zde možné definovat externě typedefové aliasy, není možné je definovat jako datový typ void.

Použitým řešením bylo odstranění shell_handle_t z argumentů funkce úplně. Funkčnost nebyla ovlivněna a program po nahrání do paměti a zkušebních příkazech poslaných přes sériový port z programu PuTTY nevykazoval nestandartní chování.

9.2.2 Ostatní argumenty argc a argv

Zbylé dva argumenty v originální funkci LedControl jsou argumenty získané z příkazové řádky programu, v tomhle případě ze shellu. To při volání funkce přímo z klienta není možné. Je proto potřeba najít adekvátní náhradu.

Argc s datovým typem init32_t ukládá informaci o počtu argumentů na vstupu. Typ není potřeba měnit, protože IDL má svůj typ init32, který se během generace přepíše v C generovaném kódu na init32_t.

U druhého argumentu nastal problém, protože originálně jeho definice je: char** argv. Tento datový typ nelze v IDL definovat. IDL sice automaticky typ string převádí na char*, ale nelze zde, pokud se jedná o vstup, definovat druhý pointer. Z toho důvodu byla tato vstupní hodnota definovaná jako textové pole o velikosti tři. Protože předpokládáme při volání funkce formát: led, index led diody, požadovaný stav (on/off).

9.2.3 Definování návratového typu

Vzhledem k tomu, že návratový typ z funkce LedControl je aktuální stav shellu. Jestliže funkce proběhne v pořádku, je vrácen status kStatus_SHELL_Success. Tento datový typ je v originálním zdrojovém kódu definován jako datová struktura se všemi možnými návratovými typy. Aby mohl být tento návratový typ použít i v případě IDL popisu je potřeba jej v souboru definovat.

Jak již bylo řečeno výše, IDL umožňuje definování externích proměnných. Tedy proměnných, které jsou již definované v jiném souboru a nedojde tak k jejich redefinici v souborech vygenerovaných eRPC generátorem. V tomto případě se definice datového typu shell_status_t nachází v hlavičkovém souboru fsl_shell.h, který byl s pomocí příkazu include definován i v IDL souboru.

Kromě toho je také potřeba, pro správnou generaci, definovat externí proměnou znova i pro IDL. To hlavně z důvodu, že i když je proměnná extérní při generaci stále dochází ke kontrole, zda se nachází v IDL souboru, ale již nedochází k vytvoření definice ve vygenerovaných souborech.

Pro toto bylo vyžito enum, pojmenovaného stejně jako chtěný datový typ, ve kterém byly obsaženy všechny elementy extérního enum. S tím rozdílem, že narozdíl od extérního enum, který má ještě vazby na další proměnné obsažené v hlavičkovém souboru, mají všechny elementy svou fixní definici hodnot.

```
@external enum shell_status_t{
    kStatus_SHELL_Success              = 0*100L + 0,
    kStatus_SHELL_Error                = 140*100L + 1,
    kStatus_SHELL_OpenWriteHandleFailed = 140*100L + 1,
    kStatus_SHELL_OpenReadHandleFailed  = 140*100L + 1,
    kStatus_SHELL_RetUsage             = 140*100L + 1,
};
```

```
typedef enum _shell_status
{
    kStatus_SHELL_Success              = kStatus_Success,
    kStatus_SHELL_Error                = MAKE_STATUS(kStatusGroup_SHELL, 1),
    kStatus_SHELL_OpenWriteHandleFailed = MAKE_STATUS(kStatusGroup_SHELL, 2),
    kStatus_SHELL_OpenReadHandleFailed  = MAKE_STATUS(kStatusGroup_SHELL, 3),
    kStatus_SHELL_RetUsage             = MAKE_STATUS(kStatusGroup_SHELL, 4),
} shell_status_t;
```

Obrázek 18. Definice datového typu shell_status_t, vlevo IDL, vpravo fsl_shell

Jak je vidět na obrázku výše, vzhledem k tomu, že stejně nedojde k použití definice z IDL je zbytečné se zaobírat správnými hodnotami použitými pro chybové stavy. Použitá bude i pro vygenerované soubory originální definice vpravo.

9.3 Server

Server je v tomto případě vývojová deska. Proto při kompilaci v souboru CMakeLists.txt nahradíme soubor shell.c serverovou implementací, abychom byli schopni během běhu aplikace dostávat stav, ve kterém se Server nachází a my tak měli přehled, jestli je vše v pořádku. Kromě vlastní úpravy main funkce, kde se za inicializacemi všech požadovaných vlastností a funkcí desky, inicializuje vlastní server, který pak běží v nekonečné smyčce, dokud se nedostane do chybového stavu. Je potřeba přidat definici a inicializaci nově úpravné funkce LedControl.

9.3.1 LedControl

Úpravná funkce ledControl neobsahuje argument shell_handle_t a jinou definici argumentu argv. Avšak logika funkce zůstává nezměněna, protože data pořád přebírá stejným způsobem, jako více rozměrové textové pole, s tím rozdílem že v aktuální podobě je rozměr pole fixní. Funkci tedy není kromě její definice nějak dál potřeba upravovat.

9.3.2 Main

Prvně dochází k inicializaci hlavních funkcionalit vývojové desky a povolení clock brány pro použité GPIO1, které se používá pro komunikaci s LED diodou. Tato část zůstává od originálního main nezměněna. Změna nastává před spuštěním nekonečné smyčky. Zde dochází k definování přenosu, v tomto případě UART, a zásobníku pro zprávy. Také se spouští instance serveru pomocí funkce erpc_server_init. Tato funkce definuje, jakým způsobem bude server komunikovat s klientem a velikost zásobníků zpráv, kromě serveru se vytvoří i instance servise, tedy funkce, kterou chceme volat a testovat. Tato servise se pak přidá k serveru. Následně se v nekonečné while smyčce spustí server a vyřeší se logika práce s případnými chybovými stavy na jeho straně.

UART použitý pro komunikaci je definován jako ERPC_UART s hodnotou Driver_USART1. Definice hodnoty se nachází ve zdrojovém souboru fsl_lpuart_cmsis, který je předem definován výrobcem pro použitou desku.

Pokud během běhu aplikace se stane, že server nevrátí status kErpcStatus_Success, dochází k chybě a je potřeba ji nějakým způsobem zpracovat. Dojde ke správnému odstranění servisy ze serveru, zastavení běhu serveru a vypsání chycené chyby, následně dojde k ukončení nekonečné smyčky pomocí break.

Všechny funkce použité k inicializaci a zprovoznění serveru jsou z eRPC knihovny a funkcí vygenerovaných v souborech s jménem končícím slovem server z IDL.erpc definice.

9.4 Klient

Klientská část aplikace je upravený python skript, který se nachází v ukázkových souborech pro použití eRPC pro jiné typy desky. Tento skript byl umístěn do stejné složky jako IDL.erpc soubor.

Python skript byl upraven tak, že byla odstraněna celá logika serverové části, která v ní byla definována a kterou nepotřebujeme pro potřeby testování. Následně v main funkci, byl odstraněn výběr, pomocí argumentů, mezi klientem a serverem na pouze klienta. Došlo tedy k odstranění dvou řádků starajících se o argumenty z konzole, které přebíraly informaci, jestli uživatel chce na zařízení spustit server nebo klienta. Taktéž došlo k odstranění následující podmínky, ve které se rozhodovalo, zda se zavolá funkce pro server nebo klienta. Tak aby se volal pouze klient. Kromě těchto úprav v mainu byl taky definováno základní nastavení hodnoty portu, aby se při každém spuštěné skriptu nemusel definovat ručně.

