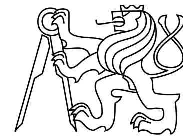
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE

FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁ SKÉ PRÁCE

Název: Validátor zdrojových kód pro vestavné systémy

Student: Jakub Šimek

Vedoucí: Ing. Jan B lohoubek

Studijní program: Informatika

Studijní obor: Teoretická informatika

Katedra: Katedra teoretické informatiky

Platnost zadání: Do konce letního semestru 2016/17

Pokyny pro vypracování

Seznamte se s automatizovanými nástroji pro statickou analýzu kódu.
Zmapujte pokrytí standardu CERT C Coding Standard v p eklada ích a analyzátorech jazyka C, zam te se zejména na p eklada e GCC a LLVM.
Ze standardu vyberte podmnožinu pravidel a doporu ení, které jsou relevantní pro programování vestavných systém .
Na základ provedené analýzy navrhn te nejvhodn jší zp sob implementace rozši itelného nástroje pro statickou analýzu kódu dle zmín ného standardu.
Realizujte omezenou demo verzi nástroje umož ující na práci dále navázat.
Rozsah a zp sob implementace stanovte s p ihlédnutím k provedené analýze a po dohod s vedoucím práce.
Realizované ešení zdokumentujte.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

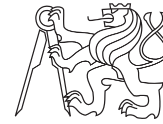
prof. Ing. Pavel Tvrdík, CSc.
d kan

V Praze dne 14. prosince 2015

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra teoretické informatiky



Bakalářská práce

Validátor zdrojových kódů pro vestavné systémy

Jakub Šimek

Vedoucí práce: Ing. Jan Bělohoubek

16. února 2017

Poděkování

Rád bych zde poděkoval vedoucímu této práce, Ing. Janu Bělohoubkovi za jeho cenné rady, ochotu a trpělivost.

Chtěl bych také poděkovat své rodině za podporu.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 16. února 2017 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Jakub Šimek. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Šimek, Jakub. *Validátor zdrojových kódů pro vestavné systémy.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Práce se zabývá validací kódů pro jednoduché vestavné systémy. Nejprve popisuje současné pokrytí relevantní podmnožiny standardu CERT C Secure Coding Standard v dostupných nástrojích. V praktické části pak popisuje implementaci kontrol některých pravidel standardu, která dosud nebyla dostatečně pokryta volně dostupnými nástroji, v novém rozšiřitelném nástroji založeném na knihovně LibTooling překladače Clang. Na zdrojových kódech skutečných projektů je pak demonstrována účinnost nástroje objevením konstrukcí, které s velkou pravděpodobností mohou vést k chybě.

Klíčová slova statická analýza, programovací jazyk C, vestavné systémy, CERT C, bezpečné programování, Clang, LibTooling

Abstract

This bachelor's thesis first describes current coverage of CERT C Secure Coding Standard guidelines relevant to software development for simple embedded systems. The practical part then focuses on implementation of some guidelines of the Standard – which were not previously well covered by freely available tools – in a new extensible tool based on Clang's LibTooling library. The efficacy of the tools is then demonstrated on real projects' source code by finding very likely sources of errors.

Keywords static analysis, C programming language, embedded systems, CERT C, secure coding, Clang, LibTooling

Obsah

Úvod **1**

1 Cíl práce **3**

2 Teoretický úvod **5**
2.1 Statická analýza 5
2.2 CERT C Secure Coding Standard 6

3 Nástroje pro statickou analýzu **11**
3.1 Splint 11
3.2 GCC 13
3.3 Clang 13
3.4 Cppcheck 14
3.5 Rosecheckers 15
3.6 Komerční nástroje 15
3.7 Shrnutí 16

4 Analýza **17**
4.1 Zvolené řešení 17
4.2 LibTooling 17
4.3 Struktura kontroly v Clang Static Analyzer 18
4.4 Limitace Clang Static Analyzer 19

5 Implementace **23**
5.1 Vlastní Frontend Action a Analysis Consumer 23
5.2 Rozšíření o kontrolu maker 23
5.3 Implementované kontroly 25

6 Testování **27**
6.1 Implementační testy 27

6.2 Reálné projekty 27

Závěr **31**

Literatura **33**

A Implementované kontroly **37**

A.1 PRE01-C Používejte závorky uvnitř maker kolem názvů parametrů 37
A.2 PRE02-C Pravá strana by měla být uzávorkovaná 38
A.3 PRE10-C Zabalte více příkazová makra do cyklu do-while . . . 39
A.4 PRE11-C Nezakončujte makra středníkem 41
A.5 DCL30-C Deklarujte objekty se vhodnou dobou platnosti . . . 42
A.6 EXP46-C Nepoužívejte bitové operátory s booleovskými operandy 43
A.7 INT13-C Používejte bitové operátory pouze na operandy bez znaménka 45
A.8 FLP30-C Nepoužívejte floating-point proměnné jako počítadla v cyklech 47
A.9 ARR36-C Neodečítejte od sebe nebo neporovnávejte dva ukazatele, které neukazují na stejné pole 48

B Seznam použitých zkratek **53**

C Obsah přiloženého DVD **55**

Seznam obrázků

4.1 Exploded graph pro ukázku 4.3 (zjednodušeno) 20
4.2 Graf analýzy 4.4 (zkráceno) 22
4.3 Graf analýzy 4.5 (zkráceno) 22

Seznam tabulek

3.1 Pokrytí CERT C SCS komerčními nástroji 16

6.1 Detekce porušení standardu v EM68xx 28
6.2 Detekce porušení standardu v Contiki OS 3.0 28
6.3 Detekce porušení standardu v Atmel Software Framework 3.33 . . 29

Úvod

Vývoj softwaru je velmi nákladný. Velká část těchto prostředků je vynaložena na opravu chyb, které často vznikají nepozorností nebo neznalostí programátorů. Statická analýza kódu může velké množství defektů odhalit velmi rychle, ještě než způsobí problémy po nasazení do produkčního prostředí. Díky tomu je možné chyby odstraňovat rychleji a levněji.

Význam statické analýzy vzrůstá v oblasti vývoje pro vestavné systémy, kde jsou na jedné straně často používány nebezpečné programátorské praktiky, na straně druhé pak selhání softwaru řídícího např. výrobní linku může způsobit nejen materiální škody, ale v krajním případě i škody na zdraví.

Navíc je při vývoji software pro vestavná zařízení velmi často používaný jazyk C namísto modernějších programovacích jazyků, které poskytují bezpečnější abstrakce. Je to dáno jednak omezenými hardwarovými zdroji typického vestavného systému, které zvyšují požadavky na výkonnost aplikace a zamezují použití výpočetně drahých abstrakcí, jednak také nedostupností překladačů pro dané platformy.

Běžnou součástí praktik při vývoji softwaru je dodržení jednotného programovacího stylu (coding style). Tím se rozumí určitý styl zápisu programu (odsazení, psaní závorek, zásady pojmenování proměnných, apod.). Dodržení takového stylu usnadňuje orientaci v kódu a někdy také sníží pravděpodobnost některých chyb. Programovací styly často vznikají spontánně, dodržením stylu předchozího autora kódu, jindy jsou dány formální příručkou.

Důležitou součástí dobrých praktik je však i bezpečné programování (secure coding). To označuje souhrn pravidel a zvyklostí přímo zaměřených na snížení pravděpodobnosti programátorských chyb a bezpečnostních slabin.

Existuje množství standardů pro bezpečné programování. Některé vznikly pro potřeby konkrétní oblasti, jako např. uzavřený standard MISRA C, původně vytvořený pro potřeby automotive, který se následně rozšířil do dalších oblastí, kde probíhá vývoj vestavných systémů. Příkladem obecného otevřeného standardu je pak CERT C Secure Coding Standard.

CERT C Secure Coding Standard si klade za cíl svými pravidly pokrýt

praktiky, které vedou k bezpečnostním chybám v celé šíři vývoje softwaru v jazyce C. Kvůli tomu obsahuje mnoho pravidel, která se věnují např. použití knihoven operačního systému, práci s dynamicky alokovanou pamětí, či práci s vlákny (ze standardu C11), což jsou prostředky, které často nemáme v oblasti vestavných systémů k dispozici. Proto se práce zabývá pouze podmnožinou tohoto standardu.

První kapitola ve stručnosti shrnuje cíle práce. Druhá kapitola obsahuje rychlý úvod do statické analýzy a jejích technik. Dále popisuje SEI CERT C Secure Coding Standard (CERT C SCS), jeho strukturu a význam, a vybírá z něj relevantní kapitoly pro naplnění cílů této práce. Třetí kapitola mapuje pokrytí těchto relevantních kapitol v současně dostupných nástrojích pro statickou analýzu s důrazem na volně dostupné nástroje. Čtvrtá kapitola popisuje vybrané řešení a jeho omezení, v navazující páté kapitole je pak popsaná vlastní implementace. V šesté kapitole jsou shrnuty výsledky testování.

Kapitola 1

Cíl práce

Cílem rešeršní části této práce je zmapovat pokrytí podmnožiny standardu CERT C SCS, která je relevantní pro vývoj jednoduchých vestavných systémů, v dostupných nástrojích pro statickou analýzu.

Cílem praktické části je vytvoření nástroje, který bude provádět analýzu zdrojových kódů v jazyce C a hledat v nich konstrukce, které jsou v rozporu s pravidly a doporučeními standardu CERT C Coding Standard. Nástroj by se měl zaměřit zejména na kontrolu těch pravidel a doporučení, která jsou relevantní pro vývoj softwaru pro vestavná zařízení. Výsledný nástroj by měl být snadno rozšiřitelný o další kontroly.

