BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

PRACTICAL APPLICATION OF FACEBOOK INFER ON SYSTEMS CODE

PRAKTICKÁ APLIKACE FACEBOOK INFER NA SYSTÉMOVÝ KÓD

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **TOMÁŠ BERÁNEK**
AUTOR PRÁCE

SUPERVISOR **Prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2020/2021

Bachelor's Thesis Specification

Student: **Beránek Tomáš**
Programme: Information Technology
Title: **Practical Application of Facebook Infer on Systems Code**
Category: Software analysis and testing

Assignment:

1. Get acquainted with principles of static analysis based on abstract interpretation, with the Facebook Infer framework, and the different analysis plugins available for Infer.
2. Get acquainted with the csmock tool used in Red Hat for running various static analysers.
3. Propose a way for applying Facebook Infer on systems code through the csmock system.
4. Identify open-source software projects centred around Linux that Infer could successfully handle and fine-tune usage of Infer for such projects.
5. Apply Facebook Infer on the chosen projects, gather and discuss the results.

Recommended literature:

1. Rival, X., Yi, K.: Introduction to Static Analysis: An Abstract Interpretation Perspective. MIT Press, 2020.
2. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling Static Analyses at Facebook. Communications of the ACM 62(8):62-70, ACM Press, 2019.
3. Blackshear, S., Gorogiannis, N., O'Hearn, P. W., Sergey, I.: RacerD: Compositional Static Race Detection. In: Proc. of OOPSLA'18, PACMPL 2(OOPSLA):144:1-144:28, 2018.
4. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving Fast with Software Verification. In: Proc. of NFM'15, LNCS 9058, Springer, 2015.
5. Dudka, K.: Fully Automated Static Analysis of Fedora Packages. FLOCK 2014. Available online at: https://kdudka.fedorapeople.org/static-analysis-flock2014.pdf.
6. Harmim, D.: Static Analysis Using Facebook Infer to Find Atomicity Violations. Bachelor thesis, Brno University of Technology, 2019.
7. Marcin, V.: Static Analysis Using Facebook Infer Focused on Deadlock Detection. Bachelor thesis, Brno University of Technology, 2019.

Requirements for the first semester:

- Items 1, 2, and at least to some degree item 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Vojnar Tomáš, prof. Ing., Ph.D.**
Consultant: Dudka Kamil, Ing., RedHatCZ
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 12, 2021
Approval date: November 11, 2020

Abstract

Static analysis is nowadays often used in the development process to find defects in the produced software. Although static analysis tools can effectively find bugs in software with millions of lines of code, they have also some disadvantages. The main disadvantages are the difficulty to deploy the chosen tool on the given project, high numbers of false reports, and the time and space requirements. This thesis focuses on mitigating these negative features of the Facebook Infer tool mainly for the context of using it to analyse Linux utilities shipped as SRPM packages. To simplify its deployment, an Infer plugin has been created for the csmock tool, which allows static analysers to run automatically on packages for CentOS or Fedora. To reduce the number of false reports, a filter has been created, which filters Infer's output according to several proposed heuristics based on experience obtained by analysing the reports produced by Infer. The filter has been also included into the csmock plugin and tested on a number of packages. On the analysed packages, the filter was able to remove 60 % of false reports with a loss of 2.5 % of real defects. The time required to run the analysis can be reduced by using incremental analysis. Shortcomings of the incremental analysis provided implicitly by Infer were experimentally found, so this thesis also describes the creation of a wrapper for Infer, which replaces the incremental analysis in Infer.

Abstrakt

Statická analýza je dnes často využívána ve vývojovém procesu pro hledání defektů v produkovaném softwaru. I když nástroje na statickou analýzu dokáží hledat defekty v softwarech o miliónech řádků kódu, mají také řadu nevýhod. Hlavními nevýhodami jsou náročnost nasazení nástroj na vyvíjený projekt, vysoký počet falešných hlášení a časové i paměťové požadavky. Tato práce se zaměřuje na zmírnění těchto negativních vlastností u nástroje Facebook Infer, zejména pro analýzu Linuxových nástrojů v podobě SRPM balíčků. Pro zjednodušení nasazení byl vytvořen modul pro nástroj csmock, který umožňuje automaticky spouštět statické analyzátory nad balíčky pro CentOS a Fedoru. Pro snížení počtu falešných hlášení byl vytvořen filtr, který filtruje výstup Inferu podle heuristik, které byly navrženy na základě zkušeností získaných kontrolou hlášení z Inferu. Filtr byl také zapojen do modulu pro csmock a otestován na řadě balíčků. Na analyzovaných balíčcích filtr dokázal odstranit 60 % falešných hlášení se ztrátou 2.5 % skutečných defektů. Doba potřebná pro běh analýzy může být zkrácena použitím inkrementální analýzy. U inkrementální analýzy Inferu byly experimentálně zjištěny nedostatky, proto se tato práce věnuje také vytvoření nástavby nad Inferem, která nahrazuje inkrementální analýzu v Inferu.

Keywords

static analysis, Facebook Infer, csmock, incremental analysis, automated analysis, SRPM package, integration of static analysers, false positive filtering, software analysis

Klíčová slova

statická analýza, Facebook Infer, csmock, inkrementální analýza, automatizovaná analýza, SRPM balíček, integrace statických analyzátorů, filtrování falešných hlášení, analýza softwaru

Reference

BERÁNEK, Tomáš. *Practical Application of Facebook Infer on Systems Code*. Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Prof. Ing. Tomáš Vojnar, Ph.D.

Rozšířený abstrakt

V dnešní době se klade stále větší důraz na bezchybný kód a to nejen v kritických oblastech jako letectví, kosmonautika nebo lékařství. Ale i v oblastech, kde chyba v aplikaci nepředstavuje závažná rizika, příkladem jsou mobilní aplikace nebo hry. Dnes existuje mnoho technik pro odhalování chyb. Dynamická analýza je dnes přítomna téměř v každém vývojovém procesu. A přesto, že její výsledky jsou velice cenným zdrojem pro odhalování chyb, tak nemůže kompletně pokrýt všechny cesty, zejména v dnes velmi komplexních softwarech. Manuální kontrola je velmi užitečná pro odhalování komplexních chyb, které není možné najít automatizovanými nástroji, ale je také velmi časově náročná. Statická analýza zmírňuje nedostatky obou těchto metod. Oproti dynamické analýze, statická analýza uvažuje všechny možné cesty v programu, nehledě na vzácnost jejich vykonání. Oproti manuální kontrole je statická analýza mnohem efektivnější. Avšak statická analýza má i své nedostatky. Mezi hlavní problémy patří vysoký počet falešných hlášení, náročnost nasadit statický analyzátor na vyvíjený software a škálovatelnost.

