Statická analýza a verifikace
Využití bi-abdukce pro analýzu bezpečnosti paměti v rámci nástroje Meta Infer

David Hudák (xhudak03)

30. června 2024

1 Úvod

V této práci se zaměříme na využití nástroje Meta Infer pro verifikaci práce s pamětí. Konkrétně se podíváme na princip nazývaný bi-abdukce, který je obohacen o stručný úvod k tématu, ze kterého bi-abdukce vychází – separační logice, a o krátké shrnutí jejího potenciálního nástupce nazývaného jako Pulse. V následujících sekcích pak ukážeme experimenty na Inferu jak na programech, které uvádí sami autoři, tak experimenty na dalším kódu.

2 Teoretická průprava

V této sekci se budeme zabývat obecnými principy nástroje Meta Infer pro verifikaci bezpečnosti paměti. Především si uvedeme na úvod to, co Meta Infer umí řešit a následně se zaměříme na konkrétní principy separační logiky a bi-abdukce. Také si zde krátce popíšeme novější odbočku v nástroji meta infer na checker nazvaný Pulse, který by potenciálně mohl právě řešení založené na bi-abdukci nahradit[1].

2.1 Meta Infer

Meta Infer je nástroj pro statickou verifikaci programů, který společnost Meta (tehdy Facebook) v roce 2013 odkoupila od společnosti Monoidics [2]. Nástroj společnost využívá především pro verifikaci svých vlastních aplikací pro různá zařízení, kterým tak předává zpětnou vazbu svým zaměstnancům při každém významném „diffu“. Hlavní devízou pak je, že umí dávat zpětnou vazbu na neúplný program, tj. nemusí verifikovat vyvíjený produkt jako celek, ale jako sjednocení menších dílčích částí. To se hodí právě pro princip toho, jak jsou produkty Meta (Facebook, Instagram, Snapchat. . .) vyvíjeny – jedná se o tzv. trvalý vývoj, kdy například u webové aplikace, alespoň dle článku z roku 2015 [2], přichází nové updaty i dvakrát denně.

Nástroj Infer existuje dostupný v různých verzích – alespoň dle stránek ve verzi předchozí (1.0.0), současné (1.1.0) a Next, která obsahuje nejnovější nástroje a technologie, které ještě nebyly uvolněny do stabilní verze projektu[5]. Každá verze pak obsahuje extenzivní množství různých oblastí a nástrojů, se kterými je schopen pracovat, ať už se jedná například o živost, přetečení bufferů, hladovění, neinicializované hodnoty či z pohledů nástrojů v této práci zmiňovaná bi-abduce a Pulse[2].

[1] Toto alespoň tvrdí stránka z dokumentace k Next verzi nástroje Meta Infer, viz https://fbinfer.com/docs/next/checker-pulse/.

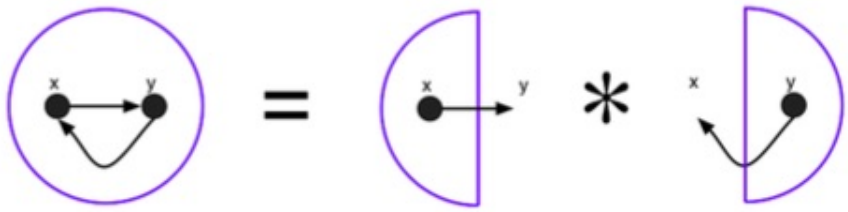
[2] Celý rozsáhlý seznam řešených problémů můžeme najít na stráncehttps://fbinfer.com/docs/all-issue-types.

2.2 Separační logika

Článek [8] popisuje úplně původní přístup k separační logice, kdy rozšiřují Hoarovu logiku o schopnost popisovat programy, které mění (dynamické) datové struktury. Tam pracují s typickými Hoarovými trojicemi $\{P\}C\{Q\}$, kdy P nám popisuje podmínku, co platila předtím(precondition) , C nějaký program, který je prováděn a Q, která popisuje podmínku, která má platit pro provedení programu (postcondition). Ty rozšiřují o operátor tzv. „prostorové konjunkce“ $*$[3], kdy pro nějaké $P * R$ platí, že P a R platí nezávisle pro různé části struktury. Například zápis $\{P * R\}C\{Q * R\}$ říká, že po provedení operací C zůstává zbytek struktury R stejný. Tj. pokud bych například odebral první prvek ze zásobníku, tak pořád musí platit, že zbytek zásobníku zůstává stejný. Důsledkem tohoto operátoru například je, že pokud uvážíme následující zápis:

$$(x \to y * y \to x) \tag{1}$$

kde operátor $\to$ si můžeme přeložit jako ukazatel x ukazuje na kus paměti y a separátně y ukazuje na kus paměti x [5]. Zakazuje to tak aliasing, viz vizualizace z dokumentace Meta Infer:



Obrázek 1: Ukázka operátoru prostorové/separační konjunkce pro Rovnici 1 z dokumentace.

V článku [1] říkají tomuto rozdělení rozdělení globálního heapu na tzv. heaplety. Z toho pak vychází tzv. frame rule (rámcové pravidlo):

$$\frac{[P]C[Q]}{[R * P]C[R * Q]}$$

kde C nezapisuje do R. Toto nám umožňuje využít principu lokality, kdy pro verifikaci nemusíme pracovat s celou strukturou, ale stačí nám popsat konkrétní datovou buňku [4]. Někdy tento přístup bývá popisován, že využívá principu footprintů (stop).

Prvním nástrojem, který, alespoň dle vývojářů Meta Infer, přináší aplikaci (a další rozšíření) sperační logiky k verifikaci, byl Smallfoot [1]. Ten, vyjma tedy již zmíněného frame rule uvádí ještě dvě pravidla pro souběžné procesy. První je tvaru:

$$\frac{[P]C[Q] \qquad [P']C'[Q']}{[P * P']C||C'[Q * Q']}$$

kde program C nezasahuje do volných proměnných v P', C' a Q' a program C' nezasahuje do P, C a Q. Například v případě heapu $[P * P']$ zajišťuje, že proměnné obsažené v P a v P' se navzájem nealiasují. Druhým pravidlem je popis práce s podmíněnými kritickými regiony, které autoři zapisují jako `with` r `when`$(B)\{C\}$, kde r je nějaký společný zdroj, ke kterému je nutné přistupovat s výlučným přístupem, B je podmínkou pro práci s tímto zdrojem a C je nějaký kód, který má být na daném zdroji vykonán. Z toho pak vychází inferenční pravidlo:

$$\frac{[(P * R_r) \wedge B]C[Q * R_r]}{[P]\ \texttt{with}\ r\ \texttt{when}\ (B)\{C\}[Q]}$$

[3]Originálně spatial conjunction. Zdroj [5] pak říká, že se jedná o separation conjuction.

kde R_r je nějaká formule týkající se zdroje r.