Kapitola **2**

Teoretický úvod

2.1 Statická analýza

Statická analýza obecně označuje proces automatické analýzy chování programu podle jeho zdrojového kódu bez jeho spuštění. [1]

Techniky statické analýzy původně pochází z kompilátorů, kde jsou využívány jednak pro kontrolu souladu se samotnou specifikací jazyka (např. kompilátor kontrolující soulad datových typů provádí statickou analýzu), jednak pro optimalizace generovaného kódu. Postupně poté začaly vznikat samostatné nástroje pro odhalování programátorských chyb. [1]

Analyzátor může pracovat buď intra-procedurálně, nebo inter-procedurálně. Při intra-procedurální analýze zpracovává každou proceduru (funkci, metodu) samostatně, nezávisle na ostatních. V případě volání jiné procedury pak může buď hodnotu modifikovanou tímto voláním označit za neznámou, nebo může implementovat různé heuristiky, které hodnotu pomohou odhadnout. Při inter-procedurální analýze analyzátor dokáže rozhodovat i o vztazích mezi volanými procedurami. V kontextu jazyků C a C++ je také důležité, zda analyzátor provádějící inter-procedurální analýzu umí rozhodovat i o chování procedur napříč překladovými jednotkami. Intra-procedurální analýza je výrazně jednodušší na implementaci a méně výpočetně náročná. [1]

Mezi základní techniky statické analýzy patří [1][2]:

Control flow analysis – analýza toku kontroly má za úkol rozhodnout, jaké základní bloky mohou vést k dalším základním blokům. Tedy např. jaké větvě výrazů `if-else if-else` jsou možné. [2]

Data flow analysis – analýza toku dat je souhrn technik pro výpočet „faktů" (v praxi především rozsah možných hodnot) v různých průchodech programem. Kromě nástrojů pro validaci kódů se hojně využívá v kompilátorech pro účely optimalizace generovaného kódu. [3]

Abstract interpretation – abstraktní interpretace je technika simulace běhu programu na abstraktním stroji, který nereprezentuje přesně sémantiku konkrétního programovacího jazyka. Podobnost takového abstraktního stroje se strojem skutečným je dána především požadovaným poměrem přesnosti výsledků a náročnosti výpočtu. [1]

Model checking – kontrola na základě modelu je založena na matematických modelech, které jednoznačně popisují zkoumaný systém. Model systému je obvykle automaticky generovaný z metajazyka (anotací). Ověřovač modelů pak ve všech možných stavech systému ověřuje, že systém splňuje všechny předpoklady dané modelem. V případě nevyhovění poskytne protipříklad, který ukazuje, jak systém může dosáhnout nechtěného stavu. [4]

Type checking – kontrola typů ověřuje, zda je program v souladu s typovými pravidly jazyka. V silně typovaných programovacích jazycích tuto analýzu provádí kompilátor. [1]

2.2 CERT C Secure Coding Standard

CERT C SCS byl vyvinut s cílem vytvořit pravidla, která pomohou vývojářům vytvořit bezpečné a spolehlivé systémy. Podle [5] je dodržení těchto pravidel nutnou (ale nikoliv postačující) podmínkou bezpečnosti a spolehlivosti systémů vyvinutých v jazyce C.

Standard se dočkal rozšíření v komerční sféře, jako základní standard pro vývoj všech svých systémů napsaných v jazyce C ho používá firma Cisco, Oracle pak zakomponoval CERT C SCS do svého existujícího standardu [5].

Pracovní verze standardu vzniká na wiki `securecoding.cert.org`. Jednou za čas jsou poté pravidla standardu zkompilována do oficiálního dokumentu, edice z roku 2016 je dostupná v PDF [6].

Standard definuje tzv. *guidelines*, které se dělí na pravidla a doporučení. Doporučení jsou uvedena pouze pro informaci a jejich dodržení, ač samozřejmě doporučeno, není povinné pro to, aby mohlo být o projektu tvrzeno, že je v souladu se standardem [6]. Aby byl bod standardu považován za pravidlo, musí splňovat následující podmínky:

- Porušení pravidla pravděpodobně způsobí defekt, který negativně ovlivní bezpečnost či spolehlivost programu.
- Pravidlo nespoléhá na anotace kódu, nebo domněnky o platformě, kde je kód kompilován nebo provozován.
- Soulad s pravidlem lze detekovat automatickou analýzou (statickou nebo dynamickou), formálními metodami, nebo *manuální inspekcí* (tedy např. čtením kódu)

Doporučení pak splňují následující 2 podmínky:

- Aplikace doporučení pravděpodobně zlepší bezpečnost a spolehlivost systému.
- 1 (nebo více) podmínek požadovaných po pravidlech nemůže být splněna.

(Seznam přeložen z [7])

Pro každé pravidlo či doporučení standard uvádí zhodnocení rizika. To se skládá z 5 částí, každá z nich nabývá hodnot 1 až 3:

Severity (závažnost) – jak závažné jsou následky nedodržení pravidla. Od neobvyklého ukončení (1) po vykonání libovolného kódu útočníkem (3).

Likelihood (pravděpodobnost) – jak velká je pravděpodobnost, že ignorování pravidla povede k bezpečnostní slabině. 1 = nepravděpodobné, 3 = velmi pravděpodobné.

Remediation cost (cena nápravy) – jak náročné je dodržet pravidlo. Od vysoké náročnosti (1) po nízkou náročnost (3). Především se hodnotí, zda je možné dodržení pravidla detekovat a opravit automaticky.

Tyto 3 hodnoty se poté spolu vynásobí, jejich produkt je pak zván Priority (priorita). Ta může dosahovat hodnot 1, 2, 3, 4, 6, 8, 9, 12, 18 a 27. Podle priority je poté přiřazena úroveň (Level) L1-L3. Úroveň je možné použít pro postupné sladění projektu s pravidly standardu, počínaje úrovní L1, která obecně označuje pravidla, jejichž nedodržení pravděpodobně způsobí závažnou bezpečnostní chybu a jejichž oprava není náročná. Podle potřeb pak může organizace pokračovat přes L2 k úrovni L3, která obecně označuje pravidla, jejichž nedodržení s nízkou pravděpodobností povede k málo závažným chybám a jejichž oprava je velmi náročná. [6]

Pravidla a doporučení jsou rozdělena do následujících 17 sekcí, z nichž 3 (POSIX, Microsoft Windows, API) nejsou součástí oficiálního vydání standardu [6]:

Preprocessor (PRE) Kapitola obsahuje 3 pravidla a 14 doporučení k použití maker preprocesoru. Většinou se jedná o doporučení, která předchází neočekávanému výsledku při substituci makra.

Declarations and Initialization (DCL) Obsahuje pravidla, která se týkají deklarací.

Expressions (EXP) Kapitola s velmi širokým záběrem, mezi pravidla patří např. „nespoléhejte na pořadí operací pro vedlejší efekty,“ „nečtěte z neinicializované paměti,“ „nemodifikujte konstantní objekty“ a další. Na počet pravidel a doporučení se jedná o nejbohatší kapitolu.

Integers (INT) Použití celých čísel. Jde především o chyby a bezpečnostní slabiny způsobené neošetřením možností přetečení nebo podtečení celého čísla.

Floating Point (FLP) Použití čísel s plovoucí desetinnou čárkou. Pravidla se týkají převážně neočekávaného nebo nebezpečného chování v souvislosti s omezenou přesností těchto čísel.

Arrays (ARR) V rozporu s názvem obsahuje i pravidla, která se explicitně týkají ukazatelů, které neukazují na pole.

Characters and Strings (STR) Práce s řetězci a proměnnými typu `char` (v kontextech, kde reprezentují znak řetězce).

Memory Management (MEM) Práce s pamětí. Např. „nepřistupujte k uvolněné paměti,“ „uvolňujte pouze paměť alokovanou dynamicky.“

Input Output (FIO) Pravidla a doporučení týkající se funkcí pro práci se soubory.

Environment (ENV) 5 pravidel 3 doporučení týkající se interakce s prostředím. Pravidla se týkají především standardní funkce `getenv()`.

Signals (SIG) Správné zpracování signálů.

Error Handling (ERR) Převážně se týká správného ošetření chyb vzniklých ve funkcích standardní knihovny (např. správné použití `errno`, použití `ferror()` místo `errno` pro chyby při používání `FILE` a souvisejících API).

Application Programming Interfaces (API) Prozatím neobsahuje žádné pravidlo, pouze 11 doporučení.

Concurrency (CON) Pravidla a doporučení ke správnému použití funkcí z hlavičkového souboru threads.h, který je součástí standardu C11. Pravidla k POSIXovým vláknům jsou v sekci POSIX.

Miscellaneous (MSC) Sekce pro pravidla, která se nehodila do jiných kapitol. Mezi pravidla patří např. „Nepoužívejte rand() pro generování pseudonáhodných čísel“ nebo „zajistěte, že řízení nedojde na konec *non-void* funkce.“

POSIX (POS) Obsahuje 17 pravidel a 4 doporučení, která se převážně týkají správného použití funkcí obsažených v rodině standardů POSIX, ale ne v C standardu. Jedná se o nejrozsáhlejší kapitolu, která není součástí oficiálního standardu.

Microsoft Windows (WIN) Prozatím obsahuje 5 doporučení a pouze 1 pravidlo, které se týká správného párování alokačních a de-alokačních funkcí, které obsahuje Windows API. Pravidlo zakazuje použít např. `LocalFree` pro uvolnění paměti alokované pomocí `GlobalAlloc`.

V této práci nás nejvíce zajímají pravidla relevantní pro programování jednoduchých vestavných systémů. Proto jsem pro porovnání existujících nástrojů pro kontrolu souladu se standardem a pro samotnou implementaci uvažoval především pravidla a doporučení z kapitol *Preprocessor*, *Declarations and Initialization*, *Expressions*, *Integers*, *Floating Point*, *Arrays* a některé body z kapitoly *Miscellaneous.*

Kapitola 3

Nástroje pro statickou analýzu

V této kapitole se zaměřuji zejména na popis volně dostupných nástrojů a jejich pokrytí pravidel a doporučení CERT C SCS.

3.1 Splint

Splint je nástroj pro statickou analýzu programů psaných v jazyce C, který se zaměřuje na bezpečnostní chyby [8]. Původní verzi nástroje, zvanou LCLint, vytvořil v roce 1994 David Evans v rámci své diplomové práce [9].

Nástroj sám kontroluje mnoho programátorských chyb, které mohou vést k nečekaným výsledkům nebo bezpečnostním chybám. Důležitou součástí nástroje jsou ale anotace, pomocí kterých může programátor linteru „pomoci“ k důkladnější a přesnější kontrole kódu.

Výpis 3.1 předvádí velmi jednoduchou ukázku použití anotací k označení vstupního parametru `a` a výstupního parametru `b`. Z výpisu 3.2 je pak vidět, že vstupní parametr `a` má být inicializován před voláním funkce `foo`, výstupní parametr `b` pak musí být inicializován uvnitř funkce.

```
 1 static int func(/*@in@*/ int *a, /*@out@*/ int *b)
 2 {
 3     return 0;
 4 }
 5
 6 int main(void)
 7 {
 8     int a, b;
 9     return func(&a, &b);
10 }
```

Výpis 3.1: Ukázka anotovaného kódu pro Splint

```
1  splinttest.c: (in function func)
2  splinttest.c:3:14: Out storage b not defined before return
3    An out parameter or global is not defined before control
4     is transferred.
5  splinttest.c: (in function main)
6  splinttest.c:9:17: Passed storage &a not completely defined:
7     func (&a, ...)
8     Storage derivable from a parameter, return value or
9     global is not defined.
10    Use /*@out@*/ to denote passed or returned storage which
11    need not be defined.
```

Výpis 3.2: Výstup pro ukázku 3.1 (zkráceno)

Na rozdíl od moderních nástrojů pro statickou analýzu neprovádí plný symbolický průchod programem. Výpis 3.3 ukazuje příklad, kde nástroj vypíše varování o použití proměnné, která může být v té době v některých průchodech programem neinicializována, ačkoliv tomu tak ve skutečnosti v jediném možném průchodu programem nemůže být.

```
1 int main(void)
2 {
3     int a;
4     if (1)
5     {
6         a = 32;
7     }
8     return a;
9 }
```

Výpis 3.3: Kód, který chybně vyvolá varování

```
1  splinttest.c:8:12: Variable a used before definition
2      An rvalue is used that may not be initialized to a
           value on some execution path.
```

Výpis 3.4: Výstup pro ukázku 3.3

Ze standardu CERT C Secure Coding Standard pokrývá (bez použití anotací) podle [10] 10 relevantních pravidel a 11 doporučení. Podle mého průzkumu pokrývá ještě 2 doporučení z kapitoly *Preprocessor*, která nejsou v seznamu uvedena. Pomocí anotací by bylo teoreticky možné pokrýt další pravidla standardu, to by ale vyžadovalo snahu a pozornost ze strany uživatele a modifikaci kontrolovaného kódu.