Pravě tyto nevýhody statických analyzátoru jsou hlavními překážkami při zasazování statické analýzy do vývojového procesu. Proto se tato práce věnuje zmírnění těchto tří negativních vlastností u nástroje Facebook Infer. Infer je volně dostupný nástroj pro vytváření vysoce škálovatelných, kompozičních, inkrementálních a interprocedurálních analýz. Nástroj je aktivně vyvíjen firmou Facebook s příspěvky od vývojářů z celého světa. Aktuálně je nástroj nasazen ve vývojovém procesu firmy Facebook, Amazon, Microsoft, Mozilla nebo Instagram. I když se jedná o framework, tak Infer obsahuje také řadu základních analýz. Tyto analýzy detekují úniky paměti, dereference prázdného ukazatele, přetečení proměnných, přístup za hranice pole, použití neinicializované proměnné, ale i synchronizační chyby typu deadlock nebo data race. Infer podporuje analýzu programů napsaných v jazycích C, C++, Objective C, C# a Java. Jednotlivé analýzy mohou být jazykové závislé. Aby Infer mohl zanalyzovat zdrojový kód musí mu být předány fungující překladové příkazy, které překládají soubory určené k analýze. Pomocí nich si Infer přeloží zdrojové soubory do své interní reprezentace, nad kterou je poté spuštěna analýza.

Pro zjednodušení nasazení nástroje byl vytvořen Infer modul pro nástroj csmock, který umožňuje automaticky spouštět statické analýzy nad SRPM balíčky pro operační systémy Fedora a CentOS. Csmock byl vytvořen a je udržován Kamilem Dudkou z firmy Red Hat. Infer modul musí být navržen tak, aby byl schopen se bezpečně napojit na překladový proces SRPM balíčku uvnitř uměle vytvořeného prostředí nástrojem csmock a extrahovat informace potřebné pro spuštění analýzy. Infer modul byl otestován na řadě SRPM balíčků určených pro operační systém Fedora. Analyzované balíčky obsahovaly systémové nástroje jako například zip, less, sed, grep, make a další. Výsledky na těchto nástrojích odhalily reálné chyby, které byly nahlášeny a některé již byly opraveny v nejnovějších verzích těchto nástrojů.

Problémem už již zasazené statické analýzy je, že jsou její výsledky často vývojáři ignorovány. Toto je způsobeno zejména velkým počtem falešných hlášení, které musí vývojáři projít, než je nalezena skutečná chyba. Pro zmírnění tohoto problému byl vytvořen filtr, který používá navržené heuristiky založené na zkušenostech z manuální kontroly hlášení vyprodukovaných nástrojem Infer.

Filtr je implementován jako Python skript, který postupně aplikuje implementované heuristiky na výstup z Inferu a odstraňuje hlášeni, která jsou podle navržených heuristik falešná. Filter je součástí Infer modulu pro nástroj csmock. Díky tomu jej bylo možné otestovat na řadě SRPM balíčků pro operační systém Fedora. Na analyzovaných balíčcích filtr dokázal odstranit přibližně 60 % falešných hlášení se ztrátou pouze 2.5 % skutečných chyb. Analýza byla provedena na více než 400 tisících řádcích kódu a bylo manuálně roztříděno přibližně 500 hlášení z Inferu podle jejich pravdivosti.

Pro nasazení statického analyzátoru do vývojového procesu s průběžnou integrací, musí být analyzátor schopen velmi dobře škálovat a hlásit výsledky řádově do desítek minut. Jelikož jsou dnešní softwary velmi rozsáhlé, tak by takovéto rychlosti nebylo možné dosáhnout bez inkrementální analýzy. Pokud analyzátor běží v inkrementálním režimu, tak analyzuje pouze ty části softwaru, do kterých mohla být zanesena chyba z provedené změny, místo analýzy celého softwaru. Nástroj Facebook Infer obsahuje inkrementální analýzu na úrovni jednotlivých funkcí. Avšak při experimentech s inkrementální analýzou v Inferu byla nalezena řada nedostatků. Infer na daných experimentech nedokázal správně identifikovat části kódu do kterých mohla být zanesena chyba ze změny. Pro opravení těchto nedostatků byla navržena nástavba nad Inferem, která identifikuje zasažené části kódu místo Inferu a poté spouští analýzu nad těmito částmi. Nástavba dokáže správně identifikovat části kódu k analýze, ale snižuje efektivnost inkrementální analýzy, protože je inkrementalita posunuta na úroveň souborů.

Redukování počtu falešných hlášení bude hlavním cílem budoucích vylepšení. Vzhledem k tomu, že manuální kontrola hlášení z Inferu je velmi časově náročná a také není garantováno, že bude možné na základě této kontroly navrhnout další heuristiky, tak se budoucí práce nebude zaměřovat na filtrování, ale na řazení. Konkrétně se plánuje zapojení strojového učení pro řazení hlášení podle jeho pravděpodobnosti na reálnou chybu. Dříve navržené heuristiky budou použity jako vstupy pro strojové učení spolu s dalšími metrikami výsledků analýz a metrikami zdrojových kódů. Tato budoucí práce bude probíhat ve spolupráci s IBM Research a Red Hat.

Infer modul pro nástroj csmock, filtr falešných hlášení i nástavba Inferu pro inkrementální analýzu jsou součástí projektu Arrowhead Tools (H2020 ECSEL). Zdrojové soubory všech částí této práce jsou volně dostupné v repozitářích na serveru GitHub.

Practical Application of Facebook Infer on Systems Code

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of prof. Ing. Tomáš Vojnar, Ph.D. Supplementary information was provided by Ing. Kamil Dudka from Red Hat. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Tomáš Beránek
May 19, 2021

Acknowledgements

I would like to thank my supervisor Tomáš Vojnar and my consultant Kamil Dudka from Red Hat for helpful suggestions about the Infer plugin for csmock and the thesis. Further, I would like to thank Nikos Gorogiannis from the Infer team in Facebook for detailed information about RacerD and incremental analysis in Infer. Lastly, I thank my colleagues Ondřej Pavela, Dominik Harmim and Tomáš Dacík for helpful discussions about Infer. Finally, I acknowledge the financial support of the H2020 ECSEL project Arrowhead Tools.

Contents

Chapter 1

Introduction

Nowadays, more and more emphasis is put on error-free code (or as error-free as possible). This tendency is not only limited to critical areas such as aviation, aerospace, or medicine, but it is visible also in areas where a bug in an application does not pose a critical risk, such as mobile applications or games. Today, there are many techniques for detecting bugs. *Dynamic analysis* is present in almost every development process today. And although its results are a very valuable resource for bug detection, it cannot completely cover all paths, especially in today's very complex software. *Manual code review* is very useful for detecting complex bugs that cannot be found by automated tools, but it is also very time-consuming. *Static analysis* mitigates the disadvantages of both of these methods. Unlike dynamic analysis, static analysis considers all possible paths in a program, regardless of the rarity of their execution. Compared to manual inspection, static analysis is much more efficient. However, static analysis also has its shortcomings. The main problems are the difficulty to deploy the chosen tool on the given project, high numbers of false reports, and its computational time and space requirements.