Ve funkci runClient došlo k předefinování volaných funkcí pro proměnou client, protože dochází k volání jiné funkce než v případě ukázkového projektu, ve kterém dochází k volání funkce násobení matic. Ve smyčce pak dochází k posílání požadavků na server. Vzhledem k logice testovaného souboru dochází k posílání dvou argumentů. A to velikosti pole a samotného pole s příkazovým textem. Na konci funkce se čeká na stisknutí klávesy od uživatele a ukončení běhu klienta.

9.5 Problémy s implementací

Vzhledem k tomu, že došlo k vytvoření nových zdrojových souborů, které nejsou popsány v předdefinovaných cmake souborech je potřeba je doplnit. Jednalo se převážně o soubory,

které vznikli po generování IDL.erpc. Do složky multicore byly přidány dva cmake soubory definující složky a zdrojové soubory pro kompilaci serverových funkcí a klientských. Kromě těchto cmake byly udělány změny i v hlavním config.cmake souboru, all_lib_device.cmake a v CMakeLists.txt.

Do config.cmake byli přidány chybějící konfigurace pro použité komponenty. Převážně se jednalo o erpc komponenty, které nebyli použité v původním shell příkladu a následně i pár komponent, které definovali funkcionalitu použité komunikace přes UART.

Do all_lib_device byly přidány dva include_if_use pro oba nové erpc shell cmake soubory, aby pokud tyto cmake byli nakonfigurovány v config.cmake souboru došlo k jejich správnému vložení a použití.

Jako poslední byl upraven CMakeLists.txt, ve kterém došlo ke změně v add_executable. Zde byl přidán konfigurační hlavičkový soubor pro erpc a vzhledem k tomu, že došlo k přejmenování hlavního zdrojového souboru, ve kterém se vyskytuje main celé funkce, ze shell na main_server. Došlo k přepsání této hodnoty i v tomto souboru.

9.6 Zkušební běh eRPC

Pokud došlo k nahrání programu přes SEGGER debugger na zařízení, program se jevil jako funkční, avšak při pokusu poslat požadavek na server, přes definovaný python skript došlo k chybě v CRC hlavičce a nestejných bytech. Toto se jevila zpočátku jako chyba na straně klienta, protože program se zdál být funkční, ale vzhledem k tomu že v interní paměti se nacházela jeho předchozí implementace, docházelo při zkušebních volání přes puTTy k běhu právě této předchozí verze programu.

Reálně totiž došlo k zacyklení během inicializace UARTu pro komunikaci se zařízením. To lze vyčíst, pokud po spuštění debuggování několikrát ručně stopneme ladící proces. Pokaždé, i když se o to pokusíme řádově několik minut po sobě, program skončí v inicializaci UART a nastavování parametrů, handlování zpráv (přestože nemá žádné, které by mohl zpracovávat) atd.

Po tomto zjištění bylo vyzkoušeno rozběhnout na desce zkušební projekt, který přímo implementuje použití UART na testované desce.

9.6.1 Problém s CMSIS UART driverem

Projekt s názvem cmsis_lpuart_edma_transfer je ukázkový projekt, který má vracet zadaných 8 znaků. Tento projekt, ale není možné, přestože byl napsán jako ukázkový projekt pro vývojovou desku mimxrt1064_evk, rozběhnout na této desce.

Při pokusu o nahrání a spuštění tohoto souboru na vývojovém kitu, se program sice načte a vypíše do konzole začínající ukázkový text a informaci o tom, že má posílat všech osm znaků zpět. Ale ihned se dostane do SIGTRAP, způsobené chybou s definicí uartHandle, který je NULL.

K definování uartHandle dochází ve zdrojovém souboru fsl_adapter_lpuart, přesněji k získání jeho aktuálního stavu, ten je, avšak Null. Nedojde tak ke kompletní inicializaci UARTu a zacyklení okolo této chyby.

```
__assert_func (file=0x6468 "/home/pavlina/Documents/SDK/EVK-MIMXRT1064-hello_world/SDK1064/components/uart/fsl_adapter_lpuart.c",
line=433, func=0x70c8 <__func__.13> "HAL_UartInterruptHandle", failedExpr=0x6454 "NULL != uartHandle") at /home/pavlina/Document
s/SDK/EVK-MIMXRT1064-hello_world/SDK1064/devices/MIMXRT1064/utilities/debug_console_lite/fsl_assert.c:67
67              __BKPT(0);
```

Obrázek 19. uartHandle je Null

Vzhledem k nemožnosti rozběhnout ani ukázkový projekt, který by mělo být možné na desce rozchodit se s největší pravděpodobností jedná o problém s drivery na zařízení, ať už hardwarové nebo softwarové nastavení. Tento problém se nepovedlo vyřešit a eRPC se tak nepovedlo rozchodit pro vyzkoušení testování na zvoleném zařízení se zvoleným projektem. Avšak vzhledem k tomu, jakým způsobem funguje eRPC došlo k jednoduché úpravě proxy, tak aby odpovídala logické užití s eRPC.

9.7 Nahrazení eRPC logiky pro účely testování

Vzhledem k tomu, že základní logika komunikace se zařízením je již obsažena v předchozí implementaci pro běh na desce, došlo pouze k překopírování souborů zastřešujících logiku do složky dedikované pro testování běhu s použitím eRPC.

9.7.1 AFL++ proxy

Vzhledem k tomu, že proxy zastřešující hlavní komunikaci mezi zařízením a fuzz frameworkem funguje v logice věci stejně je zde potřeba udělat pouze jedna změna.

Snaha je zde aby implementace co nejvíce připomínala použití eRPC, hlavní myšlenkou pro použití tohoto přístupu bylo odlehčení vývojové desce. Na serveru, zařízení, běží program,

kterému jsou posílány vstupní data. Tím pádem je potřeba, aby narozdíl v předchozí implementaci, byl program nahrán na desku ihned ze začátku a nedocházelo k jeho opětovnému nahrávání při novém testovacím případu.

Jediná větší změna v struktuře tak spočívá v posunutí funkcionality, která nahraje testovaný program. Ta se posune před hlavní smyčku, pod definice fixních proměnných jako cesty k logovacímu souboru nebo inicializaci paměti a forkserveru. Ponechá se i sleep, aby mohl být program v pořádku nahrán.

Drobnými změnami v kódu jsou pak dvě věci. Změna logovacího souboru na soubor pro logování v tomto použitém přístupu a změna skriptu, který spouští debuggování. Oba tyto soubory jsou přímo vytvořeny jenom pro účely testování tímto přístupem, a to z toho důvodu, aby bylo snadné změnit způsob testování. Tudíž pokud bychom chtěli běžet testování na zařízení s čistým nahráváním programu pro každý test case, stačí nám tak jenom změnit cesta k použitému afl proxy programu při spouštění fuzz frameworku.

9.7.2 Start_debug_session_erpc a .gdbLedControlERPC soubory

Tyto dva soubory, stejně jako AFL++ proxy vycházejí z již předchozí implementace. Dochází zde jenom k malým změnám. V souboru start_debug_session_erpc dochází k změně názvu souboru, do kterého se ukládá logovací záznam z debuggování a změně skriptu, který definuje chování na nastavených breakpointech, na .gdbLedControlERPC.