Článek [4] pak rozšiřuje separační logiku o zavedení výpočtu invariantů cyklů na základě aritmetiky pevného bodu a přizpůsobuje tak separační logiku abstrakční interpretaci, kdy popisují aplikaci na lineárních seznamech. Ta pracuje v sémantickém prostředí úplných svazů D, která vychází z potenční množiny nějaké množiny S sjednocené s $\top$. Speciální element $\top \notin S$ koresponduje s paměťovou chybou. Příkazy c pak na těchto svazech definují spojité funkce $[[c]] : D \to D$. Pro booleovské proměnné používají notaci $\texttt{filter}(b) : D \to D$, kdy D dle definice sestává z nějaké podmnožiny množiny stavů a `filter` z ní odfiltruje stavy nesplňující podmínku b. Dále rozšiřuje sémantiku o:

- Posloupnost příkazů: $[[c; c']] = [[c]]; [[c']]$
- Podmínky: $[[\texttt{if } b \texttt{ then } c \texttt{ else } c']] = (\texttt{filter}(b);\ [[c]]) \sqcup (\texttt{filter}(\neg b);\ [[c']])$
- Cykly: $[[\texttt{while } b \texttt{ do } c]] = \lambda d.filter(\neg b)(fix\ \lambda d'.d \sqcup (\texttt{filter}(b); [[c]])(d'))$, kde d' popisuje nějaký invariant cyklu

Krom této sémantiky ještě v článku [4] následuje rozsáhlý popis jednak konkrétní, jednak symbolické sémantiky pro heap a pro vykonávání příkazů. Tyto sémantiky z toho důvodu, že vedou na nekonečné domény, pak aproximují na korektní nad-aproximující abstrakce, které popisují další sérií přepisovacích pravidel. Například pro cykly, které vedou potenciálně na nekonečnou posloupnost příkazů, provádí sjednocení více seznamů do jednoho či převádí buňky na seznamy s nějakým definovaným chováním.

2.3 Bi-abdukce

Dokumentace Meta Infer [5] k bi-abdukci přímo uvádí[4] *„Bi-abdukce je formou logické inference pro separační logiku, která automatizuje hlavní ideje o lokálním usuzování.“* Přesněji, bi-abdukce nástroji Infer slouží k tomu, aby sám automatizovaně odvodil platné předpoklady (preconditions) a důsledky na základě principu odvozování rámců a anti-rámců za předpokladu, že známe specifikace primitiv, se kterými pracujeme. Ve zdrojích, které se věnují tomuto tématu, se úloze, kterou se snažíme bi-abdukcí řešit, také říká „shape analysis“ (analýza tvaru). Obecněji se dané úloze říká verifikace datových struktur[3, 7].

2.3.1 Základní pojmy a vlastnosti

Samotná bi-abdukce vychází z principu abdukce, kterou popsal jeden z nejvýznamnějších amerických filozofů Charles Sanderse Peirce [3] ve svých psaních o vědeckém procesu [9], kdy jejím základním filozofickým principem je mechanická tvorba hypotéz. Její hlavní motivací v Inferu je urychlit verifikaci kódu z hlediska zmíněné paměťové bezpečnosti, a to jak v oblosti škálovatelnosti samotných nástrojů na velikosti úlohy, tak i především na snížení množství času, které je potřeba k vytvoření verifikační specifikace ještě před spuštěním samotného nástroje.

Jedním z přístupů, kterým toho dosahuje, je kompoziční přístup k verifikaci, kdy verifikovaný program není verifikován jako celek, ale jako soubor dílčích částí. Této kompozičnosti využívá i k tomu, že při inkrementálním vývoji stačí výsledky předchozí verifikace někam uložit, načež následující verifikační proces nemusí začínat na „zelené louce“. Z toho také plyne, že můžeme verifikovat nekompletní program. Důležitou vlastností zmíněného kompozičního přístupu také je takzvaná *graceful imprecision* [3], která nám umožňuje verifikovat i bez znalosti univerzální tvarové domény díky zavedení nepřesnosti (nad-aproximací).

[4] Volně přeloženo.

2.3.2 Princip

Samotná abdukce nabývá ve standardní logice tvaru:

$$A \wedge M \vdash G$$

kdy A je nějaký dostupný předpoklad, G je nějaký cíl a M je chybějící předpoklad, který se snažíme dovodit a který má nějaká předem definovaná pravidla – například o tom, že musí být minimální, že se musí jednat o konjunkci literálů apod. V případě abdukce, která je uvažována ve formální verifikaci vyplývající ze separační logiky popsané výše, se snažíme podobně odvodit:

$$A * M \vdash G$$

kde vzniká ten rozdíl, že A a M musí být paměťově separované [8]. Konečně, z této definice pak odvozujeme definici dotazu inference bi-abdukce:

$$A * \texttt{anti-frame} \vdash G * \texttt{frame}$$

kdy se snažíme odvodit nějaký anti-rámec (`anti-frame`) a rámec (`frame`) tak, aby byla daná formule platná.

2.3.3 Ukázka z článku

Pro ukázku, jak bi-abdukce pracuje na jednoduchém kódu, uvádíme příklad z článku [3], který se zaměřuje na jednoduchou strukturu:

```
struct node* q(struct node *y) { // Zisk. precondition: list(y)
    struct node *x, *z;
    x = malloc(sizeof(struct node)); x->tail = 0;
    z~= malloc(sizeof(struct node)); z->tail = 0;
    // Anti-ramec: list(y), Ramec: z|->0
    foo(x, y); // Ziskana postcondition: list(x)*z|->0
    // Anti-ramec: emp, Ramec: emp
    foo(x, z); // Ziskana postcondition: list(x)
    return x;
} // Ziskana postcondition: list(ret)
```

Důležitou poznámkou k tomuto kódu je, že pro funkci `foo` již byly odvozeny počáteční a výstupní předpoklady (buď použitím automatizované bi-abdukce, nebo ručně). Článek [3] konkrétně uvádí:

```
void foo(struct node *x, struct node *y){//SHRNUTI
    // Ziskana precondition: list(x) * list(y)
    // Ziskana postcondition: list(x)}
```

To nám říká, že na počátku víme, že existují nezávisle definované seznamy `x` a `y`, a že na konci existuje seznam `x`.