Therefore, this thesis focuses on mitigating these three disadvantages in the context of Facebook Infer – an open-source framework for static analysers – mainly in the context of using it to analyse Linux utilities shipped as SRPM packages. Infer is a tool for creating highly scalable, *compositional*, *incremental*, and *interprocedural* analyses. The tool is being actively developed by Facebook with contributions from developers around the world. Currently, the tool is deployed in the development process in Facebook, Amazon, Microsoft, Mozilla, or Instagram. While it is a framework for writing analyses, it offers several analyses for detecting various defects such as *memory leaks*, *invalid pointer dereferences*, *variable overflows*, *accesses beyond array boundaries*, *uses of uninitialized variables*, as well as synchronization errors like *deadlocks* or *data races*. Infer supports analysis of programs written in C, C++, Objective C, C#, and Java. Individual analyses can be language-dependent. In order for Infer to analyse the source code, working compilation commands that translate the files to be analysed must be passed to Infer. Using these compilation commands, Infer compiles the source files into its internal representation over which the analysis is then performed.

To simplify the deployment of the tool, as our first contribution, in the context of SRPM packages, we developed an Infer plugin for the csmock tool, which automatically runs various static analysers on SRPM packages for Fedora and CentOS operating systems. Csmock was created and is maintained by Kamil Dudka from Red Hat. The Infer plugin is designed

to securely connect to the SRPM package's build process inside the artificial environment created by csmock and extract the information needed to perform the analysis.

As our further contribution, we applied our plugin to several SRPM packages designed for the Fedora operating system. The analysed packages contained system tools such as `zip`, `less`, `sed`, `grep`, `make`, and others. Using our plugin, we have discovered real bugs that have been reported and some have already been fixed in the latest versions of these tools.

A problem with static analysis even if it is deployed in a development process is that its results are often ignored by developers. This is mainly due to the high number of false reports that developers have to go through before a real bug is found. To mitigate this problem, we developed a filter that aims to reduce the number of false reports. The filter is based on proposed *filtering heuristics* based on an experience obtained by analysing the reports produced by Infer.

The filter is implemented as a Python script that applies the implemented heuristics to the Infer's output and removes reports that are false according to the proposed heuristics. The filter is a part of the Infer plugin for the csmock tool. This allowed it to be tested on several SRPM packages for the Fedora operating system. On the analysed packages, the filter was able to remove approximately 60 % of false reports with a loss of only 2.5 % of real errors. The analysis was performed on more than 400 KLoC in total.

To deploy a static analyser in a *continuous integration* development process, the analyser must be able to report results in tens of minutes. If the analyser runs in an incremental way, it analyses only those parts of the project into which a bug could have been propagated from a change, which decreases its computational time and space requirements. Facebook Infer performs the *incremental analysis* at the function level. However, several shortcomings of the incremental analysis provided implicitly by Infer were experimentally found. In these experiments, Infer was unable to correctly identify parts of the code that may have been affected by a change. To fix these shortcomings, we developed a wrapper for Infer that was designed to identify the affected parts of the code instead of Infer and then run the Infer's analysis only on these parts. Our wrapper somewhat reduces the level of incrementality (from functions to files), but unlike the original solution it – to the best of our knowledge – correctly recognizes code that needs to be re-analysed.

Acknowledgment The development of Infer plugin for csmock has been discussed with Kamil Dudka, the author of csmock. The Infer plugin for csmock, the filter and the wrapper for the incremental analysis are a part of the H2020 ECSEL project Arrowhead Tools.

Structure of the thesis The rest of this thesis is structured as follows. Chapter 2 explains basic principles of static analysis and describes the tools used in the thesis – Facebook Infer and csmock. Chapter 3 discusses the design and implementation of the Infer plugin for csmock. Chapter 4 discusses the results of analyses with the Infer plugin

for csmock on SRPM packages. Chapter 5 describes the design and implementation of the Infer wrapper for incremental analysis. Chapter 6 discusses the results of plain Infer analyses on various projects. The chapter also describes what kind of code we found unsuitable for Infer. Chapter 7 concludes the thesis. In addition, there are four appendices. Appendix A lists the contents of the attached memory media. Appendix B serves as an installation and user manual for the Infer plugin for csmock. Appendix C provides detailed graphs with the results of analyses on SRPM packages. Appendix D shows an experiment that demonstrates the shortcomings in Infer's incremental analysis and shows the functionality of the Infer wrapper for the incremental analysis.

Chapter 2

Preliminaries

This chapter explains the basic principles of static analysis and its disadvantages, which this thesis tries to alleviate. This chapter also describes the static analyser used in this thesis – Facebook Infer. Lastly, the chapter describes the tool for automated analysis on SRPM packages – csmock, for which an Infer plugin has been created.

2.1 Static Analysis

Static analysis is a method for detecting various types of defects in computer programs. Unlike *dynamic analysis*, static analysis does not run the programs or at least not with their original semantics. The analysis is performed only on the source code of the program. Static analysis – at least in some of its forms – is able to analyse millions of LoC[2] in a very short time. A manual code review cannot ever achieve such efficiency. However, manual code review still remains useful since a human can see complex errors beyond the reach of automated tools – and it is also often required to check whether potential defects found by static analysis correspond to real bugs and whether and how they should be corrected.

The usage of static analysis in a development process has its advantages:

- **Speed** – with comparison to manual code view, the static analysis is much more efficient.
- **Analysis context** – since the software can have very high complexity, in manual code review a developer is not able to keep track of, e.g., a return value of every function, every value which a global variable can acquire, and so on. But static analysis tools can handle such an enormous amount of information and use them accordingly.
- **Code coverage** – unlike dynamic analysis which covers only some execution paths, static analysis considers every possible execution path, regardless of its rarity of execution.
- **Early defect detection** – static analysis can be performed even before the software is built or unit tests are performed, thus defects can be discovered very soon in the development process.

However, usage of static analysis has also some disadvantages:

[2] **Lines of Code (LoC)** – a software metric used to measure the size of a software project.

- **Speed** – even if the static analysis is way more efficient than manual code review, some static analysis tools still fail to meet the needed requirements while using them repeatedly in *continuous integration*[1].
- **Large number of false positives** – some static analysis tools can produce a large number of reported defects that cannot happen in the real execution of the given program.
- **False negatives** – some static analysis tools may also miss some defects, which can occur in a real execution of the given program.

Various static analysis tools implement various approaches and thus have different properties. Rice's theorem [23] states that: *„Any nontrivial property about the language recognized by a Turing machine is undecidable"*. To overcome this problem abstract interpretation-based static analysers only approximate possible executions [8]. If the approximation contains every possible execution (over-approximation), we can call the tool *sound*. If the approximation does not contain every possible execution we call the tool *unsound*. Since sound tools may use over-approximation of possible executions to overcome undecidability, this can lead to detecting defects that cannot happen in any of execution. These falsely introduced defects are called *false positives*. If a static analyser is unsound, it under-approximates the possible executions, which can lead to a defect being omitted. These omitted defects are called *false negatives*.

Most of the existing and practically applicable static analysis tools are unsound. But even if these tools cannot find every defect, they have proven to be highly useful. They do not focus on perfection but usability. Some tools skip some expensive execution paths to improve scalability. Some tools also try to decrease the number of false positives at the cost of introducing some false negatives.