Splint umí zpracovat ANSI C a některá rozšíření z C99. Nejnovější verze nástroje vyšla v roce 2007 a není tedy pravděpodobné, že by byl dále rozvíjen a získal by tak v budoucnosti plnou podporu C99, či dokonce C11.

3.2 GCC

GNU Compiler Collection v rámci varování pokrývá 21 pravidel a doporučení, ne všechna z nich však kompletně [11]. Na rozdíl od většiny ostatních volně dostupných nástrojů kontroluje i jedno doporučení z kapitoly *Preprocessor*, ovšem jedná se jen o doporučení nepoužívat několik otazníků za sebou (dva po sobě následující otazníky značí začátek trigramu).

3.3 Clang

Clang v rámci svých vlastních varování pokrývá 8 relevantních pravidel standardu, z toho 5 pouze částečně [12].

3.3.1 Clang Static Analyzer

Clang Static Analyzer (CSA) je součástí distribuce překladače Clang. Ze standardu CERT C částečně nebo zcela pokrývá 9 pravidel [12][13], z toho 3 pravidla (například nekopírovat objekt FILE) nejsou relevantní pro vývoj softwaru pro jednoduchá vestavná zařízení. Struktura kontrol a princip funkce jsou podrobně popsány v kapitole 4.

3.3.2 Clang Tidy

Clang Tidy je nástroj využívající libclang, který provádí kontroly na abstraktním syntaktickém stromu a na úrovni preprocesoru. Kromě kontrol odhalujících „skutečné“ chyby má i několik kontrol zaměřených čistě na styl.

Nástroj obsahuje modul nazvaný CERT, ale většina v něm obsažených kontrol je specifická pro jazyk C++. Z relevantních kapitol CERT-C standardu pokrývá kromě 5 pravidel také první 2 doporučení z kapitoly *Preprocessor*, ta se týkají uzávorkování maker.

Struktura checkeru pro Clang Tidy je vidět na ukázce 3.5.

```
 1 class ExampleCheck : public ClangTidyCheck {
 2 public:
 3   ExampleCheck(StringRef Name, ClangTidyContext *Context)
 4       : ClangTidyCheck(Name, Context) {}
 5   void registerMatchers(
 6     ast_matchers::MatchFinder *Finder) override;
 7   void check(
 8     const ast_matchers::MatchFinder::MatchResult &Result) override;
 9   void registerPPCallbacks(CompilerInstance &Compiler) override;
10 };
```

Výpis 3.5: Struktura Clang Tidy checkeru

Pro kontroly direktiv preprocesoru slouží metoda `registerPPCallbacks`, ve které si checker může zaregistrovat zpětná volání preprocesoru. Samotná

kontrola těchto direktiv poté probíhá přímo uvnitř objektu typu PPCallbacks, který si checker zaregistruje.

Clang Tidy používá pro kontroly na úrovni AST třídy MatchFinder a Matcher z Clangu. Checkery využívající Abstract syntax tree – abstraktní syntaktický strom (AST) si v metodě registerMatchers zaregistrují alespoň 1 Matcher. Všechny uzly, které odpovídají alespoň jednomu Matcheru jsou poté předávány jako parametr metody check, ve které probíhá samotná kontrola.

Matcher používá zajímavé syntaktické konstrukce, které jsou velmi podobné funkcionálním programovacím jazykům. Na ukázce 3.6 je příklad, který najde cyklus `for`, jehož inicializační část deklaruje právě jednu proměnnou, a tato je inicializovaná na nulu. Poté provede `bind`, díky čemuž se cykly odpovídající těmto podmínkám uloží do výsledku (`MatchResult`).

```
1  StatementMatcher LoopMatcher =
2    forStmt(hasLoopInit(declStmt(
3      hasSingleDecl(varDecl(
4        hasInitializer(
5          integerLiteral(
6            equals(0)
7          )
8        )
9      ))
10   ))).bind("forLoop");
```

Výpis 3.6: Příklad AST Matcher, převzato z [14]

3.4 Cppcheck

Cppcheck je primárně určen pro kontrolu kódu v jazyce C++, tomu odpovídá i množství kontrol specifických pro tento jazyk. Jedná se například o práci s šablonami či s kolekcemi ze standardní knihovny STL. Plně však podporuje i validaci zdrojových kódů napsaných v jazyce C. Ze standardu CERT C pokrývá celkem 12 pravidel, převážně ze sekce *Expressions* [15].

Hlavním cílem jeho vývojářů je vytvořit nástroj, který bude detekovat pouze skutečné chyby, tedy nebude mít pokud možno žádná falešná pozitiva [16]. To omezuje možnost přidání některých kontrol, které nemohou ze své podstaty mít 100% přesnost.

Kromě kontrol na úrovni AST umí také zpracovávat tok hodnot (value flow). Podobně jako CSA ve value flow analýze nepodporuje čísla s plovoucí desetinnou čárkou.

CppCheck poskytuje výstup svých vnitřních dat pomocí přepínače `--dump`. Výsledkem je xml soubor, který obsahuje AST, seznam tokenů, databázi symbolů a tok hodnot. Tento soubor je poté možné zpracovat knihovnou cppcheckdata napsanou v jazyce Python. Je tak možné si napsat vlastní kontroly. Nevýhodou z pohledu této práce je nemožnost použít takový výstup pro kon-

trolu maker, ta totiž nejsou jeho součástí, resp. je výstup generován až po provedení kompletního preprocessingu.

3.5 Rosecheckers

Nástroj vyvinutý Davidem Svobodou v rámci organizace CERT. Je postavený na kompilátorové infrastruktuře ROSE, která byla vyvinuta v Lawrence Livermore National Laboratory. Cílem projektu je vytvořit knihovnu a sadu nástrojů k rychlé a efektivní tvorbě nástrojů pro analýzu a transformace zdrojového kódu. Tomu je uzpůsobena vnitřní reprezentace kódu, která zachovává co možná nejvíce doprovodných informací o každém uzlu abstraktního syntaktického stromu [17]. ROSE Compiler je dostupný pro Linux a Mac OS, port pro Windows nebyl dokončen [18].

Doporučený způsob použití Rosecheckers je v podobě obrazu pro virtualizační nástroje (např. VMWare) s názvem Rosebud [19]. Jedná se o 10 GB velký obraz založený na operačním sytému Debian, ve kterém je nainstalován kompilátor ROSE a nástroj Rosecheckers. Provoz jiným způsobem není oficiálně podporován. Není tak příliš snadné použít tento nástroj v rámci systémů průběžné integrace.

Jako jediný ze zkoumaných volně dostupných nástrojů je přímo zaměřený na kontrolu souladu se standardem CERT C. Ze všech volně dostupných nástrojů dokáže kontrolovat soulad s největším počtem pravidel, celkem 57 jich implementuje kompletně a dalších 45 částečně [20]. Ani Rosecheckers ale nekontroluje žádná pravidla a doporučení ze sekce *Preprocessor*.

V době psaní této práce pochází nejnovější změna v repozitáři Rosecheckers z roku 2013 [21] a zdá se tak, že vývoj nástroje nepokračuje.

3.6 Komerční nástroje

Existuje celá řada komerčních nástrojů pro statickou analýzu programů v jazyce C. Většina z nich se zaměřuje hned na několik standardů, převažuje mezi nimi důraz na standard z oblasti automotive MISRA C [22].

Tabulka 3.1 uvádí některé komerční nástroje a počet pravidel a doporučení z relevantních kapitol standardu, která alespoň částečně pokrývají.

Bohužel u mnoha těchto nástrojů nejsou volně k dispozici jiné, než marketingové informace, proto se tady jimi nezabývám podrobněji.

Tabulka 3.1: Pokrytí CERT C SCS komerčními nástroji

Název	Pravidla	Doporučení
CodeSonar	23	43
Coverity	13	9
Eclair	12	39
Klocwork	6	9
LDRA Tool Suite	42	82
Parasoft C/C++Test	31	58
PRQA QA-C	34	63
Polyspace Bug Finder	32	26

3.7 Shrnutí

Žádný ze zkoumaných volně dostupných nástrojů nekontroluje uspokojivě pravidla a doporučení týkající se využití preprocesoru. Při vývoji softwaru pro vestavná zařízení jsou však často používána makra místo bezpečnějších konstrukcí, např. namísto konstantních proměnných nebo inline či statických funkcí a jsou tak potenciálním zdrojem velkého množství chyb.

Kapitola 4

Analýza

4.1 Zvolené řešení

Jako možnost realizace se nabízí rozšíření existujícího nástroje pro statickou analýzu (např. CppCheck), případně využití výstupu z nějakého takového nástroje. V obou případech jsem se ale setkal s problémem, jak realizovat kontrolu pravidel a doporučení týkající z kapitoly *Preprocessor*, kterou považuji za velmi důležitou. Většina nástrojů pro statickou analýzu je totiž od základu navržena tak, že veškeré kontroly probíhají až po preprocessingu a parsingu zdrojového kódu, tedy až z abstraktního syntaktického stromu, případně podobné vnitřní reprezentace.

Jako nejschůdnější řešení jsem zvolil možnost rozšíření kompilátoru. To mi dalo nejširší možnosti, jak a kdy provádět kontrolu souladu s pravidly standardu.

Zvolené řešení využívá LibTooling, díky kterému je možné celý překladač Clang/LLVM použít jako knihovnu, a to včetně jeho statického analyzátoru, který se stal jádrem výsledného nástroje.

4.2 LibTooling

LibTooling [23] je knihovna, která umožňuje tvorbu samostatných nástrojů (tedy ne jen pluginů), které využívají infrastrukturu kompilátoru Clang.

Základem nástroje, který využívá LibTooling, je 1 nebo více tříd dědících ze třídy `FrontendAction`. Instance těchto tříd je poté předána metodě `run` třídy `ClangTool`, která každou překladovou jednotku zpracuje překladačem Clang a spustí na ni danou `FrontEndAction`.

```
1 CommonOptionsParser op(argc, argv, MyToolCategory);
2
3 ClangTool Tool(op.getCompilations(),
4                op.getSourcePathList());
5 Tool.run(
6   newFrontendActionFactory<CertCFGFrontendAction>().get());
```

Výpis 4.1: Základ použití LibTooling

Seznam překladových jednotek a další nastavení získá `ClangTool` díky třídě `CommonOptionsParser`, která zpracuje seznam předaných souborů a parametry, které by měly být společné všem nástrojům, které používají LibTooling.