V tomto výše zmíněném souboru došlo k změně na breakpointech nastavených na výstupech z funkcí. V přechozí implementaci na těchto breakpointech (3-8) došlo k ukončení debuggování. Pro tento typ přístupu je poslán signál na pokračování v běhu programu jako v případech 1 a 2 breakpointu.

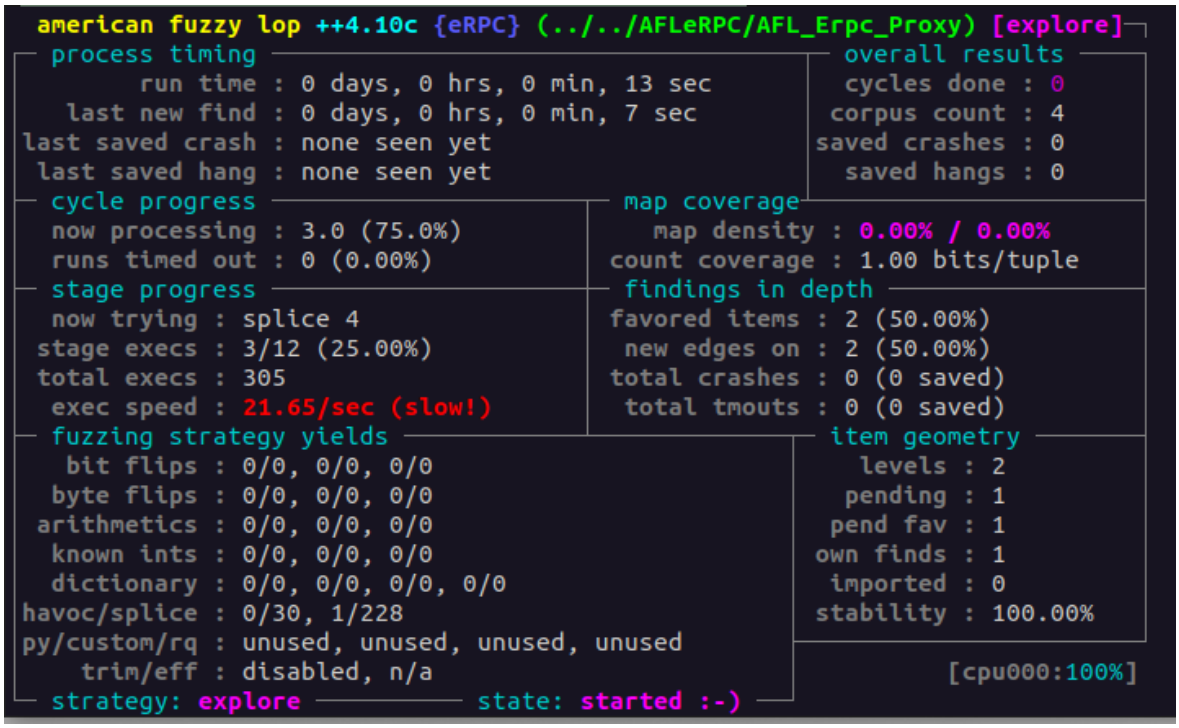
9.8 Spuštění AFL++

Vzhledem k implementaci byl AFL++ spuštěn 3x pro získání adekvátních vstupních dat.

Prvně, pro ověření toho, že je vývojová deska vůbec schopna zvládat běh bez čistého startu pro každý test case. K tomuto účelu byla instrumentace získaná z logovacího souboru nahrazena fixním nastavením hodnoty pro area mapping, které se nachází již ve vzorovém kódu. Tudíž stačilo tuto část pouze od komentovat.

Fuzz framework byl spuštěn stejným způsobem jako předchozí instance: AFL_SKIP_CPUFREQ=1 AFL_SKIP_BIN_CHECK=1 afl-fuzz -M eRPC -i INPUTS/ -o

OUTPUTS/ -t 15000 -m 200 -D -- ../../AFLeRPC/AFL_Erpc_Proxy. S rozdílem, že došlo ke změně výstupního souboru na eRPC. Samozřejmě byl fuzzer spuštěn ne na afl_proxy ale na AFL_Erpc_Proxy soubor. Po krátkém běhu, ale dochází k zmražení VM a jediná možnost východu z tohoto stavu je natvrdo ukončit virtuální stroj.



Obrázek 20. Běh AFL++ bez instrumentace

Po, ale opětovném spuštění s jiným zdrojem připojeným k hostitelskému zařízení, nedochází již k pádu virtuálního stroje, ale pouze k zmražení testování ihned po jeho startu. To je zapříčiněno přetečením zásobníků, protože AFL posílá za krátkou dobu velké množství dat a zařízení je nestíhá zpracovávat.

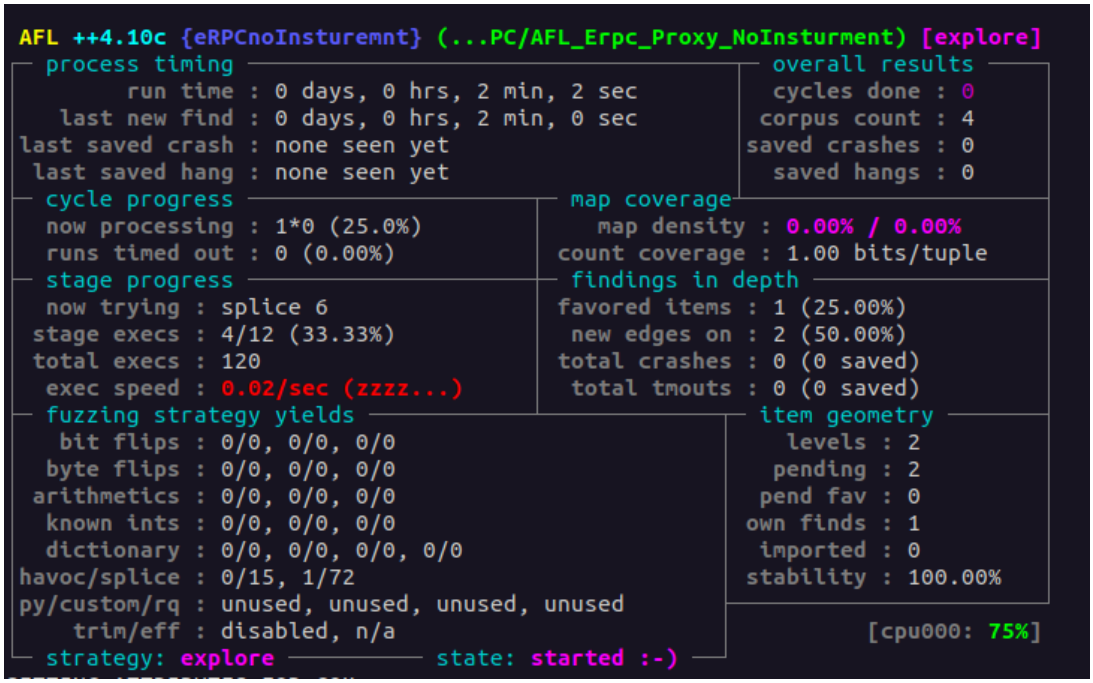
Obrázek 21. Zmražení testování v AFL UI

Na obrázku výše je vidět, že položka run time (čas po který probíhá aktuální testování) je 0, to ale není možné, protože testování běží již nějakou dobu. Je tedy jasné že během vykonávání došlo k chybě, která zapříčinila zmražení a zastavení testování. Nejspíše neschopnost fuzz frameworku navázat komunikaci se zařízením a poslat další data.