Static analysis tools with a high number of false positives are not very trusted by developers [9, 18, 21]. This problem often occurs with sound tools since they cannot afford to filter some false positives at the cost of not detecting some real defects. But even if these tools produce a high number of false positives, they can be very useful in domains where error-free programs are needed. Namely, the space industry, medicine, aviation industry, and so on. A real-life example of using Frama-C to create an RTE-free[2] X.509 parser can be found in [13].

2.2 Facebook Infer

Facebook Infer [34] is an open-source[3] static analysis framework based on *abstract interpretation*. Basic principles of abstract interpretation are explained in [8]. The majority of Infer is written in OCaml. Other used languages are Java, C++, Python, and a few more. Infer runs on Linux and macOS operating systems. Infer provides an API which helps

[1] **Continuous integration** is a development practice of merging the work of multiple developers into a shared repository several times a day. Tasks required to merge the work, like building the project, running tests, or running static analysis are in most of the cases automated.

[2] **Run Time Error (RTE).**

[3] **Facebook Infer** sources – https://github.com/facebook/infer/.

developers to write new analysers much easier. The analysers are written as Infer's plugins. Even though Infer is a framework, it already contains a set of default and non-default (need to be explicitly enabled) plugins. Infer can detect many types of defects namely, *memory leaks*, *null pointer dereferences*, *dead stores*, *resource leaks*, *buffer overruns*, *integer overflows/underflows*, *uninitialized values*, *deadlocks*, *data races*, and more. A full list of detected defects can be found in [35].

Currently, Infer supports C, C++, Objective C, Java, and its version 1.1.0 also included frontends for C# and .NET from Microsoft's open-source[1] Infer# (InferSharp). But the frontend can be adjusted to support additional languages as hinted in [16]. The frontend transforms input source code into SIL (Smallfoot Intermediate Language) [22, 2, 25] which is an intermediate language on which the analysis is later performed. Even though the analysis is performed on SIL regardless of the original language, the plugins can be language-dependent. Infer takes as its input a working compilation command and produces a list of found defects in a text and JSON format.

Infer allows developers to write *inter-procedural* analysers and scale up to millions of LoC. Infer is not aimed to be sound, which means that Infer can miss some defects, but rather to scale up well and be easy to use. Possible reasons why some defects can be missed together with real-life examples of detected false negatives can be found in Section 4. Infer's analysis can be heavily parallelized on the level of files or even on the level of functions with respect to their dependencies. Infer works in two phases:

- **The capture phase** [41] – Infer uses given compilation commands to transform source code into SIL.
- **The analysis phase** [40] – Infer runs all the enabled analysers on captured source files.

An example of the Infer's *defect trace* of a null dereference can be seen in Listing 2.1. On lines 1 and 2 Infer prints basic information about the defect like a path to file, line, type of defect, and a brief explanation. On lines 4 and 13 there are individual steps of defect trace. The report also contains surrounding code to speed up the manual checking. The surrounding code is printed only if the source files are in the same path while running the analysis phase as they were while running the capture phase. The surrounding code can be included only in text output, not in the output in JSON format.

An important note is that Infer's output is *non-deterministic* [24, 27]. This means that Infer can report a different number or type of defects each time on the same software. The non-determinism is caused by various reasons – namely, timeouts in analysers or the order in which functions get analysed. The non-determinism can be lowered by disabling parallel analysis, which will significantly or sometimes completely resolve the non-determinism. However, this step also significantly decreases the performance. This approach was also experimentally tested. Another approach is to increase timeouts in analysers, e.g., in the Bi-abduction plugin, which can be done by `--seconds-per-iteration` option.

[1] **InferSharp** sources – https://github.com/microsoft/infersharp.

```
unix/unix.c:256: error: Null Dereference
  pointer ‘t‘ last assigned on line 236 could be null and is dereferenced
  at line 256, column 10.

unix/unix.c:228:1: start of procedure ex2in()
226.    }
227.
228. >  char *ex2in(x, isdir, pdosflag)
229.    char *x;                /* external file name */
230.    int isdir;              /* input: x is a directory */

...

unix/unix.c:252:9: Taking false branch
250.            n++;            /* strip ‘share’ name */
251.        }
252. >      if (*n != '\0')
253.          t = n + 1;
254.      } else

unix/unix.c:256:10:
254.      } else
255.          t = x;
256. >    while (*t == '/')
257.        t++;                /* strip leading '/' chars */
258.      while (*t == '.' && t[1] == '/')
```

Listing 2.1: An example of Infer's detailed output about a discovered null dereference defect. Positional information is highlighted in blue. The type and severity of the defect are highlighted in red. The major brief explanation of the defect is highlighted in green and a brief explanation about every trace step is highlighted in grey. The black texts are code snippets from the source files.

Deployment of Infer

Infer is currently used in Facebook [6, 9], Amazon [7], Spotify, Mozilla, and other companies. In Facebook, Infer is mainly used in mobile app development targeted to Android and iOS platforms. These mobile apps are, e.g., Facebook, Instagram, Messenger, and WhatsApp. Unlike mobile apps, which are written in Java/C++, the web services are usually written in PHP or JavaScript, and thus Infer is not applicable to these projects. Another reason for applying Infer mainly on mobile apps is that for mobile apps it places a bigger emphasis on error-free code, since deploying bug fixes to mobile apps is much harder than to web services. Fixed versions of web services can be relatively easily updated on the servers, but fixed versions of mobile apps can be only updated on Google Play or App Store and thus it depends on the user whether he/she accepts the update or not.

Due to quickly changing demands from the users, Facebook uses continuous integration and the use of Infer must be adapted to this. At first, the analysis was run once a day

in a night on the whole mobile app codebase. The reported defects were later manually assigned to developers to fix. But the *fix rate* of reported defects was nearly 0 %. After this experience, a new approach was chosen. The analysis now runs at every code change. And the fix rate was increased up to 70 %. This was caused by the fact that programmers did not have to come back to the older code. Since programmers had to go through the results of the tests anyway, they were willing to go through Infer's results at the same time.

But this approach also brings new challenges. Since tests only take up to several minutes, the analysis needs to fulfill the time limit too. Infer is not able to analyse the whole multi-millions LoC app under several minutes, a different approach needed to be established. By running Infer in an *incremental way* (see Section 2.2) Infer is able to fulfill the time limit. While running incrementally, Infer only analyses the code into which a bug could have been propagated from the change. However, while experimenting with Infer, we have discovered some problems in the incremental analysis, and we have implemented a wrapper for Infer which handles the incremental analysis instead of Infer (see Section 5).

In Amazon, Infer is used to increase the level of security in AWS[1]. AWS is a cloud platform, which offers IT resources (computational power, storage, databases, machine learning, analytics etc.). Amazon also uses, e.g, the sound concurrency verifier VCC[2], the bounded model checker CBMC[3], or the OpenJML[4] verifier. A full list of tools used in Amazon can be found in [7].