4.3 Struktura kontroly v Clang Static Analyzer

Každá třída implementující kontrolu (checker) je potomkem třídy BaseChecker, která implementuje pouze základní vlastnosti. Vlastní metody volané při kontrole pak checker získává kompozicí.

V ukázce 4.2 je deklarace třídy, která využívá kontrolu uniklých symbolů, začátku funkce a výrazů obsahujících binární operátor.

```
1  class ExampleChecker : public clang::ento::Checker<
2      clang::ento::check::DeadSymbols,
3      clang::ento::check::BeginFunction,
4      clang::ento::check::PreStmt<clang::BinaryOperator>> {
5
6  public:
7      ExampleChecker();
8
9      void checkDeadSymbols(clang::ento::SymbolReaper &SymReaper,
10         clang::ento::CheckerContext &C) const;
11     void checkBeginFunction(
12         clang::ento::CheckerContext &Ctx) const;
13     void checkEndFunction(
14         clang::ento::CheckerContext &Ctx) const;
15 };
```

Výpis 4.2: Příklad třídy checkeru v CSA

Checker v CSA může kompozicí získat tyto metody:

checkPreStmt a checkPostStmt jsou volány před, respektive po vyhodnocení výrazu jádrem analyzátoru. Dají se specializovat pro různé výrazy. Tyto metody nejsou volány pro výrazy řídicí tok programu, např. `if`, nebo podmínku ve výrazu `for` – pro ty je volána metoda `checkBranchCondition`.

checkPreCall a checkPostCall umožňují kontrolovat událost „call.“ Ta v jazyce C symbolizuje pouze volání funkce, jinak je ale pro potřeby C++

a Objective C obecnější.

checkBranchCondition je volána pro podmíněné výrazy. Jedná se o pre-visit, tedy před zpracováním analyzátory; post-visit verze neexistuje.

checkLocation je volána pokaždé, když je přistupováno k hodnotě skrze ukazatel, tedy i pro přístupy do pole.

checkBind umožňuje kontrolovat operace přiřazení.

checkDeadSymbols je volána když symbol přestane být platný, např. dojde k dealokaci proměnné.

checkBeginFunction a checkEndFunction jsou volané na začátku, resp. na konci analýzy funkce a to bez ohledu, na to, zda byla funkce inline.

checkEndAnalysis je volána, když je explodedGraph plně expandovaný.

checkEndOfTranslationUnit je volaná na konci překladové jednotky.

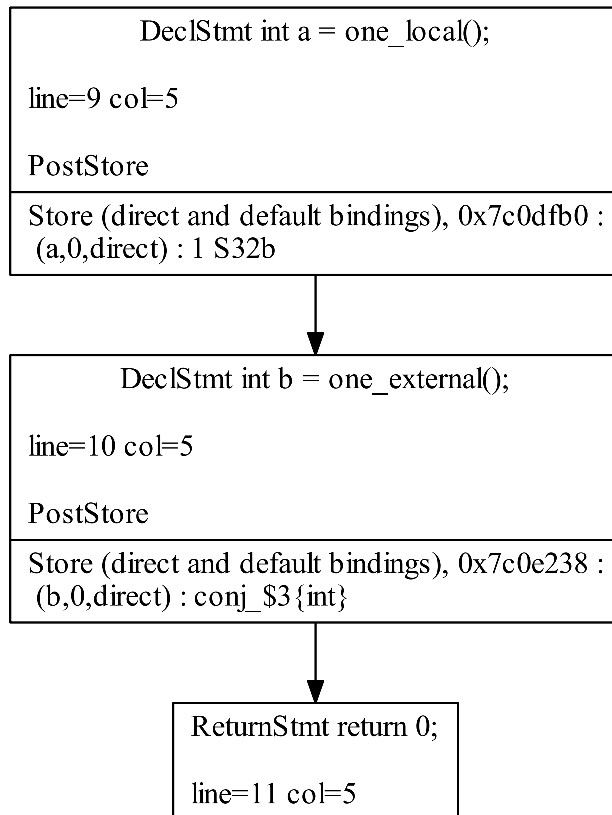
checkASTCodeBody slouží checkerům, které pro kontrolu využívají pouze AST.

4.4 Limitace Clang Static Analyzer

V současnosti CSA nepodporuje inter-procedurální analýzu mezi překladovými jednotkami (translation units). To znamená, že každé volání funkce z jiné jednotky se stává „černou skříňkou“ a například v momentě, kdy je proměnná předána takové funkci odkazem, nebo ukazatelem, ztrácí analyzátor schopnost rozhodovat, jakých může nabývat hodnot. Tuto limitaci ilustruje ukázka 4.3:

```
1  #include "ipcb.h"
2  int one_local()
3  {
4      return 1;
5  }
6
7  int main(int argc, char **argv)
8  {
9      int a = one_local();
10     // Funkce one_external je stejna jako one_local,
11     // pouze je definovana v jine prekladove jednotce
12     int b = one_external();
13     return 0;
14 }
```

Výpis 4.3: Absence inter-procedurální analýzy



Obrázek 4.1: Exploded graph pro ukázku 4.3 (zjednodušeno)

Na grafu 4.1 vidíme, že v případě volání `one_local` dokáže analyzátor rozhodnout, že do proměnné `a` bude uložena hodnota 1. V případě volání úplně stejné funkce `one_external`, která je definována v jiné překladové jednotce, není analyzátor schopný rozhodnout o hodnotě proměnné `b`.

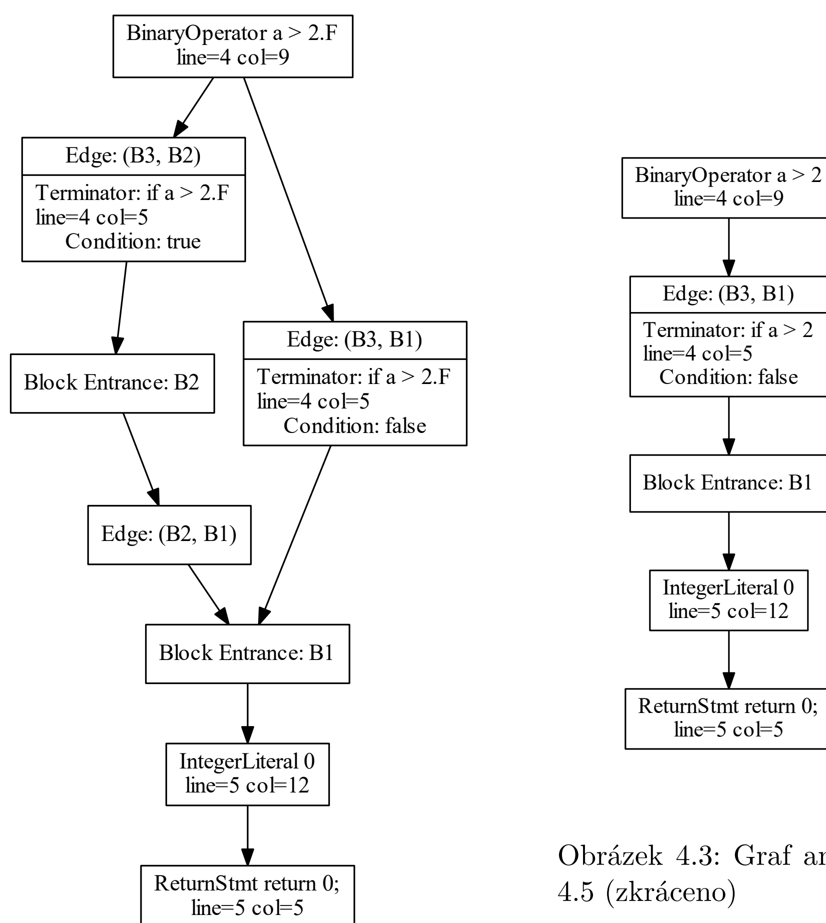
Zároveň analyzátor nepodporuje plně simulaci práce s čísly s plovoucí desetinnou čárkou (floating point). Toto omezení vede jednak k nemožnosti implementovat kontrolu některých pravidel standardu, jednak může vést k menší přesnosti dalších kontrol, protože jsou expandovány i průchody programem, které reálně nejsou možné. Ukázka 4.4 předvádí jednoduchý příklad, kde analyzátor není schopný určit, že operace v podmínce vždy skončí hodnotou false a tak zbytečně expanduje graf, viz 4.2. Pro srovnání ukázka 4.5 a výsledný graf 4.3, kde je expression engine schopný rozhodnout, že podmínka nebude splněna.

```
1 int main(void)
2 {
3     float a = 1.f;
4     if (a > 2.f);
5     return 0;
6 }
```

Výpis 4.4: if s desetinným číslem v podmínce, o kterém CSA nedokáže rozhodnout

```
1 int main(void)
2 {
3     int a = 1;
4     if (a > 2);
5     return 0;
6 }
```

Výpis 4.5: if s celým číslem v podmínce, o kterém CSA dokáže rozhodnout



Obrázek 4.2: Graf analýzy 4.4 (zkráceno)

Obrázek 4.3: Graf analýzy 4.5 (zkráceno)

Kapitola 5

Implementace

5.1 Vlastní Frontend Action a Analysis Consumer

Základem nástroje je třída `CertCFGFrontendAction`. Ta vytvoří instanci třídy `CertAnalysisConsumer`, která se stará o spouštění jednotlivých kontrol.

Třída `CertAnalysisConsumer` je forkem třídy `ClangAnalysisConsumer`. Bohužel nebylo možné čistě použít dědičnost, protože třída je deklarována v anonymním jmenném prostoru uvnitř `.cpp` souboru.

Oproti původní verzi obsahuje několik změn, především používá vlastní manažer a registr kontrol `CertCheckerManager`, resp. `CertCheckerRegistry`, díky čemuž umožňuje checkerům kontrolovat makra.

