BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2017
http://excel.fit.vutbr.cz

# Dynamická analýza funkcí a parametrů pomocí nástroje ANaConDA

Monika Mužikovská*


open("Paramete","r");


**Abstrakt**

Paralelní programy jsou náchylné ke specifickým a těžko odhalitelným chybám. Framework ANaConDA poskytuje podporu pro dynamickou analýzu, což je jedna z úspěšných technik pro testování paralelních programů. Tento článek popisuje implementaci podpory pro analyzování funkcí a jejich parametrů, jelikož se jedná o nedílnou součást každého programu a bez daných informací se při testování programů téměř nemůžeme obejít. Cílem rozšíření je zjednodušit tvorbu analyzátorů, které ke své práci potřebují informace o volání funkcí, o zadaných argumentech i návratových hodnotách. Protože framework ANaConDA provádí analýzu programu na binární úrovni, je získávání těchto informací značně obtížné. Pro demonstraci celé problematiky bude popsán analyzátor pro kontrolu práce se soubory, který využívá implementovaná rozšíření. Tato práce je prototypové řešení problému, protože v současné době ještě neexistuje podpora pro všechny datové typy (např. uživatelské objekty) a pro reálné nasazení je také potřeba ověřit správnost funkcionality na různých platformách.

**Klíčová slova:** ANaConDA — Dynamická analýza — Parametry funkcí — Práce se soubory

**Přiložené materiály:** *N/A*

*xmuzik05@stud.fit.vutbr.cz, *Fakulta informačních technologií, Vysoké učení technické v Brně*

## 1. Úvod

Charakteristickým rysem paralelních programů je jejich nedeterministické chování, které velmi ztěžuje odhalení případných chyb. Za účelem testování paralelních programů byly vyvinuty speciální techniky, jednou z nichž je dynamická analýza. Ta spočívá v analýze běhu binárních programů a typicky zahrnuje sledování nebo emulaci jednotlivých instrukcí. Dynamická analýza je velmi náročný proces, který není možné používat jako běžnou součást vývoje programů, a proto existují snahy o zjednodušení tohoto přístupu. Jedním z velmi rozšířených nástrojů, které zjednodušují dynamickou analýzu, je valgrind [1]. Ten ovšem běh programu serializuje do jednoho procesu, a proto není příliš vhodný pro analýzu paralelních programů. To byl také jeden z důvodů vzniku nástroje ANaConDA.

Framework ANaConDA, který je již několik let vyvíjen na FIT VUT v Brně výzkumnou skupinou VeriFIT, poskytuje podporu pro tvorbu dynamických analyzátorů a řeší problémy spojené se sledováním běhu programů napsaných v jazycích C/C++. V současné době se zaměřuje především na vícevláknové programy. Je postaven na nástroji Intel PIN, který ANaConDA využívá pro monitorování běhu programu.

ANaConDA poskytuje rozhraní pro tvorbu analyzátorů za účelem odhalení konkrétních chyb. Dosud ovšem chyběla podpora pro sledování parametrů a návratových hodnot volaných funkcí. Motivací pro

implementaci podpory byl aktuálně probíhající výzkum ohledně tzv. kontraktů. Zjednodušeně řečeno se jedná o sekvenci funkcí, které operují nad stejným objektem, takže konkrétní argumenty jednotlivých funkcí jsou klíčovou informací pro jejich analýzu.

Tato práce v sekci 2 popisuje návrh rozšíření pro sledování parametrů funkcí a detailně popisuje celou problematiku včetně implementovaného řešení v jazyce C++. Také je zde uveden příklad analyzátoru, který by bez rozšíření bylo velmi obtížné implementovat. V 3. sekci je popsán analyzátor, který kontroluje práci se soubory a demonstruje použití daného rozšíření. Další možná rozšíření a cíle budoucí práce jsou diskutovány v sekci 4.

## 2. Rozšíření frameworku o podporu parametrů funkcí

V této sekci je popsána architektura frameworku ANaConDA a způsob, jakým předává informace analyzátorům. Tyto informace jsou nutné pro vysvětlení komplikací, se kterými se musí rozšíření pro podporu parametrů funkcí vyrovnávat. Dále je podrobně popsáno řešení jeho implementace v jazyce C++ a také je uveden příklad analyzátoru, který toto rozšíření využívá.