Infer Plugins

Infer provides a set of analyses, which are implemented as plugins. The below presented detailed information about Infer plugins was taken from [35] and plugin sites on Infer web [34]. In Infer, there two types of plugins as shown in Table 2.1:

- **default** – these plugins are enabled by default while running the analysis phase, e.g., Bi-abduction, Starvation, or RacerD.
- **non-default** – these plugins are not enabled by default and thus need to be activated explicitly, e.g., InferBO or Quandary.

Different plugins support different languages as shown in Table 2.1. The below provided discussion of plugins applies to the 1.1.0 version. While writing the thesis a new version 1.1.0[5] was released and the following information may not be true for the 1.0.0 version, e.g., the Pulse checker is now also detecting uninitialized values and the Uninit plugin will likely be fully removed in the future versions.

Bi-abduction

Bi-abduction is a default plugin that detects various problems linked to memory safety. Namely, null pointer dereferences, resource leaks, memory leaks or possible retain cycles, which are described in more detail in Section 2.2. The plugin also detects possible division

[1] **Amazon Web Services (AWS)** website – https://aws.amazon.com/.
[2] **VCC** website – https://archive.codeplex.com/?p=vcc.
[3] **CBMC** sources – https://github.com/diffblue/cbmc.
[4] **OpenJML** website – https://www.openjml.org/.
[5] **Infer v1.1.0 release** – https://github.com/facebook/infer/releases/tag/v1.1.0.

Table 2.1: The basic division of the mentioned Infer plugins into default and non-default, together with information about language support.

Plugin	C	C++	Objective C	Java	C#	Default
Bi-abduction	✓	✓	✓	✓	✓	✓
InferBO	✓	✓	✓	✓	✓	
Cost	✓	✓	✓	✓	✓	
Eradicate				✓	✓	
Fragment Retains View				✓	✓	✓
Immutable Cast				✓	✓	
Impurity	✓	✓	✓	✓	✓	
Inefficient keySet Iterator				✓	✓	✓
Loop Hoisting	✓	✓	✓	✓	✓	
‘printf()‘ Argument Types				✓	✓	
Pulse	✓	✓	✓	✓		
Purity	✓	✓	✓	✓	✓	
Quandary	✓	✓	✓	✓	✓	
RacerD		✓		✓	✓	✓
.NET Resource Leak					✓	
SIOF		✓				✓
Self in Block		✓	✓			✓
Starvation	✓	✓	✓	✓	✓	✓
Uninit	✓	✓	✓			✓

by zero. A full list of defects that are detected by the plugin can be found on [35]. Bi-abduction is built on *separation logic* [36], which is a kind of mathematical logic that models computer memory and associated memory operations.

InferBO

InferBO (Buffer Overrun Analysis) [20] is a non-default plugin that detects defects like buffer overrun, variable overflow/underflow, unreachable code, or too big/negative allocation size. A full list of defects that are detected by the plugin can be found on [35]. InferBO uses the conventional analysis technique of intervals. But instead of using only bounds in the form `[min, max]`, InferBO can store bounds (if needed) as *linear expressions* with additional `min` and `max` operators. It calculates the bounds of array size and array offset. If an access violation is detected, then a warning is generated. Based on the type of intersection of array size bound and offset bound a different type of `BUFFER_OVERRUN` defect is generated:

- `L1` – most likely to be *true positive*[1]. If Infer was able to always correctly approximate the bounds, then the `L1` defect type would be always true positive. This is because the intersection of array size bounds and offset bounds used for accessing the array are required to be empty in this case, and thus overrun happens every time, for example: `[1,3]` $\bigcap$ `[4,4]` = $\emptyset$.

[1] **True Positive (TP)** – a defect that can happen in a real-life execution and was reported correctly by a static analyser.

- `L2` – less likely to be true positive than `L1`. Because the intersection of array size bounds and offset bounds used for accessing the array are required not to be empty in this case, and thus there is a safe combination, for example: `[1,3]` $\bigcap$ `[2,4]` = $\emptyset$.
- `L5` – least likely to be true positive. Array size bounds or offset bounds used for accessing the array are required to consider every possible value in this case, for example: `size: [1,3] offset: [-`∞`,+`∞`]`.
- `L4` – more likely to be true positive than `L5`. Because bounds are required to have a maximum of one infinity in them in this case, for example: `size: [1,3] offset: [2,+`∞`]`.
- `L3` – the reports that do not belong to the mentioned cases.
- `S2` – the bounds are required to contain symbolic values in this case, for example: `size: [x,x+10] offset: [x,+`∞`]`.
- `U5` – the bounds are required to contain unknown values in this case, which are usually from unknown return values from function calls.

The `INTEGER_OVERFLOW` defect has similar subtypes (`L1`, `L2`, `L5`, and `U5`) with similar meaning as the `BUFFER_OVERRUN` defect.

While experimenting with InferBO, we have discovered, that analysis on a large project can get stuck. For example on package curl-7.71.1-8 in Fedora 33. This is caused by an enormous amount of generated internal values during the analysis. In some cases, this issue could be resolved by setting the maximum number of internal InferBO values (option `--bo-field-depth-limit N`) as hinted in [26]. But the solution also decreases the precision of the analysis.

Cost

Cost is a non-default plugin that computes the *time complexity* of given functions and methods. The reported time complexity is an estimated upper bound on the worst-case execution. The analysis computes for each node in the CFG (Control Flow Graph) its cost and maximum executions. The total cost of each node is then calculated as $instrCost = costPerExecution * maxExecutions$. The function cost is then calculated as $funCost = \sum instrCost$. For calculating the bounds of the control variable, Cost uses internally InferBO. There are few known limitations of this plugin. The plugin is not able to handle recursion. And if the plugin is not able to calculate the cost of a function call (when a model is missing), then the plugin is not able to compute the upper bound and returns a `T` value, which indicates unknown time complexity. Detailed explanations can be found in [5].

Eradicate

Eradicate is a non-default plugin, which searches for unprotected uses of Java parameters, fields, and returns values annotated as `@Nullable`. If, e.g., a method parameter is annotated `@Nullable`, the programmer explicitly said that this parameter can be null and thus the usage of the parameter needs to be protected.

Fragment Retains View

Fragment Retains View is a default plugin, which checks if Android fragments are nullified before becoming unreachable. Android fragments are reusable parts of UI [38]. This plugin is currently unmaintained due to low precision and is deprecated.

Immutable Cast

Immutable Cast is a non-default plugin, which checks for casts from immutable to mutable data types. For example in Java, from `ImmutableList` to `List`. The plugin is currently unmaintained and deprecated.

Impurity

Impurity is a non-default plugin, which detects functions with possible side effects (*impure functions*), for example, modifying a global variable. The plugin is implemented on top of the Pulse plugin and is intended to replace the Purity plugin.

Inefficient keySet Iterator

Inefficient keySet Iterator is a default plugin, which detects inefficient uses of iterators. As inefficient iterator is considered, an iterator that iterates on keys and then accessing the values with the keys, instead of iterating on key-value pairs directly.