5.2 Rozšíření o kontrolu maker

Pro přidání podpory kontroly maker jsem vytvořil jednoduchou třídu Preprocessor, která umožňuje checkerům v metodě `registerPPCallbacks` zaregistrovat vlastní zpětná volání preprocessoru, což jim umožňuje kontrolovat správnost maker.

```
1  namespace cert {
2  class Preprocessor {
3  public:
4      template <typename CHECKER>
5      static void _registerPPCallbacks(void *checker,
6              clang::Preprocessor *PP) {
7          ((CHECKER *)checker)->registerPPCallbacks(PP);
8      }
9
10     template <typename CHECKER>
11     static void _register(CHECKER *checker,
12             clang::ento::CheckerManager &mgr) {
13         CertCheckerManager *ccm = (CertCheckerManager*)&mgr;
14         ccm->_registerForPPCallbacks(
15             CertCheckerManager::RegisterPPCallbacksFunc(
16                 checker, _registerPPCallbacks<CHECKER>));
17     }
18 };
19 }
```

Výpis 5.1: Třída Preprocessor

Metoda `_register` nemá za parametr původní CheckerManager (jako je tomu u vestavěných tříd v CSA), ale jeho specializaci, kterou jsem vytvořil. Poděděná třída má navíc pouze členskou proměnnou udržující ukazatel na preprocesor a novou metodu, která je volána, když je registrována nová kontrola direktiv preprocesoru.

```
1  class CertCheckerManager :
2      public clang::ento::CheckerManager
3  {
4      clang::Preprocessor* PP;
5  public:
6      CertCheckerManager(const clang::LangOptions &langOpts,
7                         clang::AnalyzerOptionsRef AOptions,
8                         clang::Preprocessor& PP)
9          : CheckerManager(langOpts, std::move(AOptions)),
10           PP(&PP)
11         {}
12
13     typedef clang::ento::CheckerFn<void(clang::Preprocessor*)>
14             RegisterPPCallbacksFunc;
15
16     void _registerForPPCallbacks(RegisterPPCallbacksFunc func)
17     {
18         func(PP);
19     }
20 };
```

Výpis 5.2: Třída CertCheckerManager

5.3 Implementované kontroly

Při výběru kontrol k implementaci jsem jako hlavní faktory zvažoval pokrytí daného pravidla v ostatních volně dostupných nástrojích, *úroveň* pravidla a složitost implementace. Podrobný popis všech implementovaných kontrol je v příloze A.

Cílem nebylo pokrýt všechna pravidla standardu, ale spíše všechny typy pravidel, která jsou relevantní pro vývoj softwaru pro jednoduchá vestavná zařízení.

Všechny checkery, které nevyžadují kontrolu direktiv preprocesoru jsou napsané tak, aby je bylo možno bez zásadních úprav buď přímo přidat do CSA, nebo z nich vytvořit plugin. Checkery, které implementují `PPCCallbacks` by bylo možné přidat do nástroje Clang Tidy; pouze by bylo potřeba upravit způsob hlášení chyb.

Kapitola **6**

Testování

6.1 Implementační testy

V průběhu vývoje jsem pro každou implementovanou kontrolu vytvořil testovací soubor, který obsahuje vyhovující i nevyhovující útržky kódu. Většina útržků byla převzata nebo inspirována přímo z CERT C Secure Coding wiki. Pro závěrečné otestování jsem tyto útržky spojil do jednoho souboru (jeho celý zdrojový kód je na přiloženém médiu ve složce `test`).

Z pohledu těchto ukázek dosahují všechny checkery, s výjimkou Dcl30-C, 100% úspěšnost. To ukazuje spíše nedostatečnost oficiálních příkladů (i s mými dodatky) jako testovací sady, jak se poté ukázalo při testování na reálných projektech, kde se začaly objevovat falešné detekce.

Poté jsem ještě vyzkoušel checkery na testovací sadě CERT Secure Validation Suite [24]. V rámci této práce jsem se však nezaměřil na checkery, které by pokrývaly pravidla z této sady. Podle očekávání dopadly testy ze složky `addrescape`, které přesně pokrývají 3 ukázky z pravidla Dcl30-C (viz A.5). Implementovaný checker zde správně detekoval v jednom ze tří případů. Dále správně detekoval několik stejných případů v dalších testech, které nebyly zaměřeny přímo na toto pravidlo.

6.2 Reálné projekty

Pro otestování implementovaných checkerů jsme zvolili několik sad zdrojových kódů dodávané renomovanými dodavateli embedded HW a SW z komerční i akademické sféry.

Zdrojové kódy testované v rámci této kapitoly se nachází na přiloženém médiu.

Tabulka 6.1: Detekce porušení standardu v EM68xx

Doporučení	Počet detekcí
Pre01-C	7
Pre02-C	2
Pre10-C	2

Tabulka 6.2: Detekce porušení standardu v Contiki OS 3.0

Doporučení	Počet detekcí
Dcl30-C	2
Pre01-C	408
Pre02-C	1282
Pre10-C	19
Pre11-C	42

6.2.1 EM68xx as I2C Master Device

Příklad kódu od EM Microelectronic, který ukazuje, jak použít mikrokontrolér EM6812 jako I2C master [25].

Většina porušení doporučení Pre01-C a Pre02-C nemá závažné důsledky.

Zajímavý problém má makro `CheckComingAckI2C` (na výpisu 6.1).

```
1 #define CheckComingAckI2C \
2   if ((RegInI2C & SdaI2C) == SdaI2C) { \
3          (VarStatusRegI2C |= ErrorStatusI2C); \
4          (VarStatusRegI2C &= ~AckStatusI2C); \
5     return; \
6   }
```

Výpis 6.1: Makro CheckComingAckI2C z EM68xx

Pokud by bylo toto makro použito uvnitř konstrukce `if-else` bez použití složených závorek, vedlo by to k nečekané logice programu (viz 6.2).

```
1 if (podminka)
2     CheckComingAckI2C
3 else // else ve skutecnosti patri k~if z~makra
4     udelejKdyzJePodminkaFalse();
```

Výpis 6.2: Potenciální chyba při použití makra CheckComingAckI2C

6.2.2 Contiki

Otevřený operační systém pro internet věcí [26]. Testována verze 3.0.

Obě hlášení porušení pravidla Dcl30-C byly falešnou detekcí.

Porušení doporučení Pre01-C je v tomto projektu opravdu mnoho a velká část z nich může s velkou pravděpodobností vést k neočekávaným výsledkům.

Tabulka 6.3: Detekce porušení standardu v Atmel Software Framework 3.33

Doporučení	Počet detekcí
Pre01-C	178
Pre02-C	52
Pre11-C	17

Zajímavé je, že v mnoha makrech jsou smíchány parametry, které jsou správně uzávorkované spolu s neuzávorkovanými. Jeden příklad je ve výpisu 6.3.

```
1 // parametr shift uzavorkovany spravne, size nikoliv
2 #define MAKE_MASK(shift,size) (((1 << size) - 1) << (shift))
```

Výpis 6.3: Makro MAKE_MASK z Contiki OS

Viníkem velkého množství porušení Pre02-C je soubor `regs.h`, který definuje velké množství maker pro přístup k hodnotě jednotlivých registrů (příklad 6.4). Takové makro sice porušuje normu, ale skutečné nebezpečí je minimální. Na druhou stranu důsledné dodržení doporučení by v tomto případě nemělo negativní vliv na využití makra. Z tohoto souboru pochází celkem 1132 varování ohledně porušení doporučení Pre02-C.

```
1 #define SCS_SCR *((volatile uint32_t *)0xE000ED10u)
```

Výpis 6.4: Makro SCS_SCR z Contiki OS

V rámci varování o porušení Pre10-C se vedle legitimních problémů ukazuje i falešná detekce makra uvedeného ve výpisu 6.5. Je jasné, že takové makro nemůže být obaleno v cyklu `do-while`. Tato falešná detekce je dána příliš zjednodušenou detekcí toho, jaké makro je více příkazové (viz sekce A.3).

```
1 #define _IO(x) \
2     union { struct { uint32_t x##0; uint32_t x##1; }; \
3         struct { _REP(x, 1) }; \
4         struct GPIO_##x { _REP(GPIO, 1) } x; };
```

Výpis 6.5: Makro _IO z Contiki OS

6.2.3 Atmel Software Framework

Atmel Software Framework je sbírka software pro mikrokontrolery Atmel, která zjednodušuje vývoj pro tato zařízení [27]. Testován byl adresář avr32/drivers z verze 3.33.

Celkem 82 porušení doporučení Pre01-C se objevuje v makrech ze souboru `canif.h`, který definuje několik desítek maker pro přístup k poli. Velké množství těchto maker vypadá podobně jako 6.6, kde je sice doporučení formálně porušeno, ale toto porušení nemůže způsobit chybu. Na druhou stranu by přísné dodržení doporučení standardu nenarušilo funkčnost kódu.

```
1 #define CANIF_get_interrupt_status(ch) \
2   ( AVR32_CANIF.channel[ch].canisr)
```

Výpis 6.6: Makro CANIF_get_interrupt_status z ASF

Problematické je už ale např. makro `CANIF_get_mobctrl` (6.7). Pokud by zde byl za parametr `mob` dosazen složitější výraz, mohlo by dojít k přístupu na jiný index, než programátor zamýšlel.

```
1 #define CANIF_get_mobctrl(ch,mob) \
2   (((unsigned volatile long*)& \
3   (AVR32_CANIF.channel[ch].mobctrl))[mob*3])
```

Výpis 6.7: Makro CANIF_get_mobctrl z ASF

Z porušení Pre02-C se opakuje zejména definice chybových hodnot, které jsou definovány jako záporné celé číslo. Toto číslo by mělo být uzávorkováno.

```
1 #define TWI_INVALID_ARGUMENT      -1
2 #define TWI_ARBITRATION_LOST      -2
```

Výpis 6.8: Typický příklad porušení Pre02-C v ASF

Problém může nastat v případě, kdy dojde k překlepu a vynechání porovnávacího operátoru. V takovém případě je znaménko mínus vyhodnoceno jako binární a výraz dostává zcela jiný význam. Kdyby byla pravá strana makra uzávorkovaná v souladu s doporučením, zahlásil by kompilátor syntaktickou chybu (viz kapitola A.2 a výpis A.3).

Závěr

V rámci implementační části práce jsem vytvořil nástroj postavený na knihovně LibTooling překladače Clang. V rámci tohoto nástroje jsem implementoval 4 pravidla a 5 doporučení standardu. Každá z prvních 6 kapitol standardu, které jsem vyhodnotil jako nejrelevantnější pro cíle této práce, je zastoupena alespoň jednou kontrolou.

Z rešerše pokrytí standardu v nástrojích pro statickou analýzu se ukázalo, že hlavní slabinou těchto nástrojů je slabé pokrytí doporučení z oblasti preprocesoru. Proto jsem se v implementaci na tuto část zaměřil a z kapitoly *Preprocessor* jsem implementoval 4 kontroly.

V průběhu testování pak nástroj odhalil v reálně používaných kódech pro vestavná zařízení velké množství porušení standardu. Většinou se jednalo o chyby z kapitoly *Preprocessor*. V některých případech špatné použití závorek v makru vedlo k vysoké pravděpodobnosti chyby.

Výsledný nástroj je možné snadno rozšířit o další kontroly. Případně je možné s minimem úsilí změnit směřování vývoje díky tomu, že všechny kontroly, které nekontrolují příkazy preprocesoru, je možné použít v pluginu pro Clang Static Analyzer.

Literatura

[1] Brumley, D.: Static Analysis. In *Encyclopedia of Cryptography and Security*, editace H. C. A. van Tilborg; S. Jajodia, Boston, MA: Springer US, 2011, ISBN 978-1-4419-5906-5, s. 1254–1256.

[2] Nielson, F.; Nielson, H. R.; Hankin, C.: *Principles of Program Analysis*. Springer Berlin Heidelberg, první vydání, 1999, ISBN 978-3-642-08474-4, XXI, 452 s.