Na hlavním kódu pak můžeme vidět, jak autoři s pomocí bi-abdukce odvodili, že celá funkce `q` má jako vstupní předpoklad, že existuje seznam `y`. To zjistili přes odvození anti-rámce u prvního volání funkce `foo`, kdy podle předchozí definice je separátně definován jak první, tak druhý vstupní parametr. U druhého volání `foo` pak můžeme vidět, že žádný anti-rámec ani rámec odvozen nebyl (empty). Tato odvození autoři nazývají jako tzv. „malé specifikace“. Výsledný vyabdukovaný rámec celé funkce pak je `list(ret)`, který popisuje, že v návratové hodnotě je správně inicializovaný, správně zakončený seznam.

V tomto příkladu byla ukázána přesná bi-abdukce. Pokud by se ovšem vyskytl v daném kódu cyklus, pak nastává čas pro nad-aproximaci, kdy se objevují abstrakce, které vedou k tzv. „weakening a precondition“. To omezuje především vznik nekonečných sekvencí precondition a postcondition.

2.4 Pulse

Přístup, který by mohl časem nahradit zmíněnou bi-abdukci, je checker nazývaný Pulse (v článku Pulse-X), který vychází z takzvané Incorrectness Logic (možná lze nepěkně přeložit jako nesprávná/nekorektní logika). Ta dle článku [7] poskytuje především dvě klíčové vlastnosti. První je kompozičnost (Compositionality) – Jednotlivé komponenty systému jsou verifikovány nezávisle na sobě a celkový výsledek se spočítá až z vyřešených dílčích částí. To zvládá na základě podmíněných informací, které říkají, za jakých podmínek je daná procedura/funkce volána.

Druhou je tzv. pod-aproximace (Under-Approximation). Jak jsme si řekli v předchozích sekcích, nástroje se využívající separační logiku [4] a bi-abdukci [3] pracují s nad-aproximací, tj. s přístupem, kdy dokazujeme, že žádná chyba v systému není za cenu toho, že se můžeme setkat s neexistujícími chybami navíc. Ty právě zmíněné systémy řeší tak, že za použití určitých heuristik buď nahlásí chybu, nebo ji ignorují. Pulse na to jde obráceně, tj. pod-aproximovává zkoumaný prostor, ale zato když vrátí chybu, tak tam chyba skutečně je. Nicméně, tento fakt je částečně podrýván předchozí zmíněnou vlastností, kdy my předpokládáme nějaké podmínky, které ale nemusí nikdy nastat – například při hlášení chyby při přístupu na NULL, kdy kupříkladu při samotném volání funkce už můžeme testovat, že daný vstupní parametr NULL není.

V tomto autoři tohoto přístupu rozlišují tzv. manifestační (Manifest) a latentní (Latent) chyby. Manifestační chyby jsou nezávislé na kontextu, tj. chyba nastává nezávisle na vstupních proměnných. Tyto chyby jsou v rámci kompozitní analýzy obecně vždy hlášeny uživatelům daného nástroje. Oproti tomu latentní chyby nastávají pouze v daném kontextu a hlášeny jsou pouze v momentu, kdy se při nějaké další analýze kontextu podaří dokázat, že chyba je manifestační. Oproti předchozím verzím tohoto přístupu (nekorektní separační logice) navíc přidávají podporu hlášení tzv. memory-leaků [7].

3 Analýza nástroje na původních úlohách

V této a následující sekci se budeme zabývat experimenty, které uvádí sami autoři daných nástrojů a jejich modifikacemi. Pro experimentování s úlohami je využit počítač s operačním systémem Ubuntu ve verzi 22.04, procesor AMD Ryzen 5 5600, grafická karta NVIDIA RTX 3070 a 16 GB paměti RAM. Všechny použité kódy se nachází v adresáři examples, vždy ve vlastní složce. Jak experimentovaný Pulse, tak bi-abdukce jsou z Inferu verze 1.1.0, byť u verze Next má Pulse mnohem více funkcionalit[5]. Volání programu, pokud není uvedeno jinak, probíhá jako:

```
infer run --{pulse, biabduction} -- clang cesta/example.c
```

nebo v případě větších projektů jako:

```
infer run --{pulse, biabduction} -- make
```

V případě jazyka C++ jsme narazili na problém, že instalace Inferu na Ubuntu má problém s překladači jazyka C++. Po stejné instalaci na notebooku s Debianem jsme zjistili, že na Debianu funguje i verifikace programů napsaných v C++. Proto experimenty týkající se tohoto jazyka testujeme na notebooku s Debianem ve verzi 12.1, procesorem AMD Ryzen 5 3500U a 6 GB paměti RAM.

3.1 Experiment na příkladu z teorie

Jako první se podíváme na stejný kód, který jsme si uváděli u příkladu bi-abdukce 2.3.3. Zde bylo akorát potřeba naivně doimplementovat strukturu pro uzel a funkci `foo` jsem ponechal prázdnou.

[5] Předpokládáme, že verze Next je pouze pro vývojáře a jiné povolané osoby.

Bi-abdukce pak na tomto kódu hlásí 2 chyby, zatímco Pulse 4 – oba hlásí jako chybu null dereferenci, kdy, pokud malloc není úspěšný, na stejném řádku přistupujeme k proměnným, které mohou být null. Pulse hlásí to stejné, akorát duplicitně (2 null dereference a 2 nullptr dereference, což bude asi problém verze 1.1.0).

Tuto chybu jsem se pak pokusil opravit tak, že mezi alokování paměti a přiřazení jsem vložil kontrolu na null a v případě chyby vrátil null následujícího tvaru:

```
if (x == NULL){
    return NULL;
}
```

kde pro `z` jsem akorát přepsal proměnnou `x`. Infer je při použití bi-abdukce spokojen a nehlásí žádnou chybu. Zato Pulse už jeden problém hlásí, a to potenciální memory leak, kdy při druhé podmínce neuvolňujeme potenciálně alokovanou proměnnou `x`. Po tomto rozdílu jsem kontroloval, zdali bi-abdukce obsahuje hlášení memory leaků a ano, měla by je obsahovat, tudíž v tomto případě se ukázala bi-abdukce jako slabší než Pulse. To je proti původním předpokladům z teorie, jelikož Pulse by měl pod-aproximovávat a bi-abdukce nad-aproximovávat.