Toto se nejspíše stalo i v případě výše, kdy po pár sekundách došlo k zahlcení zařízení a hostitelské zařízení se nejspíše samo pokusilo ukončit komunikaci čímž se zamrzlo. Tato myšlenka se objevila jako mylná a chyba byla v rámci proxy AFL, kdy došlo k posunutí __afl_end_testcase funkce, která informuje, že došlo k dokončení celého testovacího případu, který by měl být až po podmínce týkající se délky vstupu.

Po vyřešení této chyby a opětovnému spuštění se jevilo testování více stabilněji, ale po pár sekundách došlo k tomu, že se sám odpojil od serveru. Server se dostal do neodpovídajícího stavu a bylo nutné ho ručně ukončit. S největší pravděpodobností došlo k násilnému ukončení komunikačního portu na zařízení a nucenému restartu, přičemž ztratil možnost komunikace se serverem.

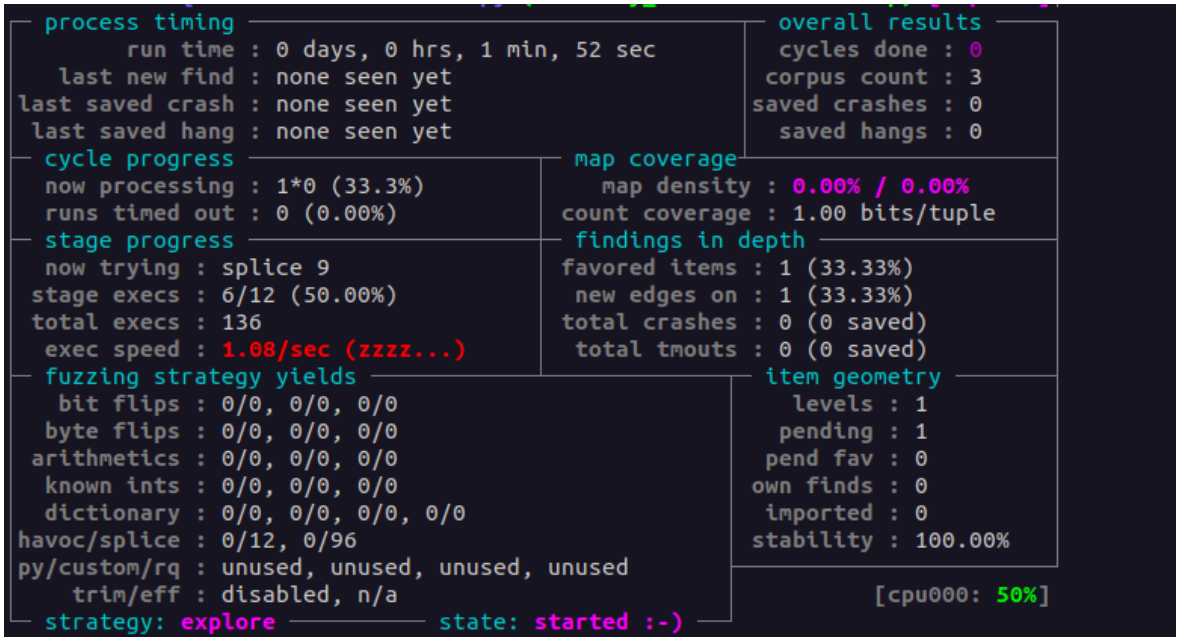
To, co je, ale zajímavé tak došlo k rychlému poklesu výkonosti. Kdy začátku se opravdu testování drží u 20 exekucí za sekundu, avšak po chvíli začne rapidně klesat. Konečná hodnota je pak skoro nulová.



Obrázek 22. Konečná rychlost testování

Druhý běh byl stále s vypnutou instrumentací, ale s implementovaným čekáním na zapsání dat. Tím jsme dali zařízení čas na to se vzpamatovat a nezásobovat ho velkým množstvím dat za krátký čas.

Vzhledem k tomu, že implementována hodnota čekání je 1 sekunda, celkový počet exekucí za sekundu je okolo 1. Po minutě, ale nastává zlom, kdy dojde k zmražení celého virtuálního stroje. I při opětovném restartování stroje a spuštění, dochází k zmražení. Vzhledem k tomu, že při „neuspáni“ procesu jsme schopni shodit server, je dost možná, že je i toto chyba serveru.



Obrázek 23. Běh bez instrumentace s implementovaným umělým čekáním

Pokud jde o třetí běh, již se zapnutou instrumentací, spuštěn úplně stejným příkazem jako výše vypsaným, nedostaneme se ani k základní informační obrazovce. Testování skončí ve čtyřech možných stavech: chyba v komunikaci s forkserverem, žádná instrumentace, zacyklení ve vykonávání programu a problém s aktualizací pokrytí.

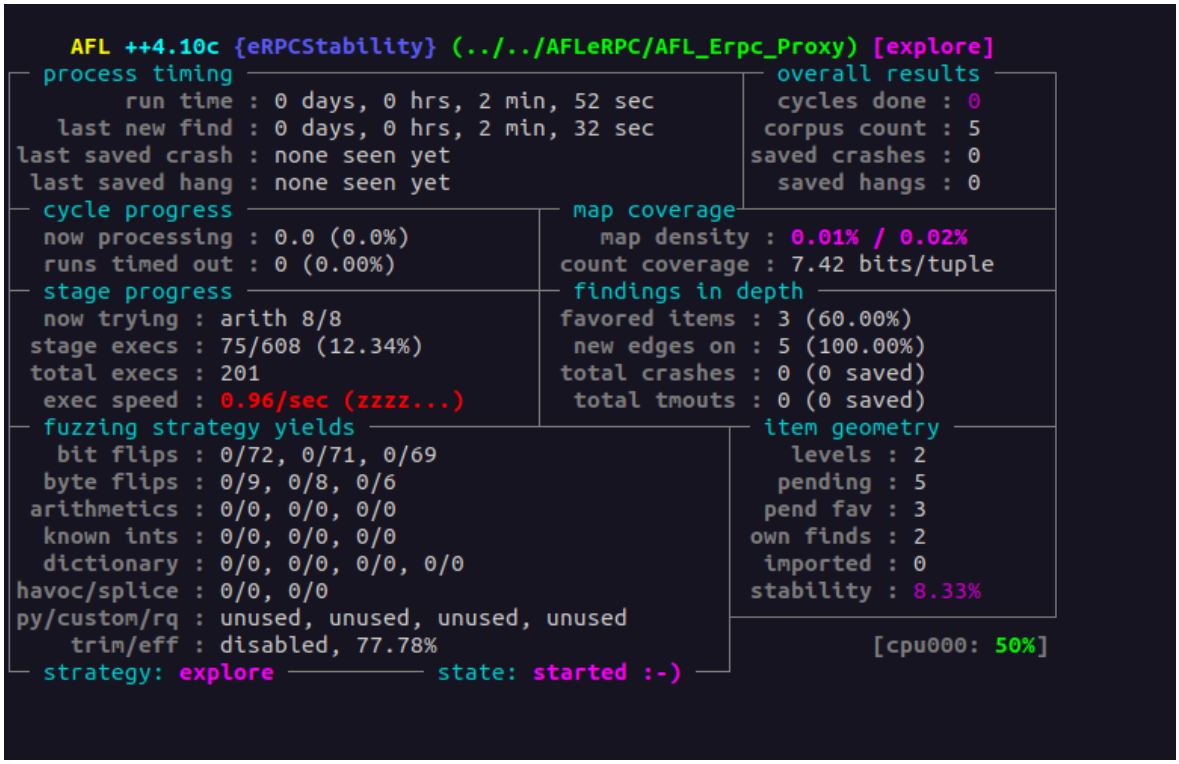
Chyba s fork serverem není tak častá a je zapříčiněna tím, že dojde k násilnému a nečekanému ukončení gdb, během běhu. Povětšinou vyvolaná v moment, kdy po zacyklení dojde ručně k zastavení exekuce arm-none-eabi-gdb. Taktéž k ní může dojít, pokud není dobře ukončené spojení přes sériový port (otevřeno je pak několik možných spojení) nebo pokud fuzzer posílá stále další a další data. V ten moment dojde nejspíše k pádu komunikace mezi zařízením a GDB serverem.