### 2.1 Framework ANaConDA

Prostředí ANaConDA (**A**daptable **Na**tive-code **Con**currency-focused **D**ynamic **A**nalysis) poskytuje podporu pro tvorbu dynamických analyzátorů a následnou dynamickou analýzu binárních vícevláknových programů napsaných v jazyce C/C++ [2]. Obstarává sledování běhu programu a má zabudovanou komunikaci s analyzátory. Tím vývojáři ulehčuje práci a umožňuje mu soustředit se pouze na tvorbu analyzátorů.

Schéma celého frameworku je na obrázku 1. Informace o běhu programu získává ANaConDA pomocí nástroje PIN od firmy Intel. Ten provádí dynamickou instrumentaci binárního programu a při každé důležité události, která v programu nastane (přístup do paměti, volání funkce), informuje jádro prostředí ANaConDA [3]. Získané informace jsou předávány analyzátorům, které je využijí ke zhodnocení situace a mohou tak odhalit riziko vzniku chyby. Pro získání ladicích informací a jejich výpis uživateli (*backtrace*) používá ANaConDA knihovnu `libdie`.

Komunikace mezi jádrem prostředí a analyzátory je založena na tzv. zpětných voláních. Jedná se o speciální funkce, které se zavolají, pokud nastane sledovaná událost. V analyzátorech se musí zaregistrovat zpětná volání pro obsluhu požadovaných událostí. Dané funkce také obdrží informace, které se podařilo z vykonávaného programu získat. Například pokud chce analyzátor získávat informace o volání funkcí, musí mít zaregistrované zpětné volání pomocí funkce `THREAD_FunctionExecuted()`, která má dvě přetížené varianty. Jedna se použije v případě, že analyzátor nepotřebuje informaci o návratové hodnotě, v opačném případě se použije druhá varianta. Standardní informací, kterou zpětné volání obdrží, je identifikace vlákna, které provedlo danou akci. V případě volání funkce je další informací ukazatel do paměti, kde se nachází sledovaný argument funkce. Nástroj PIN nepodporuje extrakci informací o datovém typu argumentu. Z toho pramení komplikace při analýze funkcí a jejich parametrů a cílem této práce je co nejvíce zjednodušit tvorbu takových analyzátorů.

ANaConDA poskytuje základní sadu analyzátorů, mezi které patří:

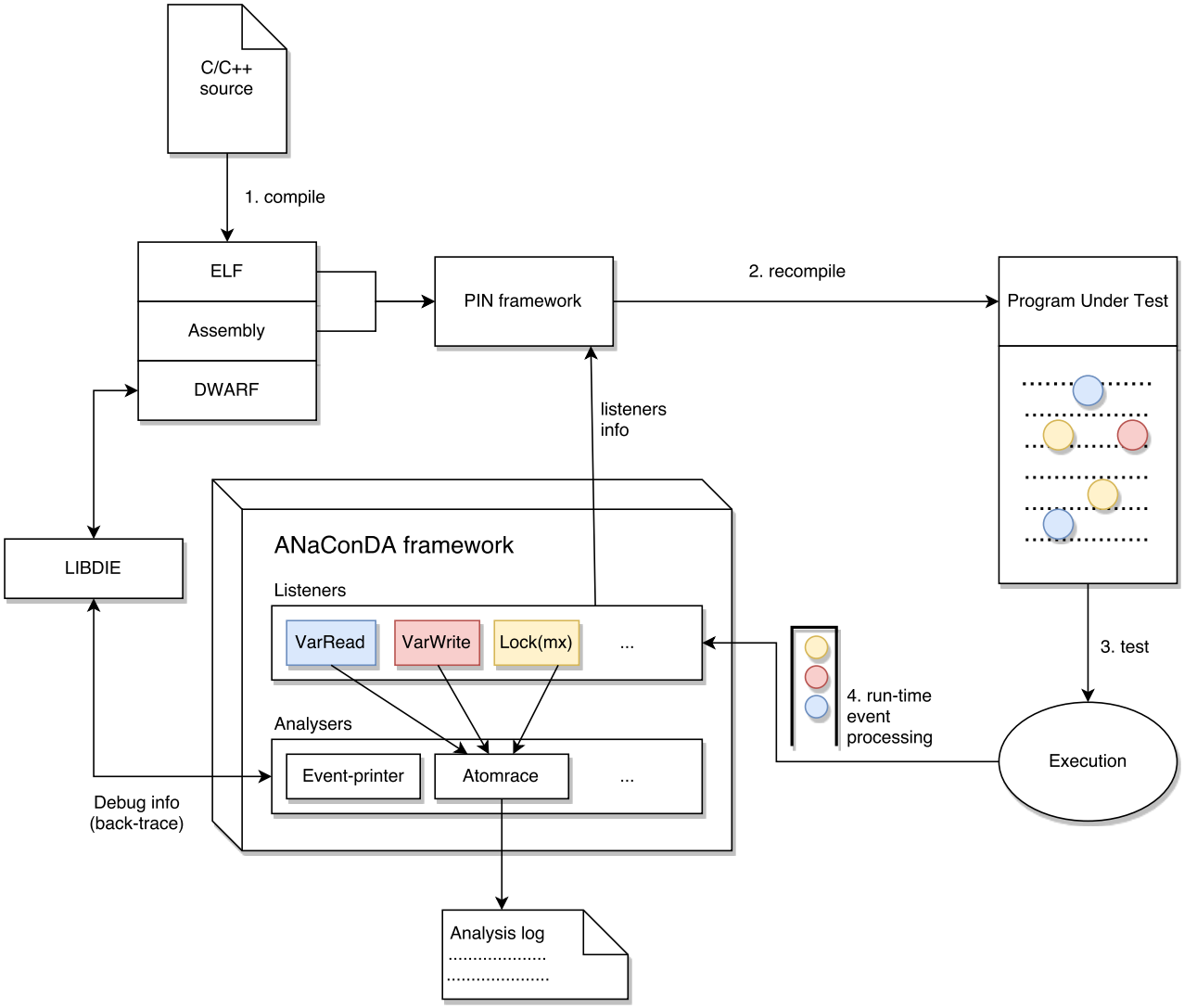
- AtomRace – Implementace algoritmu AtomRace pro detekci souběhu (*data race*).
- Event-printer – Na standardní výstup vypisuje informace o událostech, které nastaly v monitorovaném programu.
- GoodLock – Implementace algoritmu GoodLock pro detekci uváznutí.
- HLDR-detector – Detektor vysokoúrovňového souběhu.
- TX-monitor – Monitoruje použití transakčních pamětí.