V tomto příkladu jsem tak doplnil uvolnění proměnné x a přešel k poslednímu experimentu, kdy jsem upravil funkci `foo` následujím způsobem:

```
void foo(struct node *x, struct node *y){
    x->tail = y;
}
```

Po této úpravě se v obou případech nic nezměnilo.

3.2 Příklad s cyklem

V článku [3] se nachází ještě k předtím uvedenému příkladu ilustrační příklad (lehce upraveno o deklarace proměnných) na inferenci pro cykly:

```
// Inferred pre: list(x)
void free_list(struct node *x) {
    struct node *t;
    while (x!=0) {
        t=x;
        x=x->tail;
        free(t);
}}
```

Na tomto příkladu se nezdařilo najít žádnou chybu ani jednomu ze dvou nástrojů. Pro rozšíření tohoto experimentu jsme definovali funkci `main` následujícím způsobem:

```
int main(){
    struct node *x = q(0);
    for(int i=0; i < 10; i++){
        add_to_list(&x, i);
    }
    if (x == NULL){
        return 0;
    }
    print_list(x)
    free_list(x);
    return 0;
}
```

kde funkce `add_to_list` je verifikovaná funkce, která přidává na začátek seznamu nový uzel s daty, a `print_list` je funkce na vypsání hodnot seznamu. I s tímto nastavením funkce pro uvolnění listu funguje. Nakonec jsme vyzkoušeli poslední experiment, a to odstranit funkci `free_list`, abychom porovnali schopnost nalézt memory leaky v programu. Zde selhaly oba dva programy a ani jeden program nenahlásil potenciální memory leak (byť tedy není potenciální, jako spíš skutečný). Na závěr jsem vytvořil funkci, která vytváří ze seznamu cyklický seznam, tj. propojí poslední prvek seznamu s prvním. Kód:

```
void make_cycled_list(struct node **x) {
    struct node *t;
    t = *x;
    while ((*x)->tail!=0){
        (*x)=(*x)->tail;
    }
    (*x)->tail = t;
}
```

Volání tohoto kódu jsem umístil před tisknutí a čištění seznamu – při spuštění způsobí nekonečný cyklus tisknutí čísel, a pokud odstraním tisknutí, tak program hlásí chybu dvojnásobného uvolnění stejné proměnné a je zabit signálem `SIGABRT`. Pokusil jsem se tuto chybu odhalit i Inferem, ale ani bi-abdukce, ani Pulse ji nezvládli odhalit.

3.3 Úlohy ze SV-Compu

Článek týkající se nástroje Pulse-X [7] ve svých experimentech vychází z reálných kusů kódu, kdy pracuje například s OpenSSL, kde je ale instalace daného nástroje poměrně komplexní. Z důvodu demonstrace problémů tak v této podsekci využijeme úloh poskytovaných v soutěži SV-COMP z roku 2023.

První potíží v těchto experimentech, na kterou jsme narazili, je absence přímé účasti Inferu v soutěži (tzv. hors concours), takže aplikace byla komplikovaná. Naštěstí se podařilo najít wrapper [6] na tento nástroj, který se právě snaží začlenit Meta Infer do soutěže. V rámci skriptu jsme bohužel nezjistili, jak vypnout jeden či druhý checker (volá jak Pulse, tak bi-abdukci), takže zmíněné úlohy budou řešeny obojím. U úloh jsme volili vlastnost `valid-memsafety`, které se nachází podadresáři `properties` u úloh v jazyce C. Narazili jsme ještě na `valid-memcleanup`, ale ten se nám nepodařilo zprovoznit (spouštěcí skript jej není schopen identifikovat).

3.3.1 Dvojitá dealokace paměti

První úlohou, kterou jsme zvolili do této dokumentace, se stává úloha z adresáře memsafety-ext3, která obsahuje jednoduchou dvojitou dealokaci paměti. Kód vypadá následovně:

```
void freePointer(int* p) {
    free(p);
}
int main(void) {
    int* p = malloc(10 * sizeof(int));
    freePointer(p);
    p[0] = 1;
    return 0;
}
```

V kódu můžeme vidět, že uvolnění paměti je schováno ve funkci, což nakonec nedělalo problém ani bi-abdukci, ani pulse. Oba hlásí víceméně shodně:

```
Found 3 issues
  Issue Type(ISSUED_TYPE_ID): #
  Use After Free(USE_AFTER_FREE): 1
  Null Dereference(NULL_DEREFERENCE): 1
  Nullptr Dereference(NULLPTR_DEREFERENCE): 1
```

kde opět můžeme vidět problém, který má Infer ve svých výpisech, tj. duplicitní nahlášenou chybu (Null Dereference a Nullptr dereference).

3.3.2 Nesprávná konverze polí

Úlohu, kterou naopak ani Pulse, ani bi-abdukce správně nezvládli, je špatná práce s pamětí v následující úloze:

```
int areNatural(int *numbers){
    int i = 0;
    while(i < 10) {
        if(numbers[i] <= 0){
            return 0;
        }
        i++;
    }
    return 1;
}
int main(void) {
    char numbers[] = {0,1,2,3,4,5,6,7,8,9};
    // numbers is array of chars, but should be integers
    int result = areNatural((int*) numbers);
    return 0;
}
```

kde problém nastává při konverzi pole charů na pole intů. Pokud si vytiskneme výsledek, pak zjistíme, že program hlásí 0, ale správně bychom očekávali 1 (hned druhé číslo v poli numbers je interpretováno po spuštění jako -19964863920). Infer v tomto případě žádnou chybu nehlásí.

3.3.3 Komplexnější práce s pamětí – bftdp

V rámci rozsáhlého archivu úloh soutěže SV-COMP jsme narazili na adresář plný komplexnějších úloh sestávajících z nástroje pro unixový FTP server bftpd. Zde jsou tři delší kódy, kdy u prvních dvou bychom neměli najít žádnou chybu a u třetího bychom měli narazit na chybu (u druhého byla špatně vlastnost, kterou se nepodařilo rozchodit). Celý kód se nachází v rámci archivu SV-COMPu se nachází ve složce `./c/memsafety-bftpd`.

Jak jsme si tedy řekli, tak u prvních dvou úloh bychom neměli narazit dle specifikace na chybu a ani jsme nenarazili. U třetího bychom měli narazit na chybu v případě úseku:

```
void c1() {
    char *x = (char *)malloc(sizeof(char));
    if(!x) {
    // out of memory
    return;
    }
    if(global) {
        // free(global); // memory-leak if c1 is executed twice
```

```
    }
    global = x;
    state = STATE_2;
}
```

Jak vidíme, tak autoři zde smazali uvolnění paměti z globální proměnné `global`, kdy v případě druhého spuštění kódu bychom měli uvolnit jeho původní obsah. Nicméně, tuto chybu nezvládl najít žádný z obou spuštěných checkerů, kdy problém způsobuje právě zmíněná globální proměnná.