Druhá chyba je o něco častější a vyskytuje se v moment, kdy dojde k přístupu k logovacímu souboru moc brzo. Debugger nebyl schopný přesměrovat výstup do onoho souboru a funkce checkExecution již začala s čtením. Tato chyba se dá vyřešit nastavením uspání na určitou dobu.

Třetí zmíněná chyba je fatální. Jedná se o chybu, kdy během toho, co program ještě běží dochází k poslání dalších dat na vstup. Ladící proces se zastaví a celá aplikace čeká na signál k pokračování, který ale nemůže dostat. Vše se tedy jeví že funguje až na to že vůbec.

Poslední chyba, je zamrznutí fuzz frameworku během aktualizace pokrytí programu po získání adres z logovacího souboru. Tato chyba byla, ale zapříčiněna chybou v debugConsole

souboru, kdy docházelo k přetečení paměti při ukládání breakpointu. Po nastavení podmínky, aby pokud se hodnota proměnné j, udávající aktuální počet nalezených breakpointu překročí 8, obnovila zpět společně s hodnotou délky proměnné len pro strukturu x, na nulu. Tím začalo prohledávání jakoby od začátku. Tato chyba podle všeho byla schopná vyvolat i některé ostatní výše zmíněné chyby.

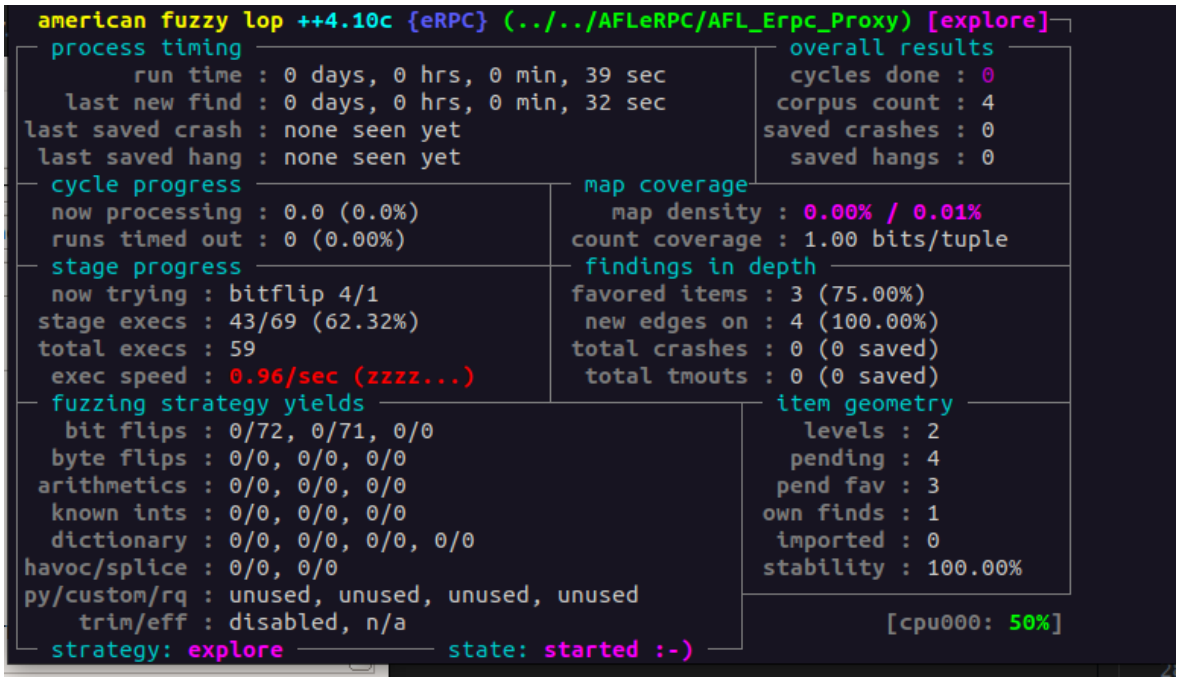


Obrázek 24. Běh na zařízení s nahrazením eRPC

Hlavní, ale zásadní problém ale vzniká v práci se samotnou instrumentací. Kromě nemožnosti přerušení nebo automatického restartování při výše zmíněné chybě, je zde i nemožnost nějakým způsobem promazávat logovací soubor. V tomto souboru se tak hromadí všechna data ze všech běhů testovacích případů a tím pádem dochází i ke zkreslení při zjišťování použité cesty přes breakpointy. Tedy kromě zkreslení a další nutné úpravy práce s logovacím souborem, časem dojde i k zpomalení, kdy jsme nuceni parsovat soubor s několika stovkami řádků informací, z nichž nás zajímají například poslední dva. Pokud neupravíme logiku práce s logovacím souborem, dochází ke snížení stability, jak je vidět na obrázku číslo 26 výše.

Pokud ovšem opravíme problémy se stabilitou, tím že kromě hodnoty proměnné j budeme hlídat i to, jestli právě nejsem na breakpointu 2, který je počáteční a všechny vstupy by jim

měly projít jako první. Stabilita se rázem dostane zpátky na 100 %. Avšak i zde občas dochází k zamrznutí virtuálního stroje.



Obrázek 25. Běh s opravenou stabilitou

Celá tato implementace se vzhledem k tomu, že je o něco náročnější na implementaci než samotné běžení na zařízení, jeví nepoužitelně, avšak vzhledem k nemožnosti získání dat v reálném čase z připojeného ladícího zařízení, je to nejlepší možný výsledek, co jsme schopni z testování dosáhnout.