### 2.2 Uvedení do problematiky

Situace, ve které se dosud nacházel vývojář analyzátoru, byla následující: Pro analýzu daných argumentů funkcí je často nezbytné získat jejich skutečně zadanou hodnotu příslušného datové typu. Jelikož nástroj ANaConDA není schopen získat informaci o datovém typu argumentu funkce (PIN extrakci těchto informací z ladicích informací nepodporuje), je nutné hodnotu argumentu předávát v obecném formátu, který bude vyhovovat všem datovým typům. Proto každé zpětné volání dostane pouze ukazatel do paměti, kde se nachází data ve formě bajtů. Z tohoto vyplývá nutný předpoklad, že vývojář analyzátoru ví, jakého datového typu jsou daná data.

Motivací pro implementaci rozšíření byly jednak komplikace spojené s konverzí ukazatele do paměti na hodnotu daného argumentu, ale také požadavky na tvorbu generických analyzátorů, které by v okamžiku překladu ještě neměly vědět, nad jakými datovými typy argumentů pracují.

Pokud se jedná o konverzi základních číselných datových typů jako `int` či `float`, situace je poměrně jednoduchá, protože bajty na dané adrese skutečně




**Obrázek 1.** Schéma prostředí ANaConDA. Binární program, který vznikl překladem zdrojového souboru v jazyce C/C++, je podroben dynamické instrumentaci pomocí nástroje PIN. O běhu programu a událostech, které nastaly, je pak informováno jádro prostředí ANaConDA. Tyto informace jsou dále zpracovány v analyzátorech.

odpovídají hodnotě argumentu, který byl zadán při volání funkce. Je tedy nutné pouze přetypování. Komplikace nastávají u objektů typu `std::string` a nejobtížnější je řešení uživatelských datových typů a objektů. Například u objektu typu `std::string` se doposud nepodařilo určit, jestli je vždy zaručeno, že překladač požádá o alokaci místa na haldě a hodnota argumentu funkce tedy představuje pouze referenci na tento objekt. Mohlo by se stát, že hodnota argumentu by byl přímo objekt typu `std::string`. Byla implementována podpora pro první možnost, protože takové chování dosud vykazovaly analyzované programy. Pokud by se při reálném nasazení zjistilo, že nastává i druhá zmíněná situace, bude potřeba testovat programy na cílových architekturách, které takové chování vykazují. Díky tomu budeme schopni určit heuristiku pro identifikaci situace, která v analyzovaném programu nastala. Pak bude možné chování nástroje ANaConDA takové situaci přizpůsobit.