3.3.4 Úlohy z adresáře memsafety

Jako další úlohy jsme zvolili úlohy z adresáře memsatefy. Jako první můžeme vidět následující kód:

```
int main ()
{
    n = BLOCK_SIZE;
    a = malloc (n * sizeof(*a));
    b = malloc (n * sizeof(*b));
    *b++ = 0;
    foo ();
    if (b[-1])
    { free(a); free(b); } /* invalid free (b was iterated) */
    else
    { free(a); free(b); } /* ditto */
    return 0;
}
```

Pokud se tento kód pokusíme spustit, pak narazíme na jednoduchý problém, kdy havarujeme na uvolnění paměti na příkazu `free`, což je i cílem hledání v této úloze. Infer skutečně chybu najde, bohužel ne tu správnou – stěžuje si na potenciální NULL dereferenci, pokud neověříme, že `b` bylo skutečně alokováno a na potenciální memory leak u téže proměnné. Nesprávné dealokace si Infer nevšímá a chybu nehlásí. Při úpravě z druhé verze tohoto kódu, kdy je druhé volání `free(b);` upraveno na `free(b-1);` dochází k hlášení stejných chyb, tedy i memory leaku, ke kterému zde nedochází.

U dalších testů jsme narazili na podobné závěry – Infer mnohdy chybu hlásí, ale jinde, než kde to úloha očekává. Tam, kde by měl najít null dereferenci, tam najde memory leak. Kde má najít memory leak, tam si stěžuje na dvojité uvolnění paměti v podmínce, která není dosažitelná. Občas se ale trefí, například v úloze:

```
int main()
{
    union {
        void *p0;
        struct {
            char c[2];
            void *p1;
            void *p2;
        } str;
    } data;
    // alloc 37B on heap
    data.p0 = malloc(37U);
    // this should be fine
```

```
    data.str.p2 = &data;
    // this introduces a memleak
    data.str.c[1] = sizeof data.str.p1;
    return 0;
}
```

správně odhalí memleak. Přesto spíše považujeme za úspěch, když řekne správný a očekávaný verdikt a obecně mu hodně dělají problém činnosti, které se dějí v podmínkách, které buď nemohou nikdy nastat, a nebo naopak nastávají vždy.

3.4 Závěr převzatých experimentů

Na převzatých experimentech se ukázalo, že, alespoň dle první úlohy, Pulse má o něco vyšší úspěšnost, co se týká verifikace memory leaků. Na úlohách ze SV-COMPu jsme si pak ukázali poměrně širokých slabin Inferu, kdy se mu nepodařilo najít ani některé základnější chyby, především v nevalidní konverzi, dvojitém uvolnění paměti a činnostech v podmínkách. Nicméně, v tomto případě může být problematický i převzatý wrapper [6], který jednak neumožnil použití druhého souboru s verifikovanými vlastnostmi a potenciální chyby v něm samém. Z tohoto důvodu jsme u úloh, které šly triviálně přeložit, aplikovali infer zvlášť. Žádná další zjištění/opravy jsme ale neobjevili.

4 Vlastní experimenty

V této sekci si ukážeme výsledky nástroje na vlastních experimentech, většinou na školních projektech, ale i některých open source programech.

4.1 IZP projekty

Jedním z prvních věcí, se kterými musí studenti FITu bojovat, jsou projekty v předmětu IZP. V této sekci si ukážeme experimenty na nich.

4.1.1 Projekt 1

Projekt 1 se týkal tvorby jednoduchého číselníku. V tom se nachází i lineární seznam pro správu telefonního seznamu, tudíž je zde prostor pro verifikaci paměti. Co se ale nakonec týká výsledků samotné verifikace, tak ta pouze poukázala, a to jak v případě Pulse, tak i za využití bi-abdukce, pouze na jednu neinicializovanou proměnnou v podmínce.

4.1.2 Projekt 2

Projekt 2 jsme měli za úkol řešit výpočet pracovního bodu diody s pomocí bisekce. Zde se jak při využití biabdukce, tak při využití Pulse nepodařilo najít žádnou chybu.

4.1.3 Projekt 3

V projektu 3 jsme měli řešit průzkum labyrintu. Tento projekt autorovi příliš nefungoval a i výsledek analýzy s nástrojem Infer potvrzuje nepříliš vysokou kvalitu kódu[6]. Ve výpisu se nachází následující seznam chyb:

- Dead Store(DEAD_STORE): 4

[6]Na obranu autora je nutné uvést, že v době tvorby projektu se několik týdnů potýkal se zápalem plic.

- Uninitialized Value(UNINITIALIZED_VALUE): 1
- Resource Leak(RESOURCE_LEAK): 1

Pokud začneme neinicializovanou proměnnou, tak tento problém se týká funkce pro pohyb v bludišti. Zde se nachází dvě úrovně podmínek – ty zajišťují, že se vybere správný směr cesty. Vypadají následovně:

```
if((*r+*s)%2==0) // Seznam moznosti
{
    if(hranice%3==0){...}
    else if(hranice%3==1){...}
    else if(hranice%3==2){...}
}
else
{
    if(hranice%3==0){...}
    else if(hranice%3==1){...}
    else if(hranice%3==2){...}
}
```

přičemž v každém z míst, kde jsou tři tečky, je první přiřazení do proměnné pro směr cesty. Jelikož jde o kompletní vyjmenování možností, tak by měl být vždy směr inicializován, tudíž se dá označit chyba za false alarm. Na druhou stranu, upozorňuje to na ne úplně dobrou programátorskou praktiku, kterou student prvního semestru na VUT FIT může dělat.

Čtyři chyby odkazující na „mrtvé uložení“ možná odhalují důvod nepříliš velkého počet získaných bodů, jelikož zde dochází opakovaně k volání:

```
Ruka(&m, r--, s--, 0);
```

Problémem je, že toto volání zahajuje hlavní proceduru programu a s proměnnými `r` ani `s` se nadále nepracuje. Bohužel se nám nepodařilo dohledat zadávanou mapu ani očekávaný tvar vstupních parametrů, jelikož Wiki stránky se zadáním z autorova ročníku již nejsou k dispozici. Tato chyba tak možná je a možná není kritického rázu.