Liveness

Liveness is a default plugin, which detects dead stores and unused variables. Dead stores can be detected even on allocated memory.

Loop Hoisting

Loop Hoisting is a non-default plugin, which detects *loop invariant function calls*. These function calls can be moved out of the loop to optimize the program. The plugin relies on the Purity plugin to determine whether the examined function has any side effects. By passing `--hoisting-report-only-expensive` to Infer, the plugin will report only expensive (which have at least linear complexity) function calls.

'printf()' Argument Types

'printf()' Argument Types is a non-default plugin, which detects mismatches between `printf` format strings and the passed argument types. The plugin is unmaintained and deprecated.

Pulse

Pulse is a non-default plugin, which detects various problems linked to memory safety and lifetime analysis. The plugin reports problems like null pointer dereferences, memory leaks, uses after free/delete/lifetime, or uninitialized values. The plugin aims to replace Bi-abduction and Uninit plugins.

Purity

Purity is a non-default plugin, which detects functions without side-effects (*pure functions*). An important note is that functions with unknown models are treated as impure.

Quandary

Quandary is a non-default plugin that performs *taint analysis*. Taint analysis tracks the flow of user input through the program and checks if the input can affect the program in an undesirable way. Infer has a built-in list of *sources* and *sinks*, but it is also possible to define custom ones in `.inferconfig` file[1].

RacerD

RacerD is a default plugin, which detects data races. The plugin does not check all the code for possible data races, but only checks:

- methods and classes annotated with `@ThreadSafe`.
- code with explicit usage of locks via the `synchronized` keyword.

If these conditions are detected, then RacerD looks for data races in the code satisfying the condition and all of its dependencies. To improve the analysis results – to increase coverage and decrease the false positive ratio, annotations can be used. For example, `@Functional` annotation tells RacerD, that the annotated method always returns the same value, and thus data race reported on a variable into which only the annotated method writes is a *benign data race* [32]. Benign data races are indeed true data races, but they do not negatively affect the program, they are present mainly for efficiency reasons. RacerD's principles are described in more detail in [3]. Description of possibly detected defect types can be found on a plugin website on [34].

.NET Resource Leak

.NET Resource Leak is a non-default plugin, which checks for resource leaks in C# programs. The plugin was added into Infer in the latest release[2]. The plugin was developed in Microsoft for Infer#[3] and added together with C# frontend to Facebook Infer.

SIOF

SIOF (Static Initialization Order Fiasco) is a plugin, which detects problems in static initializations in C++. The problem can occur when a static object `X` manipulates with a static object `Y` located in a different file. Because the objects are in different files the order of their initialization is compiler dependent. Moreover, if object `X` is initialized before object `Y`, then accessing object `Y` from the constructor of object `X` will cause an error. An example of a SIOF defect can be found in [30].

[1] **.inferconfig** file for Quandary plugin – https://github.com/facebook/infer/blob/master/infer/tests/codetoanalyze/java/quandary/.inferconfig.

[2] **Infer release v1.1.0** – https://github.com/facebook/infer/releases/tag/v1.1.0.

[3] **InferSharp** sources – https://github.com/microsoft/infersharp.

Self in Block

Self in Block is a default plugin, which detects occurrences of `this` variable in blocks, which can lead to a *retain cycle*. Retain cycle happens, e.g., when two objects are holding a reference to each other. Then if all the external references are destroyed, the objects remain in the retain cycle and keep each other from freeing. This problem happens in the Java ARC (Automatic Reference Counting) memory management model [29], which keeps track of references to memory objects, and after the reference count reaches zero the object is freed. This approach cannot deal with retain cycles. Java is also using the GC (Garbage Collector) memory management model, which can remove the retain cycles. GC is looking for reachable objects from the program and since the whole retain cycle is not reachable from the outside the whole part of a *memory graph* containing the retain cycle will be freed. Even if the retain cycle is mentioned in connection with automatic memory management languages like Java, it can also lead to memory leaks in languages without GC, like C++ [14].

Starvation

Starvation is a default plugin, which detects the *starvation* problem in concurrency programs. The starvation plugin detects four types of starvation problems:

- Deadlocks.
- Violations of `@Lockless` annotations.
- Violations of the Android's strict mode.
- Doing expensive operations on the Android UI thread.

Uninit

Uninit is a default plugin, which detects usage of uninitialized values. The plugin may be removed in future releases, because of its low precision. The plugin will be replaced by the Pulse plugin.

Incremental Analysis

Incremental static analysis is used, when a static analyser is deployed in continuous integration, to reduce its computational time and space requirements. A static analyser running in an incremental mode only analyses parts of code into which a defect could have been propagated from the changes.

The incremental analysis is implemented in Infer on a function level. This means, that only the functions in the changed files and their callers should be re-analysed. Assume the *call graph* shown in Figure 2.1. If Infer analyses the given example for the first time, Infer at first analyses leaf functions of the call graph – `g()`, `r()`, `printf()`, and `c()`. Then Infer goes up, towards the call graph root – function `main()`. Infer only analyses the parent nodes, if all of its children were already analysed. So then Infer analyses function `f()` and lastly, the function `main()`. Now, if we change the file containing the function `f()` and we run the Infer in an incremental mode, only the function `f()` and its parent – `main()`, should be re-analysed. However, while experimenting with an incremental analysis in Infer,

we have discovered that Infer analyses wrong functions. Based on that observation, we have proposed a way that corrects this faulty behavior (see Section 5).

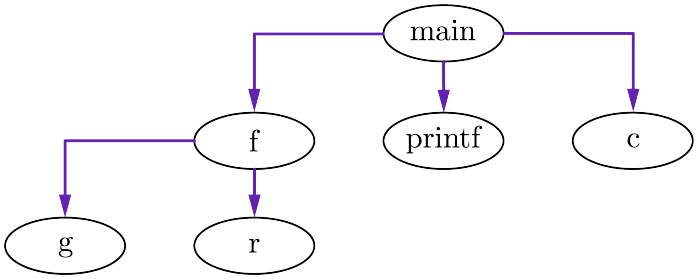

Figure 2.1: A call graph of an incremental analysis example. For experiment's source files, see Appendix D.

2.3 Csmock

Csmock is a command line tool that integrates static analysers such as Cppecheck or Shellcheck and makes it possible to run these analysers in a fully automated way on a given source RPM[1] package. Information about csmock was taken from [12, 11, 10] and csmock repository[2].

Csmock takes as input an SRPM[3] package and returns a list of found defects. Basic usage of csmock is illustrated in Figure 2.2. Csmock is available as a package for the Fedora and CentOS operating systems. Csmock is fully open-source and provides a plugin API for new static analysers. Currently supported analysers are listed in Table 2.2.