Pro případ kontraktů je také nutné mít analyzátory obecné, které budou umět pracovat nad obecnými funkcemi s parametry různých datových typů. Framework ANaConDA by měl být v budoucnu schopný uspokojit i tyto požadavky. Prozatím byla implementována základní podpora pro sledování funkcí a jejich argumentů na základě informací, které se ANaConDA dozví až za běhu programu.

## 2.3 Návrh rozšíření

Řešení tohoto problému, které bylo implementováno, je zatím pouze prototypové, protože nepokrývá všechny požadavky zmíněné výše a nebylo otestováno na různých systémech. Prvotním cílem bylo vytvořit podporu pro nejjednodušší situace a postupně se zavádí rozšíření pro ty složitější. Nemalou komplikací byla

```
int compare< std::string > (ADDRINT* x, ADDRINT* y)
{
 std::string str1 = UTILS_ConvertToString<std::string >(x);
 std::string str2 = UTILS_ConvertToString<std::string >(y);

 return str1.compare(str2);
}
```

**Zdrojový kód 1.** Porovnání řetězců.

také nutnost seznámit se s již poměrně komplexním prostředím ANaConDA, aby do něj vůbec bylo možné rozšíření zakomponovat.

V rámci této práce byla všechna rozšíření prozatím implementována pro základní datové typy `int`, `float`, `double`, `const char*`, `bool`, `std::string`, `void*` a zahrnují:

- Komparační funkce pro základní datové typy
- Konverzní funkce pro základní datové typy
- Podporu pro zpracování konfiguračních souborů s informacemi o funkcích a parametrech, které se budou monitorovat
- Dva analyzátory pro sledování práce se souborem (viz sekce 3)
- Rozšíření existujícího analyzátoru Event-printer o monitorování funkcí a jejich parametrů na základě konfiguračního souboru
- Prototypový analyzátor pro monitorování funkcí daných konfiguračním souborem a kontrolu, zda jejich argumenty nenabývají zakázaných hodnot

Komparační funkce mají za úkol porovnat buď dva argumenty zadané ukazatelem do paměti a vrátit výsledek tohoto porovnání, nebo porovnat jeden argument daný ukazatelem do paměti a druhý přímo hodnotou. Analyzátory tyto funkce mohou využívat například pro pohodlné porovnání argumentů dvou funkcí. Při implementaci komparátorů bylo využito návrhových vzorů továrna (*factory*) a jedináček (*singleton*), aby bylo možno tyto funkce používat i v situacích, kdy je datový typ argumentu znám až za běhu programu. Návrhový vzor továrna totiž umožní zaregistrovat komparační funkce na základě datového typu, který umí porovnat, a vývojář pak může požádat o přesně takovou komparační funkci, kterou momentálně potřebuje, a to bez znalosti datového typu při překladu. Ukázka takového využití se nachází ve zdrojovém kódu 2. Návrhový vzor jedináček je potřebný z toho důvodu, že k registraci funkcí dochází v jádru nástroje ANaConDA a musí být zaručeno, že existuje pouze jedna instance takové třídy, jinak by vývojář tyto funkce neměl k dispozici v analyzátorech.

Hodnoty, které komparační funkce porovnávají, musí být stejného datového typu. Byla implementována podpora pro základní datové typy, protože u uživatelských datových typů a objektů bude potřeba vyřešit specifikaci relevantních dat pro porovnávání a jejich umístění v paměti. Jedna z možností řešení problému by bylo zadání offsetu pro specifikaci oblasti pro porovnávání.

Rozšíření poskytuje konverzní funkce dvojího typu. První dokáže převést ukazatel do paměti, kde se nachází data ve formě bajtů, na hodnotu datového typu, který daným datům odpovídá. Dané řešení ovšem funguje pouze pro jednoduché datové typy. Konverze je řešena pomocí explicitního přetypování. Druhým typem konverzních funkcí je hojně využívaný převod na typ `std::string`. Často je totiž potřeba uživatele analyzátoru informovat o situacích, které ve sledovaném programu nastaly. Proto existují funkce pro převod ukazatele na `std::string`, pokud se na dané adrese nachází data zmíněných podporovaných datových typů. Je-li konvertovanou hodnotou, která se na dané adrese nachází, objekt typu `std::string`, je také využito explicitního přetypování, ale bere se v úvahu komplikace zmíněná v části 2.2 a na hodnotu argumentu se nahlíží jako na referenci. U ostatních typů dochází k převodu pomocí možností, které nabízí jazyk C++ (využívá se typu `stringstream`).