Poslední zmíněná chyba ukazuje na špatnou práci se vstupními soubory, kdy byl přístup k souboru pouze otevřen, použit a dále již pak neuzavřen. Chyba je to středně závažná.

4.2 IOS projekt 2

Dalším projektem napsaným v C byl projekt pro řešení úlohy soudce a migrantů s pomocí semaforů. Na této úloze tak demonstrujeme funkčnost Meta Infer na programech s paralelně spuštěnými procesy.

První viditelnou změnou oproti předchozím experimentům je čas verifikace. Zatímco všechny předchozí úlohy dojely v rámci vteřiny až dvou, zde se pohybujeme zhruba na úrovni čtrnácti vteřin, a to jak v případě Pulse, tak v případě bi-abdukce. S pomocí Inferu se nám pak podařilo nalézt celkem 5 chyb, jedná se o 4 chyby typu DEAD_STORE a 1 neinicializovaná proměnná.

Problém s neinicializovanou proměnnou odkazuje na typický problém při práci s více procesy, kdy hodnota `pid2` byla inicialiována pouze v případě, že se nejednalo o úvodní generátor. Na začátku totiž vznikají dva procesy – jeden pro generátor migrantů druhý pro hlavní proces, který se stará o vygenerování soudce a následně čeká na dokončení všech operací. Pouze v druhém případě byla tato hodnota inicializována. Program i s touto chybou bez problémů fungoval.

Problémy s prvními dvěma mrtvými přiřazeními se týkají špatné práce s vracením chybových hodnot celého programu – program pokud dojede, tak automaticky vrací konstantu

EXIT_SUCCESS, přičemž původní zamýšlený plán bylo vracet chybovou hodnotu dle výsledku operací. Pokud tak někdo uprostřed selhal, pak přepsal hodnotu v proměnné a ta původně měla být vrácena. Další dvě mrtvá přiřazení jsou podobného principu, kdy se měla vracet návratová hodnota čekání procesu na ostatní procesy.

To, co nenašel ani Pulse, ani bi-abdukce, ani překladač a pro štěstí autora ani hodnotící projektu, je závažná chyba, kvůli které nešlo projekt na některých zařízeních zkompilovat a mohl tak být hodnocen celkovým počtem 0 bodů (reálně měl plný počet). Viz následující kód:

```
int init()
{
    // Otevreni a nastaveni souboru pro tisk
    if(out == NULL) return -1;
    // Alokovani a inicializace sdilenych dat a promennych
    NC=mmap(...);
    // Dalsi alokace a inicializace
    judgesem=sem_open(...); // Otevirani semaforu.
    if(judgesem == SEM_FAILED) // Test otevreni semaforu
    {
    return -1;
    }
    immsem=sem_open(...);
    if(immsem == SEM_FAILED)
    {
    return -1;
    }
    // Dalsi inicializace semaforu
    // Zde chybi return 0;
}
```

Problém je spojen s překladačem gcc, který by měl hlásit chybu při kompilování funkce, která inicializuje všechny globální semafory. Ta, v případě, že selže, vrací chybovou hodnotu -1. V případě, že program funguje, funkce nevrací nic (respektive z pohledu jazyka symbolických instrukcí vrací poslední hodnotu v čítači EAX). To je poměrně závažný problém a překladač by na základě nastavených přepínačů měl zakázat kompilaci programu. Přesto chyba prošla přes tehdejšího autora, hodnotícího a nyní i verifikační nástroj Meta Infer[7].

4.3 IPK projekt 2

Zadáním druhého projektu bylo vytvořit sniffer packetů v jazyku C/C++, který měl za úkol naslouchat na daném zařízení na všech (vybraných) portech a vypisovat tak procházející síťový provoz. Nalezeny byly dvě chyby, a to první typického charakteru zápisu do NULL v případě, že nevyjde malloc (neexistuje kontrolní podmínka). Druhá hovoří o neinicializované proměnné. Zde se jedná o falešný alarm (a to jak v případě pulse, tak bi-abdukce), jelikož k dané operaci nelze přistoupit bez zavolání programu s přepínačem `-p`, který má povinně i vstupní číselný argument.

4.4 IMS projekt

Jedním z větších projektů, kterými se v této práci zabýváme, je IMS projekt (1300 řádků v C++). Zde se jedná o týmový projekt bakalářů Davida Hudáka a Daniela Karáska, který se zaměřoval na

[7]Na chybu se autorovi náhodou povedlo přijít až nějakou dobu po odevzdání, kdy kompiloval projekt na jiném zařízení, které kompilaci zablokovalo.

téma *Výhledy epidemie Covidu a vliv očkování*. Tento model byl založen na principu celulárních automatů (zde je významná práce s pamětí) a byl konzultován s experty v oboru z ÚZISu při online konzultaci, kdy výsledný projekt byl hodnocen 23 body z 20 (dle informací z okolí se jednalo o nejlepší výsledek toho semestru). Tento projekt navíc kromě hodně bodů neobsahuje dle Inferu žádné chyby. Verifikace s bi-abdukcí zabrala kolem 7 a půl vteřiny a s Pulsem něco málo přes 8.

4.5 ISA projekt

V tomto projektu bylo zadáním zkoumat zadaný program, který prováděl komunikaci v síti. Naším úkolem bylo jej s pomocí Wiresharku sledovat a následně ho reverse engineerovat. V tomto kódu navíc byla využita stažená knihovna pro implementaci base64 z GitHubu [8]. V tomto spíše menším projektu (část projektu byla v jazyce Lua pro analyzátor zkoumaného protokolu).

4.6 IAL projekty

V předmětu IAL (Algoritmy) bylo 6 menších domácích úkolů zaměřených na různé dynamické datové struktury a operace s nimi, tedy poměrně optimální cíl tohoto projektu. První dvě úlohy v obou nástrojích dopadly dobře, u třetí jsme už narazili na problém. Bi-abdukce v tomto případě hlásí 4 chyby, Pulse 6 (ovšem zde se jedná o duplicitní chyby). Bohužel, ve všech případech se jedná o neúmyslné falešné alarmy. Podívejme se na následující kód:

```
// Kod pred
    struct tDLElem *novy=malloc(sizeof(struct tDLElem));
        if(novy==NULL)
    {
        DLError();
    }
    novy->data=val;
// Kod po
```

Zde můžeme vidět, že standardně alokujeme paměť do proměnné `novy`, provádíme kontrolu a následně k dané proměnné přistupujeme. Zadávající projektu v případě, že se nezdaří alokovat paměť, požadují volat funkci `DLError`, která zřejmě hodnotícího má informovat o faktu, že není chyba v kódu, nýbrž nejspíš došla paměť. Infer tak hlásí chybu, ale z praktického hlediska tam není.