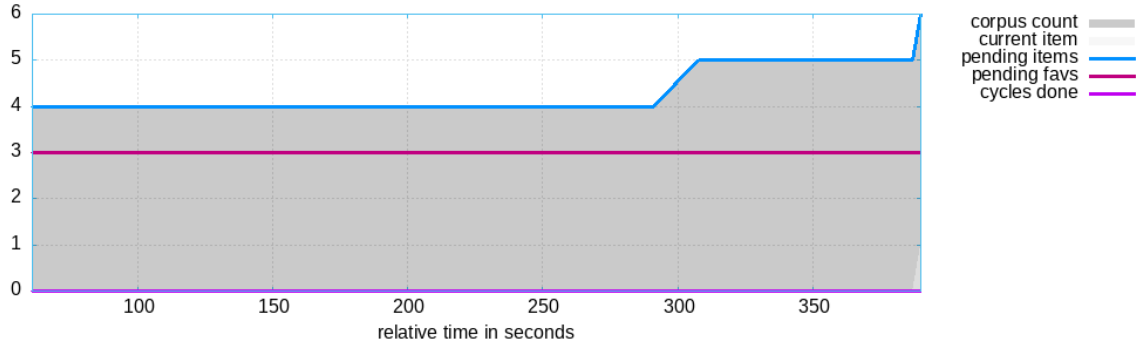
9.9 Výsledky testování

Z této implementace se kromě nějakého zrychlení řádných jiných dat nedostalo, pokrok přinesl až poslední běh s opravenou stabilizací. Avšak nelze s jistotou říct, že zrychlení, které jsem dosáhli bez použití eRPC bychom dostali i s jeho použitím. Je zde ovšem takový předpoklad, protože nejvíce času v původní implementaci pro běh na zařízení, zabírá čekání při nahrávání image do paměti zařazení.

Navíc zrychlení jsme dostali jenom v moment, kdy nepoužíváme instrumentaci. Pokud čteme data z logovacího souboru, určitou formu zrychlení dostaneme, avšak není až tak znatelné.

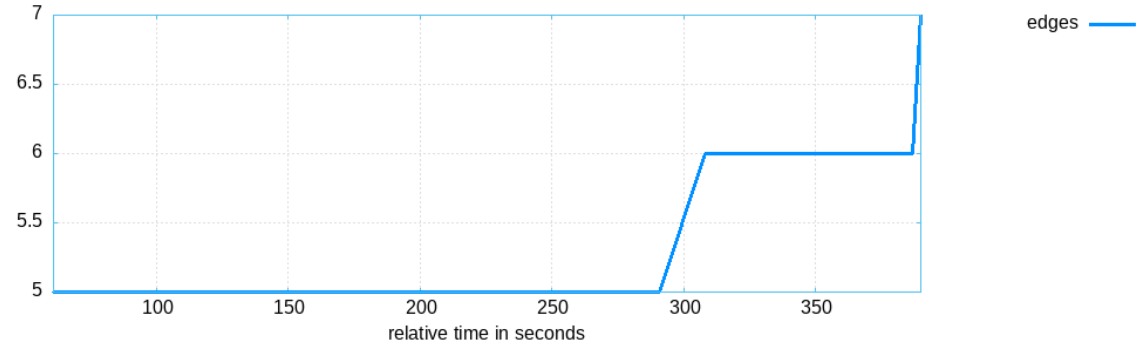
Co se týče dat získaných z posledního běhu, kromě toho, že jsme byli schopni v úseku šesti minut kdy AFL++ běželo, najít 6 různých vstupů. Což je momentálně nejvyšší možný počet

za nejkratší dobu. Avšak poslední nalezený vstup nelze prozkoumat, jelikož se jedná o vadný soubor, tudíž můžeme předpokládat, že právě to zapříčinilo jeho zařazení do nových vstupů.



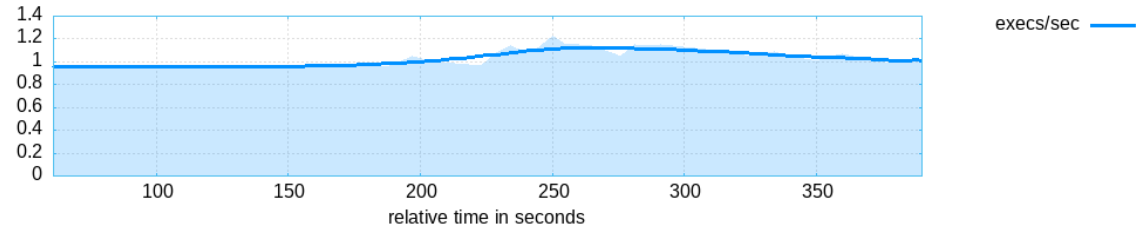
Obrázek 26. Počet generování vstupních dat pro testování s nahrazením eRPC

Jak je vidět na obrázku níže, počet hran stoupá v momentě nalezení pátého a šestého použitelného vstupu. Jinak zůstává po celou dobu testování stabilní.



Obrázek 27. Počet nalezených hranic pro testování na reálném s nahrazením eRPC

Co se týče rychlosti, jak již bylo řečeno, nedochází k razantnější změně. Čas je sice stabilní s minimálními výkyvy spíše směrem vzhůru, ale i tak to není výkonost, kterou bychom hledali. To vše je ale, stejně jako v případě běhu na zařízení, zapříčiněno čekáním před čtením logovacího souboru.



Obrázek 28. Vývoj počtu exekucí za sekundu pro testování s nahrazením eRPC

Ovšem samotná implementace eRPC je ze všech tří použitých přístup, ta nejvíce složitá. Je potřeba razantně zasáhnout do zdrojového kódu, což v předchozí dvou nebylo nutné nebo se

jednalo jenom o drobné změny. Potřeba je také kromě podpůrných podprogramů, které jsou stejné jako v případě běhu přímo na desce, ještě vyřešit soubory potřebné pro funkčnost samotného eRPC.

Kromě větší pracnosti při samotné implementaci jsou zde i další problémy. ERPC se primárně soustředí na testování jednotlivých funkcích. Což znamená, že při složitějších programech je potřeba definovat více testovacích funkcí, pro které budeme generovat dat. Ztrácíme tak výhodu u předchozích dvou řešení, kdy nejsem limitováni funkcemi, ale testujeme celý program z uživatelského pohledu.

Zásadní problém je ale s instrumentací a sběrem informací o prozkoumaných cestách programu. Od eRPC nejsem schopni získat informace o adresách na kterých se nacházel, a tak pořád naše jediná cesta je přes ladící zařízení. Tady, ale ovšem opět vzniká problém, a to je aktuální nemožnost dostat ze zařízení data v reálném čase. Jsme proto nuceni získávat je z extérního souboru, který je ale možné vymazat pouze znovu nahráním programu do paměti. Tomu se, ale chceme v rámci implementace eRPC vyhnout, je to jeho největší výhoda oproti předchozímu přístupu. Dostáváme se pak k tomu, že procházíme několika řádkový soubor, ale použitelné informace se nacházejí až na jeho samém konci. Ztrácí se čas a ovlivňuje se i výkon.

ZÁVĚR

Cílem práce bylo vyzkoušet tři možné přístupy k fuzz testování reálných zařízení a zjistit jejich časovou a implementační náročnost. Jednalo se o testování s použitím simulátoru, běhu přímo na testovaném zařízení a s použitím eRPC. Pro všechny způsoby byl použit stejný fuzz framework, vývojová deska, a i testovaný program. Vzhledem k tomu, že se kvůli chybě UART zařízení na vývojové desce nepovedlo rozběhnout eRPC, byla tato část nahrazena implementací, která simulovala eRPC logiku. Díky tomu jsme schopni si představit, jak by mohl vypadat výsledek testování s použitím eRPC.