[3] Aho, A. V.; Lam, M. S.; Sethi, R.; aj.: *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., druhé vydání, 2006, ISBN 0321486811.

[4] Baier, C.; Katoen, J.-P.: *Principles of model checking*. The MIT Press, první vydání, ISBN 978-0-262-02649-9.

[5] Seacord, R. C.: *The CERT C coding standard: 98 rules for developing safe, reliable, and secure systems*. Upper Saddle River, NJ: Addison-Wesley, druhé vydání, 2014, ISBN 03-219-8404-8, 517 s.

[6] *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition)*. Carnegie Mellon University, 528 s.

[7] Rules vs. Recommendations. 2016-10-01, [cit. 2017-01-28]. Dostupné z: `https://www.securecoding.cert.org/confluence/display/seccode/Rules+vs.+Recommendations`

[8] Splint Home Page. [cit. 2017-01-10]. Dostupné z: `http://www.splint.org/`

[9] Evans, D.: *Using Specifications to Check Source Code*. Diplomová práce, Massachusetts Institute of Technology, 1994.

[10] Splint - SEI CERT C Coding Standard. 2017-01-05, [cit. 2017-01-10]. Dostupné z: https://www.securecoding.cert.org/confluence/display/c/Splint

[11] GCC - CERT C Coding Standard. 2017-01-05, [cit. 2017-01-06]. Dostupné z: https://www.securecoding.cert.org/confluence/display/c/GCC

[12] Clang - CERT C Coding Standard. 2017-01-05, [cit. 2017-01-05]. Dostupné z: https://www.securecoding.cert.org/confluence/display/c/Clang

[13] Clang - Available Checkers. [cit. 2017-01-05]. Dostupné z: http://clang-analyzer.llvm.org/available_checks.html

[14] Tutorial for building tools using LibTooling and LibASTMatchers – Clang 5 documentation. [cit. 2017-02-12]. Dostupné z: https://clang.llvm.org/docs/LibASTMatchersTutorial.html

[15] Cppcheck - CERT C Coding Standard. 2017-01-05, [cit. 2017-01-05]. Dostupné z: https://www.securecoding.cert.org/confluence/display/c/Cppcheck

[16] Cppcheck - A tool for static C/C++ code analysis. [cit. 2017-01-05]. Dostupné z: http://cppcheck.sourceforge.net/

[17] Rose compiler infrastructure. [cit. 2017-01-05]. Dostupné z: http://rosecompiler.org/

[18] Projects | Rose compiler infrastructure. [cit. 2017-01-05]. Dostupné z: http://rosecompiler.org/?page_id=16

[19] Secure Coding: Rosecheckers | The CERT Division Projects. [cit. 2017-01-05]. Dostupné z: https://www.cert.org/secure-coding/tools/rosecheckers.cfm

[20] Rose - SEI CERT C Coding Standard. 2017-01-05, [cit. 2017-01-05]. Dostupné z: https://www.securecoding.cert.org/confluence/display/c/Rose

[21] CERT Rosecheckers Commit Browser. [cit. 2017-02-10]. Dostupné z: https://sourceforge.net/p/rosecheckers/code/commit_browser

[22] MISRA C. [cit. 2017-02-14]. Dostupné z: https://www.misra.org.uk/misra-c/Activities/MISRAC/tabid/160/Default.aspx

[23] LibTooling – Clang 5 documentation. [cit. 2017-02-15]. Dostupné z: http://clang.llvm.org/docs/LibTooling.html

[24] Secure Coding Validation Suite. [cit. 2017-02-15]. Dostupné z: https://www.cert.org/secure-coding/tools/validation-suite.cfm?

[25] EM6812 | EM Microelectornic. [cit. 2017-02-15]. Dostupné z: http://www.emmicroelectronic.com/products/microcontrollers/flash-based-mcu/em6812

[26] Contiki: The Open Source Operating System for the Internet of Things. [cit. 2017-02-15]. Dostupné z: http://www.contiki-os.org/

[27] Atmel Software Framework. [cit. 2017-02-15]. Dostupné z: http://www.atmel.com/tools/avrsoftwareframework.aspx

[28] PRE11-C - SEI CERT C Secure Coding Standard. [cit. 2017-01-28]. Dostupné z: https://www.securecoding.cert.org/confluence/display/c/PRE11-C.+Do+not+conclude+macro+definitions+with+a+semicolon

[29] Kernighan, B. W.; Ritchie, D. M.: *Programovací jazyk C*. Computer Press, Brno, první vydání, 2008, ISBN 80-251-0897-X.

Příloha A

Implementované kontroly

A.1 PRE01-C Používejte závorky uvnitř maker kolem názvů parametrů

Makra podobná funkcím nemají stejné vlastnosti, jako „skutečné“ funkce, což může vést k neočekávaným výsledkům při nepozornosti programátora, nebo pokud nejsou dodržovány jasné konvence v pojmenování maker a programátor-uživatel tedy ani neví, že nevolá skutečnou funkci, ale pouze makro.

Toto doporučení přikazuje použít závorky kolem argumentů v makru. Díky tomu má výsledné makro vlastnosti podobnější funkci. Vezměme si tento příklad:

```
1 #define MUL(a, b) (a * b)
2 int a = MUL(1 + 2, 3);
```

po nahrazení vznikne

```
1 int a = 1 + 2 * 3;
```

Hodnota proměnné `a` je tedy 7, ačkoliv programátor pravděpodobně předpokládal výsledek 9. Definice makra v souladu s tímto doporučením vypadá takto:

```
1 #define MUL(a, b) ((a) * (b))
```

čímž vznikne pravděpodobněji předpokládaný výsledek

```
1 int a = (1 + 2) * 3 // == 9
```

Z tohoto doporučení existují dvě výjimky:

1. Pokud jsou argumenty v pravé straně makra odděleny čárkami, nehrozí chybná expanze, protože čárky mají ze všech binárních operátorů v jazyce C nejnižší prioritu vyhodnocení.

2. Parametry makra nemohou být uzávorkované, pokud je na ně použit operátor sjednocení řetězců (`##`) nebo operátor konverze na řetězec (`#`).

Checker nejprve zkontroluje, zda se jedná o makro podobné funkci. Clang k tomuto účelu poskytuje metodu `isFunctionLike()`. Pokud makro vyhoví, je provedena kontrola každého identifikátoru. Checker zde využívá metodu třídy `MacroInfo` `getArgumentNum()`, která pro daný identifikátor vrátí jeho index jako argumentu pro makro podobné funkci. Nás zajímá pouze, jestli je návratová hodnota různá od nuly, tedy zda se opravdu jedná o argument makra.

```
1  for (const auto& token : macro->tokens()) {
2      const auto tokenKind = token.getKind();
3      if (tokenKind == tok::identifier) {
4          if (macro->getArgumentNum(
5                  token.getIdentifierInfo()) != -1) {
6              if (!IsIdentifierCompliant(macro, &token)) {
7                  // nahlas chybu
8              }
9          }
10     }
11 }
```

Výpis A.1: Hlavní cyklus z kontroly PRE01-C (upraveno)

Metoda `IsIdentifierCompliant` poté už jen pro daný identifikátor zkontroluje, zda je obklopen přípustnými tokeny (závorky, `##`, příp. `#`).

A.2 PRE02-C Pravá strana by měla být uzávorkovaná

Toto doporučení pokrývá podobnou skupinu problémů, jako doporučení PRE01-C popsané v sekci A.1.

```
1  #define SUM(a, b) (a) + (b)
2  int a = 2 * SUM(1, 2);
```

po nahrazení vznikne

```
1  int a = 2 * 1 + 2; // vyhodnoceno jako (2 * 1) + 2 = 4
```

K dosažení předpokládaného výsledku je třeba makro definovat takto:

```
1  #define SUM(a, b) ((a) + (b))
```

po nahrazení:

```
1  int a = 2 * (1 + 2) // == 6
```

Doporučení vyjmenovává několik výjimek, kdy pravá strana nemusí být uzávorkovaná. Myslím, že je doporučení napsáno zbytečně složitě, protože vágně zachází s výrazem „macro replacement list." Pro implementaci kontroly jsem použil svou interpretaci umožňující jednodušší implementaci: Pravá strana makra musí být uzávorkovaná, pokud:

1. obsahuje alespoň 2 literály oddělené binárním operátorem a zároveň tato dvojice není uvnitř argumentů funkci, nebo

2. obsahuje 1 literál, kterému předchází unární mínus.

Viz příklad A.2.

```
1 #define FOO pow(1 + 2, 3)          // nemusi byt uzavorkovane
2 #define BAR pow(1 + 2, 3) + 4      // melo by byt uzavorkovane
3 #define END_OF_INPUT -1            // melo by byt uzavorkovane
```

Výpis A.2: Příklad vyhovujícího a nevyhovujícího makra

Důležitost posledního příkladu ilustruje výpis A.3.

```
1 #define END_OF_INPUT -1
2 // puvodne zamysleny kod
3 while (func() != END_OF_INPUT) {}
4 // omylem nenapsany porovnavaci operator
5 while (func() END_OF_INPUT) {}
6 // vysledek nahrazeni makra
7 while (func() - 1) {}
```

Výpis A.3: Možné důsledky špatného uzávorkování záporného čísla

V případě správného uzávorkování by kompilátor takový kód vůbec nepřeložil, viz A.4.

```
1 #define END_OF_INPUT_CORRECT (-1)
2 // nahrazeni v pripade spravneho uzavorkovani makra
3 // syntakticka chyba
4 while (func() (-1)) {}
```

Výpis A.4: Správné uzávorkování předejde neočekávanému opakování cyklu

A.3 PRE10-C Zabalte více příkazová makra do cyklu do-while

Doporučení přikazuje zabalit více příkazová makra do cyklu do-while z důvodu možné chybné expanze po kontrolních příkazech.

```
1 #define SWAP(a, b) \
2     tmp = a; \
3     a = b; \
4     b = tmp;
5
6 if (a > b)
7     SWAP(a, b);
```

Výpis A.5: Ukázka porušující doporučení PRE10-C

Ve výsledné expanzi pak vidíme, že jen první řádek makra se provede jen při splnění podmínky; zbývající dva se provedou vždy.

```
1 if (a > b)
2     tmp = a;
3 a = b;
4 b = tmp;;
```

Výpis A.6: Kód z A.5 po expanzi makra

Obalení makra v bloku `do { ... } while(0)` této chybě zabrání. Pro podobný efekt je možné použít i konstrukci `if(1) { ... } else {}`, tu ale CERT C SCS jako přípustnou neuvádí.

Působení tohoto doporučení se překrývá s doporučením EXP19-C „Používejte složené závorky pro tělo příkazů if, for a while." To má ale podle vyhodnocení rizika ve standardu nižší prioritu.