Ve čtvrté domácí úloze jsme s Inferem narazili na chybu v testovacím skriptu opět ve formě netestování alokované proměnné na NULL. Dále se v něm vyskytuje `DEAD_store` chyba, jež spočívá v prázdné operaci, u které jsme nezjistili, proč ji tam autor umístil.

U páté úlohy zaměřené na binární vyhledávací stromy se opět ukázal silnější nástroj Pulse, který odhalil 3 chyby navíc, všechny typu memory leak (oba pak nalezli standardní null dereferenci). V jednom případě je chyba zjevná, kdy autor nejspíše opomenul po práci se stromem daný strom dealokovat. Viz:

```
    printf("[TEST07]\n");
        printf("Vlozime dalsi prvky a vytvorime... \n");
        printf("... \n");
        BSTInsert(&TempTree,'D',4);
        BSTInsert(&TempTree,'L',12);
        BSTInsert(&TempTree,'B',2);
        BSTInsert(&TempTree,'F',6);
        BSTInsert(&TempTree,'J',10);
        //...
```

[8] Viz: https://github.com/ReneNyffenegger/cpp-base64

```
BSTInsert(&TempTree,'M',13);
BSTInsert(&TempTree,'O',16);
Print_tree(TempTree);
printf("[TEST08]\n");
```

Zde můžeme vidět, že po testu 7 hned nastává test 8. V druhém případě se jedná o špatnou interpretaci příkazu `strcat`, která má do prvního argumentu funkce přikonkatenovat řetězec z druhého argumentu. V tomto ohledu autor špatně volá funkci, viz:

```
suf2 = strcat(suf2, "  |");
```

Zde by se dle příkladu z manuálových stránek měla funkce správně volat bez práce s návratovou hodnotou. Sice by se měla vracet stejná hodnota ukazatele `destination` (současně 1. argument funkce), ale je možné, že při konkrétní implementaci funkce může dojít k vrácení jiné hodnoty.

Poslední úloha vykazuje úplně stejné chyby jako úloha 5. Opět zde při testování nedochází k uvolnění binárního stromu a opět se zde pracuje špatně s funkcí `strcat`.

4.7 IFJ projekt

Největším projektem každého studenta FITu byl a asi i dlouho bude projekt v předmětu IFJ. Autory tohoto projektu jsou bakaláři Filip Osvald, Daniel Karásek a David Hudák a společně napsali bezmála 5 tisíc řádků. V tomto případě opět analyzátor Pulse narazil na 2 memory leaky navíc. Jinak je výstup oproti bi-abdukci, až na obligátní duplicitu v null dereferenci, víceméně stejný. Celkový výčet chyb:

```
Issue Type(ISSUED_TYPE_ID): #
Uninitialized Value(UNINITIALIZED_VALUE): 14
Null Dereference(NULL_DEREFERENCE): 7
Memory Leak(MEMORY_LEAK): 2
Nullptr Dereference(NULLPTR_DEREFERENCE): 1 // duplicita
Dead Store(DEAD_STORE): 1
```

Pokud půjdeme postupně, pak narážíme na spoustu neinicializovaných proměnných – jedna várka se týká syntaktické analýzy. Zde jsme se museli potýkat s problémem deklarací proměnných uvnitř cyklů, což nám tehdy došlo až při přidávání sémantické analýzy. Kvůli tomu jsme přidali dva průchody rekurzivního sestupu, kdy se první soustředil pouze na inicializaci hashovací tabulky pro použité pro tabulku symbolů. Problém je, že tato chyba je falešný alarm. Podívejme se na delší následující kód:

```
int returnTypesList(bool firstSearch, int **outList){
    t_Token *t = getToken(true);
    int *typeList;
    int tmp_size;
    int tmp_curPos;
    // ...
    if (firstSearch)
    {
      typeList = malloc(sizeof(int)*5);
      tmp_size = 5;
      tmp_curPos = 0;
      typeList[0] = -1;
      // ...
    }
    // ...
```

```
    do
    {
      if(...))
      {
        if (firstSearch){
          if(tmp_curPos == tmp_size -1) CHYBA HLASENA ZDE
          {
            tmp_size *= 2; // CHYBA HLASENA ZDE
            typeList = realloc(typeList, tmp_size*sizeof(int));
            // ...
          }
          // ...
        else{...}
        // Pokracovani kodu a predevsim cyklu.
}
```

Inferu vadí především práce s `tmp_curPos` a `tmp_size`, které dle něj nejsou inicializované. Problém nastává v tom, že tato chyba není reálná, jelikož s danými proměnnými se pracuje pouze v podmínkách `if (firstSearch)`, tj. tato situace nikdy nenastane. Pokud program chce číst z dané proměnné, pak to může udělat jenom za daných podmínek, za kterých je proměnná vždy inicializovaná. Nejedná se tak možná o dobrou programátorskou praktiku, nicméně se jedná o falešný poplach.

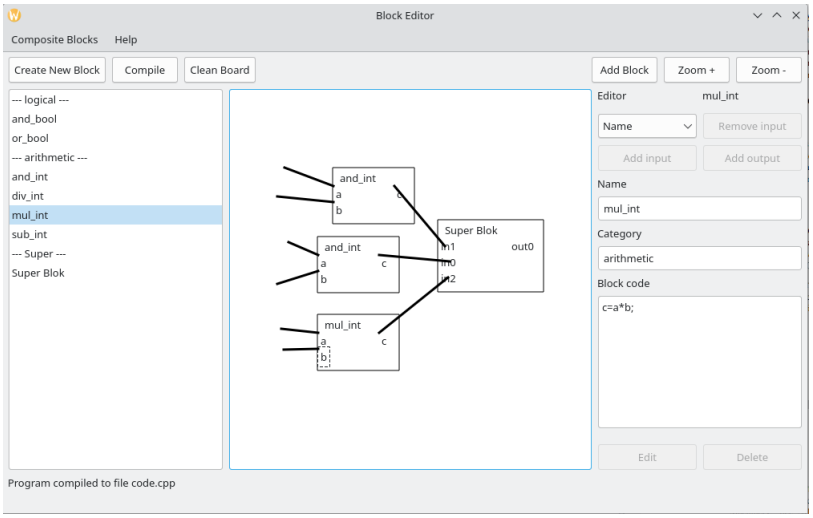
Následně se jedná o výčet chyb, které již byly zmíněny v předchozích případech – absence kontroly na null po každém alokování paměti, opomenuté uvolnění paměti již alokovaných proměnných v případě chyby po alokování paměti. Co je relativně nové, je neinicializace některých položek struktury. Opět, při přidávání sémantické analýzy byly do zásobníku využitého při precedenční analýze přidány nějaké pomocné proměnné pro informování překladače o užitečných informacích, jako je úroveň zanoření kódu v podmínce. Tato data nejsou vždy užitečná, takže bez újmy na funkčnosti mohly být vynechány a program nezhavaroval. Na stranu druhou, často se předává právě celá tato struktura, takže v programu propagujeme neinicializovanou hodnotu, která může v případě rozšiřování programu způsobit obtíže. Poslední chybou je opět tzv. `DEAD_STORE`, který se týká nevyužité proměnné.