Zdrojový kód 1 obsahuje ukázku implementace jedné z komparačních funkcí a to pro argumenty typu `std::string`. Datové typy `ADDRINT*` jsou interní datové typy prostředí ANaConDA a jedná se o zmíněné adresy, na kterých se nachází bajty představující hodnoty argumentů s rozsahem podle cílové architektury, na které analýza probíhá. V ukázce je patrné, že s danými bajty je potřeba pracovat jako s objekty, které představují, protože např. u porovnávání objektů je nutné porovnávat jejich rovnost. Funkce `UTILS_ConvertToString()` je zmíněná součást rozšíření, která ukazatel do paměti konvertuje na hodnotu typu `std::string`.

V současné době se také pracuje na podpoře pro tvorbu analyzátorů, které zpracovávají funkce a je-

```
PLUGIN_INIT_FUNCTION()
{
  std::string pth = "functions.conf";
  UTILS_MonitorFunctions(pth, monitoredFunctionEnter);
}

VOID monitoredFunctionEnter(THREADID tid, ADDRINT* arg)
{
  std::string f_name = getCurrentFunctionName(tid);
  FunctionInfo f_info = UTILS_FindFunction(f_name);

  if (f_info.errSet)
  {
    Compare cmp = UTILS_GetIsEqualToValue(f_info.intParamType);
    if (cmp(arg, f_info.errValue))
    {
        CONSOLE("Chyba!");
    }
  }
}
```

**Zdrojový kód 2.** Analyzátor monitorující hodnoty argumentů funkcí na základě konfiguračního souboru.[1]

jich parametry dané konfiguračním souborem. Bylo implementováno rozšíření, které umožňuje načtení informací o sledovaných funkcích z konfiguračního souboru. Také je možné pro všechny tyto funkce registrovat stejné zpětné volání a zpracovávat informace bez znalosti konkrétních funkcí. V konfiguračním souboru se očekává zadání jména funkce, pořadí parametru, který se bude sledovat, a jeho datový typ. Je také možné přidat informaci o hodnotě, které by argument při volání funkce neměl nikdy nabýt a pokud se tak stane, jedná se o chybu. Taková možnost je atraktivní zejména z hlediska kontraktů. Popis implementace tohoto rozšíření je nad rámec článku, ale v následující části 2.4 bude demonstrován způsob, jak je možné ho v analyzátoru využít. Také analyzátor Event-printer, který vypisuje informace o situacích, které nastaly v monitorovaném programu, byl rozšířen o možnost informovat uživatele o volání funkcí, které si sám specifikuje, s využitím tohoto rozšíření.

## 2.4 Využití rozšíření

Zdrojový kód 2 je zjednodušeným přepisem generického analyzátoru pro kontrolu hodnot argumentů, který využívá implementované rozšíření. Jeho cílem je monitorovat volání funkcí daných konfiguračním souborem a hlídat, jestli sledovaný argument nenabývá zakázané hodnoty. Pokud ano, informuje uživatele o chybě.

Funkce `PLUGIN_INIT_FUNCTION()` je úvodní funkcí každého analyzátoru a ve většině případů obsahuje registraci zpětných volání. Bez rozšíření by bylo nutné zaregistrovat zpětné volání pro každou monitorovanou funkci včetně informace o pozici parametru, který se bude sledovat. V prezentovaném analyzátoru je všech potřebných registrací docíleno pomocí funkce `UTILS_MonitorFunctions()`. Je tedy potřeba zadat pouze název konfiguračního souboru a název funkce, která bude zaregistrována jako zpětné volání. Tato funkce se zavolá pokaždé, když se v analyzovaném programu bude vykonávat některá z funkcí specifikovaných v konfiguračním souboru.