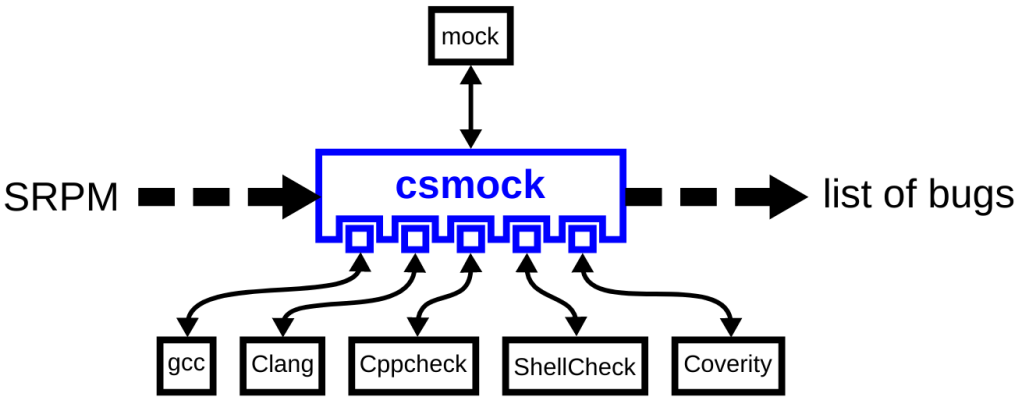

Figure 2.2: An illustration of the basic usage of the csmock tool for automated analysis of SRPM packages. The Figure was taken from [12].

[1] **RPM package manager (RPM)** – a package manager for Linux distributions.
[2] **Csmock** sources – https://github.com/kdudka/csmock.
[3] **Source RPM (SRPM)** – a package that contains sources of a RPM package.

Table 2.2: A list of supported static analysers and supported languages in csmock [12].

analyser	C	C++	Java	Go	JS	PHP	Python	Ruby	Shell
gcc	✓	✓							
Clang	✓	✓							
Cppcheck	✓	✓							
Coverity	✓	✓	✓	✓	✓	✓	✓	✓	
ShellCheck									✓
Pylint							✓		
Bandit							✓		
Smatch	✓								

Various static analysers have various output formats. Csmock internally transforms these formats into a unified one. The csmock output format is shown in Listing 2.2. More examples of the csmock format can be found in the csdiff repository[1] This format is further used by open-source[2] tools from the csdiff package:

- `csdiff` – is used for *differential filtering*. Csdiff takes as an input two lists of defects – old and new. The output is a list of newly introduced defects or vice versa removed defects.
- `csgrep` – filtering the given list of defects by given criteria.
- `cshtml` – formats the given list of defects as a human-readable HTML document.
- `cssort` – sorts the given list of defects by the given key.
- `cslinker` – adds CWE[3] numbers, scan properties etc. into the list of defects.

```
1 Error: INFER_WARNING:
2 hostname.c:130:15: note:  allocation part of the trace starts here
3 hostname.c:130:15: note:  allocated by call to ‘malloc‘ (modelled)
4 hostname.c:130:15: note:  allocation part of the trace ends here
5 hostname.c:126:9: note:  memory becomes unreachable here
6 hostname.c:126:9: error[MEMORY_LEAK]: memory dynamically allocated at line
7  130 by call to ‘malloc‘, is not freed after the last access at line 126,
8  column 9.
```

Listing 2.2: An example of an Infer's warning about a memory leak transformed into csmock the format. Positional information is highlighted in blue. The analyser and the defect type are highlighted in red. The major brief explanation of the defect is highlighted in green and a brief explanation about every trace step is highlighted in grey.

Csmock is built on mock [4]. Mock is a tool for building RPM packages from SRPM packages inside a chroot (see Section 2.3). Mock allows the user to install packages into the

[1] More examples of csmock output format – https://github.com/kdudka/csdiff/tree/master/tests/csgrep.

[2] **Csdiff** sources – https://github.com/kdudka/csdiff.

[3] **Common Weakness Enumeration (CWE)** – a list of software and hardware weakness types.

chroot, copy files from or into the chroot, run commands in the chroot, etc. In order for a user to use the mock tool and therefore csmock tool, the user must be added to the mock group. This can be done by running the command: `sudo usermod -a -G mock $USER`. The complete installation of csmock from sources is described in Appendix B.

Chroot

Chroot (change root) [39] is a technique used in the UNIX operating system, which runs processes in a separated environment. Creating chroot means changing the root directory for the current running process and its child processes. This operation creates a separated environment in which programs cannot access files (and therefore programs) above the newly created root (in the host system). This separated environment is called a *chroot jail*. The main reason the mock utility uses the chroot is that a clean and identical environment can be set for the same build every time. Csmock uses mock with turned-off internet connection, so the packages cannot download anything from repositories, sites, or storages this way. If packages were able to, e.g., download the latest libraries before the build, the idea of the same environment would be violated. However, this approach causes some problems while installing programs that are not distributed as RPM packages, e.g., Infer.

Chapter 3

Infer Plugin for Csmock

This chapter describes the implementation of our Infer plugin for the csmock tool. It also explains our original heuristics that we propose for removing false positives from Infer reports in the context of analysing SRPM packages. Lastly, this chapter explains future work on the Infer plugin and *false positive filtering.*

Our Infer plugin uses the csmock API to run various stages of Infer in chroot. The plugin works with currently the newest release of csmock, namely, 2.7.1[2]. Updated sources with Infer plugin of forked csmock can be found on GitHub[3]. The plugin is not yet merged in the csmock's official repository[4] and thus is not available as an RPM package and needs to be installed from sources, see Appendix B. The plugin does not depend on a specific version of Infer, but was tested on version 1.0.0[5]. The detailed process of passing Infer's binaries to csmock is explained in Section 3.1 and Appendix B. The plugin consists of three scripts:

- `infer.py` – a control script that directly uses csmock API and controls, e.g., which scripts should be run in chroot or which files should be copied out of/into chroot.
- `install-infer.sh` – a script which sets up the environment in running the chroot for Infer – it, e.g., installs Infer or creates *compiler wrappers*.
- `filter-infer.py` – a script that transforms Infer's output (see Listing 2.1) to csmock output (see Listing 2.2). The script also implements heuristics for filtering false positives.

Csmock sets up and cleans the chroot for every package. But all the enabled csmock plugins run in the same environment (in the same chroot). Because of this, the plugins should not interfere, with each other and if so, they need to be coordinated. To coordinate some actions executed by plugins, each plugin has a unique priority. Plugin priorities are described in more detail in Section 3.1.

To include a plugin to a csmock installation process and make it available in csmock, it must be registered in `py/CMakeLists.txt` and `scripts/CMakeLists.txt`.

[2] **Csmock v2.7.1 release** – https://github.com/kdudka/csmock/releases/tag/csmock-2.7.1.
[3] **Csmock with Infer plugin** – https://github.com/TomasBeranek/csmock.
[4] **Csmock official repository** – https://github.com/kdudka/csmock.
[5] **Infer v1.0.0 release** – https://github.com/facebook/infer/releases/tag/v1.0.0.