4.8 ICP projekt

Abychom nezůstali pouze u konzolových aplikací, vyzkoušíme v této části verifikovat projekt z předmětu ICP, který zahrnuje aplikaci s uživatelským rozhraním v Qt. V případě této verifikace jsme překvapivě nenarazili ani na jeden bug. V tomto případě se jedná o překvapivou informaci, jelikož tento projekt bakalářů Davida Hudáka a Štefana Mračny prošel velmi těsně na 50 bodů ze 100, a to po opravě nejzásadnějších chyb. Aplikace dokonce hned při spuštění při stisknutí jednoho konkrétního tlačítka občas nedeterministicky spadne. Ukázku z této aplikace můžeme vidět na obrázku 2, kdy ono nebezpečné tlačítko je Add input (aktuálně správně šedé, po spuštění je klikatelné a občas shodí program). Žádné chyby nenašel ani Pulse, a to přesto, že největší komplikací v tomto projektu byla náročná práce s pamětí s ohledem na tvorbu dynamických oken a uchovávání bloků dat.

4.9 Vlastní projekt na posilované učení v C++

V rámci experimentů jsme vyzkoušeli verifikaci menšího projektu pracujícího s dynamickou mapou a agenty, kteří hrají na daném hracím poli. V tomto projektu se nepovedlo najít ani jeden bug, a to jak s nástrojem Pulse, tak bi-abdukcí.



Obrázek 2: Naše nepříliš povedená aplikace.

5 Závěr experimentů

Na uvedených úlohách v předchozích dvou sekcích jsme si ukázali, co zvládnou, a co naopak nezvládnou checkery v nástroji Meta Infer využívající bi-abdukci a Pulse. V některých případech se podařilo narazit na vícero chyb, v některých případech se naopak povedlo i zásadní chyby minout. Co se týká časových rozdílů, tak rozdíly byly pouze zanedbatelné a ve verzi Inferu 1.1.0 docházelo k velmi mírnému náskoku bi-abdukce. Přesto se jednalo ve všech případech o přijatelné časy a všechny úlohy se zdařilo verifikovat.

Co se týká zaznamenaných chyb, tak oproti očekáváním jsme jich našli spíše pod očekávání a například na IOS projektu a ICP projektu jsme nenašli chyby paměťového charakteru, které by Meta Infer měl odhalit. Stejně tak jsme narazili na několik falešných hlášení, které by mohly být chybami, ale nikdy nemohou nastat – ať už se s nějakou proměnnou pracuje pouze za splnění daných podmínek, nebo o chybu, kterou nenašel ani překladač v případě opomenuté návratové hodnoty. Především pak ze soutěžních úloh ze soutěže SV-COMP si odnášíme nikoli dobrý dojem. Celkově se nám také jeví, že v případě C++ se nám podařilo najít závratně méně chyb než v případě jazyka C, což může implikovat menší schopnosti analyzovat tento jazyk. Analýza programů je tak spíše mělčího charakteru a může pomoci při raných fázích vývoje, kdy si můžeme představit, že při kompilaci současně spustím verifikaci nějaké aktuálně vyvinuté části kódu. V té je Meta Infer schopen najít základní chyby, které mohou nevyhnutelně vést k pádu programu.

Obsah

Reference

[1] BERDINE, J., CALCAGNO, C. a O'HEARN, P. W. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In: BOER, F. S. de, BONSANGUE, M. M., GRAF, S. a ROEVER, W.-P. de, ed. *Formal Methods for Components and Objects*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, s. 115–137. ISBN 978-3-540-36750-5.

[2] CALCAGNO, C., DISTEFANO, D., DUBREIL, J., GABI, D., HOOIMEIJER, P. et al. Moving Fast with Software Verification. In: HAVELUND, K., HOLZMANN, G. a JOSHI, R., ed. *NASA Formal Methods*. Cham: Springer International Publishing, 2015, s. 3–11. ISBN 978-3-319-17524-9.

[3] CALCAGNO, C., DISTEFANO, D., O'HEARN, P. W. a YANG, H. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM*. New York, NY, USA: Association for Computing Machinery. dec 2011, sv. 58, č. 6. DOI: 10.1145/2049697.2049700. ISSN 0004-5411. Dostupné z: https://doi.org/10.1145/2049697.2049700.

[4] DISTEFANO, D., O'HEARN, P. W. a YANG, H. A Local Shape Analysis Based on Separation Logic. In: HERMANNS, H. a PALSBERG, J., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, s. 287–302. ISBN 978-3-540-33057-8.

[5] FACEBOOK, I. *Meta Infer Online Documentation*. 2023. Dostupné z: https://fbinfer.com/docs.

[6] KETTL, M. a LEMBERGER, T. *FBInfer in SV-COMP*. Zenodo, listopad 2023. DOI: 10.5281/zenodo.10079740. Dostupné z: https://doi.org/10.5281/zenodo.10079740.

[7] LE, Q. L., RAAD, A., VILLARD, J., BERDINE, J., DREYER, D. et al. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proc. ACM Program. Lang.* New York, NY, USA: Association for Computing Machinery. apr 2022, sv. 6, OOPSLA1. DOI: 10.1145/3527325. Dostupné z: https://doi.org/10.1145/3527325.

[8] O'HEARN, P., REYNOLDS, J. a YANG, H. Local Reasoning about Programs that Alter Data Structures. In: FRIBOURG, L., ed. *Computer Science Logic*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, s. 1–19. ISBN 978-3-540-44802-0.

[9] PEIRCE, C. S. *Collected Papers of Charles Sanders Peirce*. Harvard University Press, 1958. ISBN 0674138031.