Checker nejprve zkusí uhodnout, zda se jedná o více příkazové makro tím, že se pokusí najít středník, který zároveň není posledním tokenem v makru. Jedná se o hrubý, ale jednoduše proveditelný odhad.

```
1 for (auto& token : macro->tokens()) {
2     if (token.is(tok::semi) &&
3         &token != &macro->tokens().back()) {
4         multiStatement = true;
5         break;
6     }
7 }
```

Výpis A.7: Detekce více příkazového makra v kontrole PRE10-C

Pokud je makro vyhodnocené jako více příkazové, prochází samotnou kontrolou. Nejdříve je zkontrolováno, zda makro začíná klíčovým slovem `do`. Poté odzadu kontroluje, zda konec makra odpovídá předpokládanému vzoru „} while(vyraz)". Makro také může končit středníkem, ačkoliv je to v rozporu s doporučením PRE11-C (viz kapitola A.4), aby při použití pouze vybraných kontrol dostal uživatel hlášení pouze o chybách, které ho skutečně zajímají.

```
1  llvm::SmallVector<clang::tok::TokenKind, 4U> pattern =
2      {tok::r_brace, tok::kw_while, tok::l_paren, tok::r_paren};
3  //...
4  for (auto token = macro->tokens_end() - 1;
5      token >= macro->tokens_begin(); token--) {
6          if (token->is(*patternIt)) {
7              if (patternIt == pattern.begin()) {
8                  return true; // makro je spravne zabalene
9              }
10             patternIt--;
11         } else if (*patternIt != tok::l_paren) {
12             return false; // makro neni spravne zabalene
13         }
14     }
15 }
```

Výpis A.8: Hledání vzoru v kontrole PRE10-C

Do budoucna by bylo vhodné kontrolu rozšířit o výjimku, že by nemělo být jako chybné detekováno makro, které je nahrazeno funkcí (viz příklad A.9 pocházející přímo z implementace této práce), či strukturou (viz falešná detekce v kapitole 6.2.2).

```
1 #define CHECKER_SIMPLE_REGISTRATION(CheckClass) \
2     void register##CheckClass(clang::ento::CheckerManager &mgr) { \
3         mgr.registerChecker<CheckClass>(); \
4     }
```

Výpis A.9: Makro, které by nemělo být detekováno kontrolou PRE10-C language

A.4 PRE11-C Nezakončujte makra středníkem

Ukončení makra středníkem může způsobit expanzi neočekávanou uživatelem makra:

```
1 #define FOR(n) for(i = 0; i < (n); i++);
2 #define SUM(A, B) ((A) + (B));
3
4 int i;
5 FOR(3) { /* pracuj */ } // for(...); {}
6 int result = SUM(1, 2) + 3; // (1 + 2); + 3;
```

Výpis A.10: Kód nevyhovující PRE11-C, adaptováno z [28]

V prvním případě se jedná velmi pravděpodobně o omyl autora makra (pokud se nesnažil implementovat zpoždění na platformě, která nepodporuje `sleep`).

Před touto chybou, resp. jejím důsledkem varují některé kompilátory. Bez přepínačů na oba případy reaguje `clang` – „for má prázdné tělo“ a „nepoužitý

výsledek výrazu *+ 3*." GCC s přepínačem `-Wall` varuje ve druhém případě, že výraz *+ 3* nemá žádný efekt.

Implementace je velmi jednoduchá a její zásadní část se dá shrnout do pár řádků:

```
1 void MacroDefined(const Token &MacroNameTok,
2         const MacroDirective *MD)
3     MacroInfo *macro = MD->getMacroInfo();
4     if ((macro->tokens_end() - 1)->getKind() == tok::semi) {
5         // nahlas chybu
6     }
7 }
```

Výpis A.11: Implementace kontroly PRE11-C

Stačí zkontrolovat poslední token v definici makra, zda se jedná o středník. Zbytek kódu checkeru pouze rozhoduje, zda je vhodné makro kontrolovat (tedy např. zda nepochází ze systémové hlavičky, nebo jestli má nějaké tokeny) a hlásí případnou chybu.

A.5 DCL30-C Deklarujte objekty se vhodnou dobou platnosti

Podle standardu C99 může mít proměnná dvě paměťové třídy: buď statickou, nebo automatickou [29]. Ve standardu C11 přibyla třída thread-local. Tyto třídy určují délku života objektu. Hodnota ukazatele, který ukazuje na objekt, jemuž skončila platnost, je nedefinovaná.

Pravidlo vyjmenovává tři zakázané situace:

1. Rozdílné paměťové třídy

```
1 int *global;
2 void foo(void) {
3     int local = 5;
4     global = &local;
5 }
```

Výpis A.12: DCL30-C rozdílné paměťové třídy

Pokud by byla do proměnné `global` přiřazena jiná hodnota (např. `NULL`) před koncem platnosti proměnné `local`, pak by se nejednalo o porušení pravidla.

2. Návratová hodnota

```
1 int *foo(void) {
2     int local = 5;
3     return &local;
4 }
```

Výpis A.13: DCL30-C navrácení ukazatele na lokální proměnnou

Před tímto případem varují všechny běžné kompilátory.

3. Výstupní parametr

```
1 void foo(int *outParam) {
2     int local = 5;
3     outParam = &local;
4 }
5
6 void bar(void) {
7     int *ptr;
8     foo(ptr);
9 }
```

Výpis A.14: DCL30-C adresa lokální proměnné uložená do výstupního parametru

Checker momentálně podporuje jen kontrolu velmi základního případu, uvedeného v ukázce A.12 v bodě 1., kdy je statickému ukazateli přiřazena adresa automatické proměnné.

Kontrolovat složitější případy, např. ty, které nastávají mezi dvěma automatickými proměnnými (viz. A.15), se ukázalo jako velmi složité. Zajímavé je, že pravidlo tyto případy explicitně neuvádí. Z věty „Do not attempt to access an object outside of its lifetime.“[6, str. 32] a celkového vyznění pravidla je však jasné, že takový kód pravidlu také odporuje.

```
1 void foo(void) {
2     int *a;
3     if (1) {
4         int b = 5;
5         a = &b;
6     }
7     printf("%d\n", *a) //nedefinovane chovani
8 }
```

Výpis A.15: DCL30-C ukazatel na dealokovanou proměnnou

CSA sice umožňuje checkerům implementovat metodu `checkDeadSymbols`, která má být podle dokumentace volána, když symbol „umře.“ Ta je však analyzátorem volána jen když je daný symbol asociován s dynamicky alokovanou paměť.

A.6 EXP46-C Nepoužívejte bitové operátory s booleovskými operandy

Současné použití bitových a porovnávacích operátorů v rámci jednoho výrazu může být výsledkem překlepu, kdy programátor chtěl použít operátor `&&`, ale omylem napsal `&`. V takovém případě má podmíněný výraz neočekávaný výsledek.

Toto pravidlo zakazuje použití bitových operátorů ve stejném výrazu, ve kterém je použit operátor porovnání. V případě, že je bitový operátor použit schválně, má být podvýraz, který ho užívá, uzávorkován.

Ukázka A.16 ilustruje porušení tohoto pravidla. Ukázka A.17 pak dvě možné nápravy této chyby v závislosti na původním úmyslu autora kódu.

```
1 if (a & b == 1) {}
```

Výpis A.16: EXP46-C pravděpodobný překlep

```
1 if (a && b == 1) {}
2 // nebo
3 if ((a & b) == 1) {}
```

Výpis A.17: EXP46-C dvě možné opravy A.16

Kontrola probíhá čistě z AST. Pro její implementaci je použita třída `WalkAST`, která dědí ze třídy `StmtVisitor`. Ta umožňuje jednoduše pomocí zpětných volání projít celý AST.

```
1 class WalkAST : public StmtVisitor<WalkAST>
2 {
3 public:
4     void VisitStmt(Stmt *S);
5     void VisitChildren(Stmt *S);
6     void VisitBinaryOperator(BinaryOperator *bo);
7 };
```

Výpis A.18: EXP46-C třída WalkAST (zkráceno)

Z pohledu této kontroly je důležitá metoda `VisitBinaryOperator`, která je volána, když je při průchodu AST naraženo na výraz typu binární operátor. Jádro implementace je na ukázce A.19

```
1  void WalkAST::VisitBinaryOperator(BinaryOperator *binOp)
2  {
3      if (!binOp->isBitwiseOp()) {
4          return;
5      }
6
7      bool lhsIsBoolean = binOp->getLHS()->
8          isKnownToHaveBooleanValue();
9      bool rhsIsBoolean = binOp->getRHS()->
10         isKnownToHaveBooleanValue();
11
12     if (lhsIsBoolean || rhsIsBoolean) {
13         // nahlas chybu
14     }
15 }
```

Výpis A.19: EXP46-C metoda VisitBinaryOperator (zkráceno)

Kontrola probíhá jen pokud se jedná o bitový operátor. Poté je na každý operand tohoto operátoru zavolána metoda `isKnownToHaveBooleanValue`, který vrátí `true` pro výrazy typu `_Bool` a pro celočíselné výrazy, jejichž výsledkem je 0 nebo 1 (např. výsledky porovnání).

A.7 INT13-C Používejte bitové operátory pouze na operandy bez znaménka

Toto doporučení zakazuje použití bitových operátorů na čísla se znaménkem, protože jejich výsledek je závislý na implementaci. Navíc chování bitových operátorů `<<` a `>>` je v mnoha případech pro celá čísla se znaménkem nedefinované.

Doporučení vyjmenovává dvě výjimky:

1. Když jsou konstanty použité jako bitové flagy, je povoleno je použít jako argumenty operátorů `&` a | i když nejsou explicitně označené jako `unsigned`.
2. Pokud je hodnota pravého operandu bitového posunu známá při kompilaci, může být vyjádřená typem se znaménkem, pokud je tato hodnota kladná.

Pro co nejlepší pokrytí doporučení a jeho výjimek jsem implementoval checker jako path-sensitive. Checker hlásí chybu, pokud je libovolný bitový operátor použitý na operand, který není deklarován jako `unsigned` a zároveň jeho hodnota buď není známá, nebo je záporná.

Jádro kontroly používá metodu `checkPreStmt`, implementovanou zvlášť pro parametr typu `UnaryOperator` (ve výpisu A.20) a zvlášť pro `BinaryOperator` (ta je implementovaná obdobně).

```
1  void Int13CChecker::checkPreStmt(
2      const UnaryOperator *unOp, CheckerContext &c) const
3  {
4      if (unOp->getOpcode() != UnaryOperatorKind::UO_Not) {
5          return;
6      }
7
8      auto operand = tryExtractingExpr(unOp);
9      if (operand == nullptr) {
10         return;
11     }
12
13     if (isExprCompliant(operand, c) == false) {
14         reportError(c);
15     }
16 }
```

Výpis A.20: INT3-C metoda checkPreStmt pro UnaryOperator

Metoda `tryExtractingExpr` se pokusí z potenciálně složitého předaného výrazu „vyextrahovat" výraz užitečnějšího typu: `DeclRefExpr` (přímou referenci proměnné), `IntegerLiteralExpr` (celočíselný literál), nebo `UnaryOperator` (unární operátor) typu unární minus.