Implementace jednotlivých přístupů k testování (simulátor, běh na zařízení a eRPC) mají vzestupnou složitost. Nejjednodušeji, pokud máme dostupný virtualizovaný systém, je rozběhnout fuzz framework pro testování přes simulátor. Je to také jeden z nejčastěji doporučovaných přístupu k testování binárních souborů. Co se týká rychlosti jeví se také nejlépe, avšak časem došlo k propadu, kdy rozdíl mezi jednotlivými přístupy nebyl až tak markantní jako ze začátku. Z 70 exekucí za sekundu se po několika minutách testování dostáváme na cca 3 exekuce za sekundu.

Nejsložitější implementace je implementace s použitím eRPC. Je potřeba vyřešit několik problémů vůbec předtím, než můžeme začít implementovat komunikaci mezi fuzz frameworkem a testovaným zařízením. Kromě toho byl tento přístup neúspěšný i z toho důvodu, že v průběhu implementace se přišlo na chybu, která ovlivňovala celou implementaci a nepovedlo se ji vyřešit. Jedná se o chybu s použitím UART komunikace, kdy během inicializace nedošlo k správnému nastavení a program zůstal zacyklen, přestože vše vypadá v pořádku a funkční.

Vzhledem k časovému tlaku tak nedošlo k úplnému vyřešení problému s eRPC, ale jenom k určitému nahrazení eRPC podobnou logikou, jakou mělo být použito eRPC. Zde ovšem došlo k několika zjištění neoptimálnosti celého přístupu. Ať už klíčovému získávání informací o průběhu testovacího případu na zařízení nebo i samotného vykonávání, kde nebylo možné nechat testování běžet delší část kvůli úplnému zamrznutí virtuálního stroje. I přesto byl zjištěn základní poznatek o vyšší rychlosti, která dosahovala cca poloviny rychlosti simulátoru. V případě implementované instrumentace již zrychlení nebylo tak vysoké, ale vzhledem k rychlosti běhu na simulátoru se dá považovat za dostatečné.

Zlatá střední cesta se tak jeví samotná implementace běhu na zařízení. Zde se ovšem dostáváme k velkému rychlostnímu propadu, kdy počáteční rychlost testování je 0,3 exekuce za

sekundu. Na rozdíl od simulátoru je ale tato rychlost konstantní a nemění se markantně po dobu celého běhu.

Ve všech třech výše vypsaných přístupech je možnost zrychlení a zefektivnění testování. U simulátoru můžeme zvažovat jiný, více podporovaný simulátor, například QEMU s možností persistentního módu, kdy jsme schopni fuzz frameworku říct, aby se cyklil jenom okolo určitého rozsahu adres.

Naopak u eRPC a exekuci přímo na zařízení můžeme zvažovat jiný přístup ke sběhu informací o běhu. Pokud bychom byli schopni optimalizovat získávání navštívených adres anebo skoků při vykonávání exekuce, jsme schopni dosáhnout lepších časových výsledků.

Otázkou je také použití jiného fuzz frameworku, který by mohl být lépe optimalizovaný na celou problematiku, například GDBFuzz.

Každé implementované řešení má určitě svou použitelnost, i když rychlostně oproti exekuci na hostitelském zařízení, pokulhávají. Nejlépe si, i přes to, že byla nejpomalejší, vedla implementace pro běh na zařízení. V této implementaci se totiž povedl najít vstup, který nebyl nalezen v žádném jiném běhu. Pokud by se tento přístup podařilo lépe optimalizovat, mohlo by se jednat o nejlepší přístup k fuzz testování na zařízení.

SEZNAM POUŽITÉ LITERATURY

[1] HARPER, Allen, Daniel REGALADO, Ryan LINN. 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 Hackers Playbook 2: Practical Guide To Penetration Testing. South Carolina: Secure Planet, 2015. ISBN 978-1512214567.

[3] Køien, G.M., Øverlier, L. A Call for Mandatory Input Validation and Fuzz Testing. Wireless Pers Commun (2023). https://doi.org/10.1007/s11277-023-10431-2

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

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

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

[7] EISELE, Max, Marcello MAUGERI, Rachna SHRIWAS, Christopher HUTH a Ciampaolo BELLA. Embedded fuzzing: a review of challenges, tools, and solutions [online]. 2022, 18 [cit. 2023-12-21]. Dostupné z: doi:https://doi.org/10.1186/s42400-022-00123-y

[8] CHEN, Chen, Baojiang CUI, Jinxin MA, Runpu WU, Jianchao GUO a Wenqian LIU. A systematic review of fuzzing techniques. Computers & Security [online]. 2018, (75), 118-137 [cit. 2024-01-08]. ISSN 0167-4048. Dostupné z: doi:https://doi.org/10.1016/j.cose.2018.02.002

[9] The yacc grammar file. IBM [online]. 2023 [cit. 2024-01-08]. Dostupné z: https://www.ibm.com/docs/en/aix/7.3?topic=information-yacc-grammar-file

[10] What Is Dynamic Analysis. Total View [online]. 2020 [cit. 2024-01-08]. Dostupné z: https://totalview.io/blog/what-dynamic-analysis

[11] LibFuzzer - a library for coverage-guided fuzz testing. Llvm [online]. c2003-2023 [cit. 2024-01-08]. Dostupné z: https://llvm.org/docs/LibFuzzer.html

[12] American fuzzy lop (2.52b). Lcamtuf.coredump.cx [online]. [cit. 2024-01-08]. Dostupné z: https://lcamtuf.coredump.cx/afl/

[13] AFLplusplus [online]. [cit. 2024-01-08]. Dostupné z: https://aflplus.plus/

[14] ClusterFuzz. GitHub [online]. c2024 [cit. 2024-01-15]. Dostupné z: https://github.com/google/clusterfuzz

[15] OSS-Fuzz. GitHub [online]. c2024 [cit. 2024-01-15]. Dostupné z: https://github.com/google/oss-fuzz

[16] BUTT, Muhammad Arif, Zarafshan AJMAL, Zafar Iqbal KHAN, Muhammad IDREES a Yasir JAVED. An In-Depth Survey of Bypassing Buffer Overflow Mitigation Techniques. Applied Sciences [online]. 2022, 12(13) [cit. 2024-01-08]. Dostupné z: doi:https://doi.org/10.3390/app12136702

[17] What Is Fuzz Testing? Code Intelligence [online]. c2023 [cit. 2024-01-15]. Dostupné z: https://www.code-intelligence.com/what-is-fuzz-testing

[18] What is Regression Testing? Smartbear [online]. c2023 [cit. 2024-01-15]. Dostupné z: https://smartbear.com/learn/automated-testing/what-is-regression-testing/

[19] Embedded Hardware Simulation. Swedish Embedded [online]. c2024 [cit. 2024-01-15]. Dostupné z: https://swedishembedded.com/simulation/

[20] Simulating Embedded Devices. Nuvation Engineering [online]. 2019 [cit. 2024-01-15]. Dostupné z: https://www.nuvation.com/resources/article/simulating-embedded-devices

[21] SANDOVAL, David, Antonio VÁZQUEZ, Fran Álvarez WIC a Jesús María Gómez MORENO. OWASP FSTM step 6: firmware emulation. Tarlogic [online]. 2022 [cit. 2024-01-08]. Dostupné z: https://www.tarlogic.com/blog/owasp-fstm-step-6-firmware-emulation

[22] About QEMU. QEMU [online]. c2023 [cit. 2024-01-08]. Dostupné z: https://www.qemu.org/docs/master/about/index.html