Funkce `monitoredFunctionEnter()` představuje dané zpětné volání. Prvním parametrem je identifikace vlákna, které provádí danou akci. Druhým parametrem je ukazatel do paměti, kde se nachází konkrétní hodnota sledovaného argumentu. Protože se ve funkci má provádět porovnávání argumentu a hodnoty uvedené v konfiguračním souboru, je nutné zjistit název funkce, která se aktuálně provádí. K tomu slouží funkce `getCurrentFunctionName()`. Na základě získaného jména je poté možné vyhledat všechny informace o dané funkci, které jsou uloženy ve struktuře `FunctionInfo`. Informace obsahují jméno funkce, pozici sledovaného parametru, jeho datový typ a případně také hodnotu, které nesmí nabýt. Všechny tyto informace byly získány z konfiguračního souboru.

Chybová hodnota argumentu je nepovinná část konfiguračního souboru a informace o tom, zda byla

[1] Za účelem vysvětlení podstaty analyzátoru byl jeho zápis velmi zjednodušen. Reálná implementace obsahuje ošetření různých vyjímečných stavů a detailnější zprávy pro uživatele. Pro korektní fungování analyzátoru se momentálně pracuje na podpoře pro získání jména aktuálně prováděné funkce, jelikož taková funkce dosud nebyla potřebná.

specifikována či nikoli, je uložena v položce `errSet`. Pokud chybová hodnota byla specifikována, je nutné získat komparační funkci. Dochází tedy k volání funkce `UTILS_GetIsEqualToValue()`. Parametrem je datový typ argumentu reprezentovaný pomocí výčtového datového typu (*enum*), který je také součástí struktury `FunctionInfo`. Důležité je, že v okamžiku překladu tento datový typ není znám. To je také největší přínos tohoto rozšíření. Návratovou hodnotou funkce `UTILS_GetIsEqualToValue()` je totiž ukazatel na funkci, která provede srovnání, ať už je argument jakéhokoli podporovaného datového typu. Zadá se pouze ukazatel do paměti, kde se hodnota argumentu nachází, a položka struktury `FunctionInfo`, ve které je uložena chybová hodnota specifikovaná konfiguračním souborem. Pokud bude srovnání vyhodnoceno jako pravdivé, došlo k chybě a bude informován uživatel.

Vytvoření takového analyzátoru bez poskytovaného rozšíření bylo možné pouze v případě, kdy by si vývojář obdobu rozšíření implementoval sám.

## 3. Demonstrace rozšíření

Pro demonstraci analýzy funkcí a jejich parametrů byl implementován analyzátor Open-detector, který kontroluje práci se soubory. Dokáže detekovat pokus použít soubor ke čtení či zápisu, aniž byl dříve otevřen, nebo jeho použití pro tento účel předcházelo zavření souboru.

Tato situace může v paralelních programech nastat poměrně snadno kvůli chybě zvané nesprávné pořadí (*order violation*) [4]. Jedná se o situaci, kdy korektní chování programu závisí na určitém pořadí vykonání jednotlivých akcí, které u paralelních programů může být porušeno. Konkrétně v případu práce se soubory může dojít k tomu, že vlákno uzavře soubor dříve, než do něj jiné stihne zapsat. Analyzátor Open-detector nedetekuje chybu nesprávné pořadí, ale pokud by v důsledku této chyby došlo k nesprávnému použití souboru, pak by toto poznal a uživatele o vzniklé chybě informoval.

K tomu, aby analyzátor dokázal detekovat zmíněné chyby, je nutné, aby viděl volání jednotlivých funkcí pro práci se soubory včetně jejich argumentů a návratových hodnot.

### 3.1 Sledování korektní práce se soubory

Základem je registrování zpětných volání pro funkce pracující se soubory. Analyzátor Open-detector sleduje funkce z hlavičkového souboru jazyka C/C++ `stdio.h`. Tyto funkce pracují s reprezentací souboru pomocí struktury `FILE*`. Byl implementován také podobný analyzátor, který detekuje stejné chyby, ale v případě systémových volání nad souborem. Filozofie analýzy je naprosto totožná, rozdíl je pouze v detekovaných funkcích. Obdobně by se dala analýza rozšířit např. pro kontrolu práce se sockety.