Metoda `isExprCompliant` vrátí `true`, pokud je daný operand buď typu `unsigned`, nebo je o něm analyzátoru známo, že je nezáporný.

Kontrolu nezápornosti pak provádí metoda `isKnownNonNegative`, která využívá schopností CSA a jeho třídy `ConstraintManager`.

```
1  bool Int13CChecker::isKnownNonNegative(const SVal& val,
2      CheckerContext &c) const
3  {
4      Optional<DefinedSVal> dV = val.getAs<DefinedSVal>();
5      ProgramStateRef state = c.getState();
6
7      const TypedValueRegion *TR =
8          dyn_cast_or_null<TypedValueRegion>(val.getAsRegion());
9      if (TR) {
10         auto rawVal = state->getRawSVal(*val.getAs<Loc>(),
11             TR->getValueType());
12         dV = rawVal.getAs<DefinedSVal>();
13     }
14
15     SValBuilder &svalBuilder = c.getSValBuilder();
16     auto zeroVal = svalBuilder.makeIntVal(0, false);
17
18     SVal geZeroCond = svalBuilder.evalBinOp(
19         state, BO_GE, *dV, zeroVal, svalBuilder.getConditionType());
20     Optional<DefinedSVal> geZeroDefined =
21             geZeroCond.getAs<DefinedSVal>();
22
23     ProgramStateRef geZeroState, ltZeroState;
24     ConstraintManager &constraintManager =
25         c.getConstraintManager();
26     std::tie(geZeroState, ltZeroState) =
27         constraintManager.assumeDual(state, *geZeroDefined);
28
29     if (!ltZeroState && geZeroState) {
30         return true;
31     }
32     else {
33         return false;
34     }
35 }
```

Výpis A.21: INT13-C metoda isKnownNonNegativ (zkráceno)

Symbolická hodnota `val` předaná této metodě může být dvou druhů – buď může symbolizovat výsledek nějaké operace, nebo oblast paměti. Na řádku 7 se jí proto pokusí metoda přetypovat na `TypedValueRegion`. Pokud se to povede, je třeba z této `SVal` získat `RawSVal`, která symbolizuje hodnotu v daném místě paměti.

Na řádku 16 je vytvořena `SVal` reprezentující nulu, proti které bude předaná hodnota porovnána. Na řádku 18 je pak vytvořena samotná podmínka `val >= 0`.

Metoda `assumeDual` volaná na řádku 27 podmínku vyhodnotí a vrátí dvojici stavů – první, ve kterém je hodnota větší nebo rovna nula a druhý, kde je hodnota menší než nula. Pokud je některý z těchto dvou stavů nemožný, vrátí místo něj metoda `nullptr`.

A.8 FLP30-C Nepoužívejte floating-point proměnné jako počítadla v cyklech

Čísla s pohyblivou řádovou čárkou neumožňují přesně vyjádřit všechny možné zlomky. V případě použití čísla s pohyblivou řádovou čárkou jako počítadla v cyklu, může podle CERT C SCS dojít ke dvěma problémům [6]:

```
1 for (float x = 0.1f; x <= 1.0f; x += 0.1f) {
2     ...
3 }
```

Výpis A.22: FLP30-C nejasný počet opakování, převzato z [6]

Cyklus v ukázce A.22 může mít na různých platformách různý počet opakování.

```
1 for (float x = 100000001.0f; x <= 100000010.0f; x += 1.0f) {
2     ...
3 }
```

Výpis A.23: FLP30-C nekonečný cyklus, převzato z [6]

Cyklus v ukázce A.23 může být v závislosti na platformě nekonečný.

Implementovaný checker nekontroluje přímo „počítadlo“ v cyklu, ale samotnou podmínku, zda se v ní vyskytuje výraz typu `float`. Tento přístup poskytuje dostatečnou přesnost kontroly.

Samotná kontrola probíhá čistě z AST, podobně jako v případě kontroly EXP46-C, popsané v části A.6. Třída WalkAST v tomto případě implementuje metody `Visit...` pro `ForStmt`, `WhileStmt` a `DoStmt` (viz výpis A.24). Každá z těchto metod pouze předá podmínku z daného výrazu metodě `CheckCondition`, která provede samotnou kontrolu.

```
1 void WalkAST::VisitForStmt(ForStmt *forStmt)
2 {
3     CheckCondition(forStmt->getCond());
4 }
```

Výpis A.24: VisitForStmt ze třídy WalkAST kontroly FLP30-C

```
void WalkAST::CheckCondition(Expr *condition)
{
    bool compliant = true;

    if (const BinaryOperator* bo =
            llvm::dyn_cast<BinaryOperator>(condition)) {
        if (bo->getLHS()->getType()->isFloatingType()) {
            compliant = false;
        }
        if (bo->getRHS() getType()->isFloatingType()) {
            compliant = false;
        }
    }
    else {
        if (condition->getType()->isFloatingType()) {
            compliant = false;
        }
    }

    if (compliant == false) {
        // nahlas chybu
    }
}
```

Výpis A.25: CheckCondition ze třídy WalkAST kontroly FLP30-C (zkráceno)

Metoda `CheckCondition` pak nejprve zkusí přetypovat výraz reprezentující podmínku na typ `BinaryOperator`. Pokud se to podaří, pak zkontroluje, jestli je alespoň jedna ze stran výrazu typu floating point. Pokud ano, je výraz označen za chybný. Pokud se přetypování nezdaří, znamená to, že podmínka je ve tvaru `while(samotnaPromenna)`. V tomto případě se postupuje obdobně, pouze není potřeba kontrolovat podvýrazy, ale stačí zkontrolovat samotný výraz `condition`.

Přesná kontrola jiných pravidel z kapitoly *Floating point* se zdá být, vzhledem k omezením CSA (viz kapitola 4.4), velmi náročná, až nemožná.

A.9 ARR36-C Neodečítejte od sebe nebo neporovnávejte dva ukazatele, které neukazují na stejné pole

Odečtení dvou ukazatelů, které neukazují na stejné pole, stejně jako jejich porovnání pomocí operátorů <, <=, >= a <, je nedefinované chování, proto je tímto pravidlem zakázáno. Porovnání pomocí operátorů == a != je vždy definováno, proto zakázáno není.

Toto pravidlo zakazuje odečítat či porovnávat pomocí vyjmenovaných operátorů 2 ukazatele, které ukazují na cokoliv, kromě jednoho stejného pole. Vý-

jimkou je použití operátoru porovnání na ukazatele ukazující na různé prvky stejné struktury (výsledkem takového porovnání je pořadí prvků v dané struktuře). Všechny tyto příklady porušují pravidlo ARR36-C:

```
1  struct st
2  {
3      int a;
4      int b;
5  }
6
7  void foo(void)
8  {
9      struct st s;
10     int a[10], b[10];
11     int *pa, *pb;
12     int result;
13
14     pa = &s.a;
15     pb = &s.b;
16
17     // rozdil ukazatelu na prvky struktury
18     result = pb - pa;
19
20     pa = a;
21     // rozdil resp. porovnani ukazatelu na rozdilne objekty
22     result = pa - pb;
23     result = pa < pb;
24
25     pb = b;
26     // rozdil ukazatelu na dve ruzna pole
27     result = pa - pb;
28 }
```

Výpis A.26: Kód porušující pravidlo ARR36-C

Checker kontroluje všechna použití výše vyjmenovaných binárních operátorů mezi dvěma ukazateli. Třída `CheckerContext` z CSA, jejíž instance je předávána každé metodě `check`, poskytuje metodu `getSVal()`, která vrátí pro daný identifikátor (jakožto uzel v AST) jeho symbolickou hodnotu v současném bodě aktuálního průchodu programem. Třída `SVal` pak má metodu `getAsRegion()`, z vráceného regionu pomocí metody `getBaseRegion()` získáme abstraktní reprezentaci základní paměťové oblasti, na kterou ukazatel ukazuje. Touto oblastí může být např. pole, struktura, nebo proměnná.

```
1 auto lsval = C.getSVal(lhs);
2 auto rsval = C.getSVal(rhs);
3
4 bool sameRegion =
5     (rsval.getAsRegion()->getBaseRegion() ==
6      lsval.getAsRegion()->getBaseRegion());
```

Výpis A.27: Získání základní oblasti v checkeru ARR36-C

Poté je potřeba zkontrolovat, jestli ukazatele neukazují na prvky struktury, protože – jak bylo uvedeno v prvním odstavci – pro ukazatele na prvky struktury platí jiná pravidla. Toto se nedá zjistit přímo ze třídy `MemRegion`. Deklarační kontext (třída `DeclContext`) sice má metodu `containsDecl`, ale ze třídy `MemRegion` není možné přímo získat deklaraci, pouze název regionu. Proto jsem implementoval metodu `pointsToStructMember()`, která se pokusí danou oblast najít podle názvu v seznamu deklarací.

```
1  bool Arr36CChecker::pointsToStructMember(SVal& sval,
2                                           Expr* expr) const
3  {
4      DeclRefExpr *declRef = nullptr;
5      if (!tryCastingToDeclRefExpr(expr, &declRef)) {
6          return false;
7      }
8
9      auto context = declRef->getDecl()->getDeclContext();
10     auto regionName = sval.getAsRegion()->
11         getBaseRegion()->getDescriptiveName();
12
13     // Zkusime najit strukturu daneho jmena
14     if (containsDeclWithName(context->decls(), regionName)) {
15         return true;
16     }
17     // Nenasli jsme. Zkusime to v nadrazenem jmennem prostoru
18     if (containsDeclWithName(
19         context->getEnclosingNamespaceContext()->decls(),
20             regionName)) {
21         return true;
22     }
23     // Ukazatel neukazuje na prvek struktury
24     return false;
25 }
```

Výpis A.28: Zjištění, jestli ukazatel ukazuje na položku struktury v ARR36-C

Problém je nahlášen v případě, že ukazatele neukazují na stejnou základní oblast, nebo ukazují na stejnou základní oblast, ale tato je strukturou a zároveň je použit jiný, než porovnávací operátor.

```
1 if (sameRegion == false ||
2      (pointsToStructMember(lsval, lhs)
3          && usedBinaryOperator->isComparisonOp() == false)
4      ) {
5          // nahlas chybu
6 }
```

Výpis A.29: Jádro kontroly ARR36-C

Příloha B

Seznam použitých zkratek

AST Abstract syntax tree – abstraktní syntaktický strom.

CERT C SCS SEI CERT C Secure Coding Standard.

CSA Clang Static Analyzer.

Příloha C

Obsah přiloženého DVD

```
readme.txt ........................ popis obsahu DVD
bin ............................... adresář se spustitelnou formou implementace
src
  impl ............................ zdrojové kódy implementace
  thesis .......................... zdrojová forma práce ve formátu LaTeX
text .............................. text práce
  BP_Simek_Jakub_2017.pdf ......... text práce ve formátu PDF
benchmarks ........................ soubory použité pro testování
```