[23] *Qemu fails to start when linking statically* [online]. edited on Apr 30, 2021 [cit. 2024-02-05]. Dostupné z: https://github.com/AFLplusplus/AFLplusplus/issues/892

[24] How to use the persistent mode in AFL++'s QEMU mode. *GitHub* [online]. 2020 [cit. 2024-02-05]. Dostupné z: https://github.com/AFLplusplus/AFLplusplus/blob/stable/qemu_mode/README.persistent.md

[25] ANH QUYNH, Nguyen a Dang HOANG VU. *Unicorn: Next Generation CPU Emulator Framework* [online]. 2015 [cit. 2024-02-07]. Dostupné z: https://www.unicorn-engine.org/BHUSA2015-unicorn.pdf

[26] *The JIT compiler* [online]. 2023 [cit. 2024-02-07]. Dostupné z: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=reference-jit-compiler

[27] *Unicorn & QEMU* [online]. [cit. 2024-02-07]. Dostupné z: https://www.unicorn-engine.org/docs/beyond_qemu.html

[28] *Unicorn-based binary-only instrumentation for afl-fuzz* [online]. 2019 [cit. 2024-02-07]. Dostupné z: https://github.com/AFLplusplus/AFLplusplus/tree/stable/unicorn_mode

[29] *Renode* [online]. c2010-2014 [cit. 2024-02-07]. Dostupné z: https://opensource.antmicro.com/projects/renode/

[30] *Co-simulating your verilated model* [online]. 2024 [cit. 2024-02-07]. Dostupné z: https://renode.readthedocs.io/en/latest/advanced/writing-peripherals.html

[31] *Fuzzing Zephyr with AFL and Renode* [online]. 2023 [cit. 2024-02-07]. Dostupné z: https://renode.io/news/fuzzing-zephyr-with-afl-renode/

[32] MATTURRO, Brein a Linda ROSENCRANCE. *Remote Procedure Call (RPC)* [online]. c2019-2024 [cit. 2024-02-07]. Dostupné z: https://www.techtarget.com/searchapparchitecture/definition/Remote-Procedure-Call-RPC

[33] CERVENKA, Dusan. Home. *Github* [online]. 2017, Apr 13, 2018 [cit. 2024-02-20]. Dostupné z: https://github.com/EmbeddedRPC/erpc/wiki

[34] ERPC. *Github* [online]. c2024 [cit. 2024-02-20]. Dostupné z: https://github.com/EmbeddedRPC/erpc

[35] CONTRIBUTOR, TechTarget. IDL (interface definition language). *TechTarget* [online]. c2024 [cit. 2024-02-20]. Dostupné z: https://www.techtarget.com/whatis/definition/IDL-interface-definition-language

[36] Using AFL to fuzz ARM binaries on a Rasbperry Pi and with AFL QEMU mode. *Tickelton* [online]. 2020, August 11, 2020 [cit. 2024-02-21]. Dostupné z: https://tickelton.gitlab.io/articles/afl-arm-rpi/

[37] EISELE, Max, Daniel EBERT, Christopher HUTH a Andreas ZELLER. Fuzzing Embedded Systems using Debug Interfaces. In: JUST, René a Gordon FRASER. *ISSTA 2023: Proceedings of the 32nd ACM SIGSOFT International Symposium on*

Software Testing and Analysis. 2023. New York: Association for Computing Machinery, 13 July 2023n. l., s. 1031-1043. ISBN 979-8-4007-0221-1. Dostupné z: doi:https://doi.org/10.1145/3597926.3598115

[38] GDBFuzz: Debugger-Driven Fuzzing. *GitHub* [online]. c2024 [cit. 2024-02-21]. Dostupné z: https://github.com/boschresearch/gdbfuzz

[39] How to expand AFL++ to fuzz embedded device without any binary nor debug interface? #1170. *GitHub* [online]. 2021 [cit. 2024-02-25]. Dostupné z: https://github.com/AFLplusplus/AFLplusplus/discussions/1170

[40] MUENCH, Marius, Dario NISI, Aur´elien FRANCILLON a Davide BALZAROTTI. *Avatar2: A Multi-target Orchestration Platform* [online]. 2018 [cit. 2024-02-25]. Dostupné z: https://www.s3.eurecom.fr/docs/bar18_muench.pdf

[41] MOROO, Akira a Yuichi SUGIYAMA. ARMored CoreSight: Towards Efficient Binary-only Fuzzing. *Ricerca Security* [online]. 2021 [cit. 2024-02-25]. Dostupné z: https://ricercasecurity.blogspot.com/2021/11/armored-coresight-towards-efficient.html

[42] CoreSight Architecture. *Arm Developer* [online]. c2024 [cit. 2024-02-25]. Dostupné z: https://developer.arm.com/Architectures/CoreSight%20Architecture

[43] Fuzzing binary-only targets. *AFLplusplus* [online]. [cit. 2024-02-25]. Dostupné z: https://aflplus.plus/docs/fuzzing_binary-only_targets/

[44] American Fuzzy Lop plus plus (AFL++). *GitHub* [online]. c2024 [cit. 2024-04-23]. Dostupné z: https://github.com/antmicro/AFLplusplus

[45] Satisfiability modulo theories. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, c2001-2024 [cit. 2024-04-23]. Dostupné z: https://en.wikipedia.org/wiki/Satisfiability_modulo_theories

[46] I.MX RT1064 Evaluation Kit. In: *NXP* [online]. c2006-2024 [cit. 2024-05-03]. Dostupné z: https://www.nxp.com/design/design-center/development-boards-and-designs/i-mx-evaluation-and-development-boards/i-mx-rt1064-evaluation-kit:MIMXRT1064-EVK

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

ACM	Abstract Control Model
AFL	American Fuzzy Loop
AFL++	American Fuzzy Loop plus plus
API	Application Programming Interface
ARM	Advanced RISC Machines
BIOS	Basic Input/Output System
CI/CD	Continuous Integration Continuous Delivery
CMSIS	Common Microcontroller Software Interface Standard
CPU	Central processing unit
eRPC	Embedded Remote Procedure Call
Fuzzer	Fuzz Framework
GDB	GNU Project Debugger
GNU	GNU's Not Unix
HW	Hardware
IDL	Interface Definition Language
IPC	Inter-process communication
JIT	Just In Time
JTAG	Joint Test Action Group
KVM	Kernel-based Virtual Machine
LED	Light-emitting Diode
LLVM	Low Level Virtual Machine
LPUART	Low Power Universal Asynchornus Reciver Transmitter
MIPS	Microprocessor without Interlocked Pipelined Stages
MPU	Memory Protection Unit

OS	Operační systém
PC	Personal Computer
QEMU	Quick Emulator
RPC	Remote Procedure Call
SIGTRAP	Signal Trap
SMT	Splnitelnost Modulo Teorie
SoC	Systems on chip
SPARC	Scalable Processor ARChitecture
SQL	Structured query language
SW	Software
SWD	Serial Wire Debug
TCP/IP	Transmission Control Protocol/Internet Protocol
UART	Universal Asynchronous Receiver - Transmitter
USART	Universal Synchronus Asynchornus Reciver Transimtter
USB	Universal Serial Bus

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I: OBSAH CD

Obsah CD:

- Simulátor – obsahuje upravené soubory, použité pro testování na simulátoru
- Deska – obsahuje upravené a podpůrné soubory použité pro testování na desce
- eRPC – obsahuje upravené a podpůrné soubory použité pro testování na desce s eRPC