Během implementace analyzátoru bylo nutné vypořádat se s následujícími problémy:

- Názvy funkcí – Knihovní funkce se ve většině případů na binární úrovni volají pod jinými názvy nebo dokonce dochází k volání několika různých funkcí. Proto nelze registrovat zpětné volání například pro funkci `fopen()`. Bylo nutné studovat, jaké funkce různé programy na nejnižší úrovni skutečně volaly, aby bylo možné pro ně registrovat zpětná volání. Situaci ještě komplikovala skutečnost, že pro čtení a zápis do/ze souboru existuje větší množství knihovních funkcí, které volají různé funkce nižší úrovně například i v závislosti na délce čteného či zapisovaného textu.
- Argument a návratová hodnota – Pro to, aby bylo možné určit, zda byl soubor otevřen, příp. jeho jméno, bylo nutné přiřadit k sobě informace získané ze dvou zpětných volání. Pro získání argumentu volané funkce a pro získání návratové hodnoty se totiž registrují dvě různá zpětná volání, takže nikdy není možné znát obě informace najednou.
- Falešná chybová hlášení – Pro vyřešení prvního zmíněného problému bylo nutné registrovat zpětná volání pro funkce, které se volají i v případě, že program pracuje se standardním vstupem či výstupem. Při takových situacích analyzátor hlásil spoustu falešných chyb, a proto bylo třeba implementovat mechanismus, který by dokázal rozlišit, zda se pracuje nad uživatelským souborem či nikoli. Tohoto bylo docíleno pomocí speciálního stavu souboru, viz tabulka 1.

Analyzátor pro svou práci potřebuje uchovávat informace o jednotlivých souborech. Jelikož se pohybujeme v paralelním programu, je nutné, aby přístup k informacím byl výlučný a nedocházelo k chybám na úrovni samotného analyzátoru. K tomuto účelu byla implementována třída, která poskytuje možnost výlučného použití mapy pro ukládání informací. Data o souborech se ukládají pomocí struktury `FileInfo`, která mimo jiné obsahuje název souboru a jeho stav.

Vyhodnocování situace, která nastane při použití souboru ke čtení či zápisu, je založeno právě na stavu použitého souboru. Ten může být trojího typu – stav `OPENED` získá soubor tehdy, když byla detekována funkce pro jeho otevření, stav `CLOSED` naopak při detekci

funkce zavření souboru a NOT_OPENED je výchozí stav souboru. Tabulka 1 znázorňuje přehled situací, které by mohly nastat při detekci funkce pro čtení ze souboru či zápis do něj.

**Tabulka 1.** Vyhodnocení akce čtení či zápisu z/do souboru na základě jeho stavu.

| *Stav* | *Vyhodnocení akce* |
|---|---|
| OPENED | V pořádku |
| CLOSED | Chyba |
| NOT_OPENED | Práce se standardním vstupem/výstupem |

V případě stavu OPENED se jedná o standardní situaci, kdy byl soubor korektně otevřen a je tedy možné jeho použití pro čtení či zápis. Pokud má soubor stav CLOSED, byl před použitím uzavřen a dochází k ohlášení chyby. Stav NOT_OPENED byl použit právě pro omezení falešných chybových hlášení. Pokud je nastaven jako aktuální stav souboru, se kterým se pracuje, znamená to, že se jedná o standardní vstup či výstup.

## 4. Závěr

Byly představeny podpůrné nástroje pro tvorbu analyzátorů a pro sledování funkcí a jejich parametrů pomocí nástroje ANaConDA. Zjistilo se, že se jedná o komplexní problém, který vyžaduje další práci pro plné nasazení do praxe.

V budoucnu se zavede podpora pro složitější datové typy parametrů a rozhraní frameworku ANaConDA se bude rozšiřovat dle potřeb aktuálního výzkumu v oblasti kontraktů.

## Poděkování

Chtěla bych poděkovat Ing. Alešovi Smrčkovi, Ph.D. a Ing. Janu Fiedorovi za jejich pomoc.

## Literatura

[1] Nicholas NETHERCOTE and Julian SEWARD. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN Notices*, pages 89–100, 2007.

[2] Jan FIEDOR and Tomáš VOJNAR. Anaconda: A framework for analysing multi-threaded C/C++ programs on the binary level. In *Proc. of 3rd International Conference on Runtime Verification—RV'12*, volume 7687 of LNCS, pages 35–41, Istanbul, Turkey, 2012.

[3] Jan FIEDOR and Tomáš VOJNAR. Noise-based testing and analysis of multi-threaded C/C++ programs on the binary level. In *Proc. of 10th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging—PADTAD'12*, pages 36–46, Minneapolis, USA, 2012.

[4] Emery D. BERGER, Ting YANG, Tongping LIU, and Gene NOVARK. Grace: safe multithreaded programming for C/C++. In *ACM SIGPLAN Notices*, volume 44, pages 81–96, October 2009.