3.1 Control Script

The control script `infer.py` is located in the `py/plugins/` directory with other plugins. The script directly uses the csmock API and is implemented in Python. Every registered (in `py/CMakeLists.txt`) Python script located in the `py/plugins/` directory is considered a plugin and thus needs to implement the following two classes:

- `PluginProps` – contains a constructor which sets up properties of the plugin:
  - `pass_priority` – defines the plugin's priority. Plugins are executed based on their unique priority, e.g., the order of executing Python hooks after package dependencies are installed in the chroot. Since the Infer plugin should not interfere with other plugins, its priority is set to `0x60` (Infer will be executed last).
  - `description` – a brief description of the analyser.
  - `experimental` – information if a plugin is only experimental. In the Infer plugin, the property is not set.
- `Plugin` – contains methods that implement the plugin's behavior:
  - `__init__` – a constructor of the plugin. When the plugin is instantiated its `enabled` property is set to `false` (the plugin must be explicitly enabled by csmock, based on the `-t` option [10]).
  - `get_props` – returns an instance of `PluginProps`.
  - `enable` – enables the plugin.
  - `init_parser` – specifies the plugin's options that will be available when invoking csmock through the command line.
  - `handle_args` – specifies operations that should be executed in chroot through the csmock API.

Parser Initialization Method

In the `init_parser` method of the `Plugin` class there are defined options that will be shown in the csmock's help and with which csmock could be invoked through the command line. Information about the options (if they were passed or with which value) is then accessible through the `args` parameter in the `handle_args` method. The format of an option specification can be found in the argparse library documentation[1]. The Infer plugin currently supports the following command line options:

`--infer-analyze-add-flag` – appends the given flag (except `-o`), when calling `infer analyze`. The option can be used multiple times.

`--infer-archive-path` – the plugin uses the given `.tar.xz` archive with the binary release of Infer. The default location is `/opt/infer-linux*.tar.xz`.

`--no-infer-filter` – disables the false positive filter. The Infer's output will be transformed to the csmock's output only.

`--no-infer-biabuction-filter` – disables the Bi-abduction filter.

[1]The **argparse** library documentation – https://docs.python.org/3/library/argparse.html.

`--no-infer-inferbo-filter` – disables the InferBO filter.

`--no-infer-uninit-filter` – disables the Uninit filter.

`--no-infer-memory-leak-filter` – disables the memory leak filter.

`--no-infer-dead-store-filter` – disables the dead store filter.

The Method for Handling Arguments

The `handle_args` method of the `Plugin` class takes three arguments:

- `parser` – this is an `ArgumentParser`[1] object. In this method, a `parser` object is used to generate a parser error if any of the given options do not meet the plugin's requirements, e.g., the given path does not exists.
- `args` – this is an object returned from `ArgumentParser.parse_args` that contains values of options specified in `init_parser`.
- `props` – this is a `ScanProps` object (`ScanProps` is defined in the `py/csmock` file). The object collects plugin properties and operations which should be executed, e.g., commands to run in chroot, which files should be copied out of/into chroot, etc. The object is shared by all plugins.

The method must ensure that nothing will be executed when the `enabled` property is set to `false` (plugin is disabled). The method then uses the API `props.install_pkgs` to indicate which packages are needed to successfully run Infer. The following packages (needed by Infer) are installed in chroot before the analysis begins:

- `python` – Python is needed to run the `filter-infer.py` script and even some parts of Infer which are written in Python.
- `ncurses-compat-libs` – a terminal control library, specifically the file `libtinfo.so.5` is needed by Infer.
- `sqlite` – the compiler wrapper of Infer uses the `sqlite3` utility for SQL[2] queries on the Infer's database.
- `clang` – needed our memory leak filter for AST[3] generating.

Our Infer plugin needs an archive with the binary release of Infer. The plugin is implemented as version-independent but was only tested with Infer v1.0.0. The archive can be downloaded from the Infer repository[4]. The plugin needs the Infer archive, because Infer does not exist as an RPM package and internet access is turned off in chroot, and so Infer cannot be downloaded there. But it is also possible to use the GitHub API to download the archive before the chroot is created and store it, e.g., in the `/tmp` directory. However, we did not choose this approach because as future work, it is intended to create an RPM package from Infer to make the installation to the chroot easier. Another reason is that this way the plugin can run even modified versions of Infer such as these of [25, 17].

[1] **argparse** library documentation – https://docs.python.org/3/library/argparse.html.
[2] **Structured Query Language (SQL)** – a standard language for communication with a database.
[3] **Abstract Syntax Tree (AST)** – a representation of the abstract syntactic structure of a source code.
[4] **Infer releases** – https://github.com/facebook/infer/releases.

The `handle_args` method looks for the archive in the `/opt` directory, where it tries to find an archive named `infer-linux.*.tar.xz`. If `-infer-archive-path` option is passed when invoking csmock, the method uses the user-specified path to check if the archive exists. If so the archive is used, otherwise, parser error is generated.

With csmock API `props.copy_in_files` the archive with Infer, `install-infer.sh`, and `filter-infer.py` files are copied into chroot. In chroot, the files are located in the same path as on the host system.

The installation of Infer and compiler wrappers is done by running the `install-infer.sh` script in chroot after all dependent packages are installed. This is ensured by csmock API `props.post_depinst_hooks`, which will run a Python function `install_infer_hook`, which runs the `install-infer.sh` script, and as the first argument, the Infer archive's location is passed.

To run a command in chroot, the plugin must use hooks or the `props` parameter. When a post dependency installation hook is called it gets a `MockWrapper` object as the second argument, and by calling its method `exec_chroot_cmd`, it is possible to propagate the given command to mock and then to chroot.

To run a set of commands in the chroot after the build of the SRPM package, the csmock API `props.post_build_chroot_cmds` is used. It is a list of commands to be executed. The given flags for analysis (specified by `--infer-analyze-add-flag`) are serialized and a command `infer analyze` is constructed and added to `props.post_build_chroot_cmds`. The analysis phase must be executed after the build because during the build the capture phase is translating source files to SIL. If the build is finished it is ensured that no more files are needed to be captured and thus it is safe to run the analysis phase.

After the analysis finishes, a command to run `filter-infer.sh` is constructed in the same way as the command for running the analysis phase. This script creates a file in the chroot called `/builddir/infer-results.txt`. Which contains all the defects which were left after the filtering. The file contains defects in csmock format (see Listing 2.2).

The files with analysis results are copied (together with debug files) out off the chroot by using `props.copy_out_files` API. The last step is to call the `csgrep` utility which checks if the result file has the desired format and further processes the output (e.g., adds colors). The `csgrep` output is then saved into the archive which is produced by csmock.

3.2 Install Script

The `install-infer.sh` is a bash script that installs Infer into chroot. The script is located in the `scripts/` directory in the csmock repository. A path to the archive is passed to the script as the first argument. The installation process in chroot is the same as installation to an ordinary Linux operating system[1].

[1] **Infer installation** – https://fbinfer.com/docs/getting-started/.

We have considered multiple ways of getting compilation commands needed for Infer's capture phase:

- **Obtain a compilation command from a SPEC file.** – Every SRPM package has a SPEC file, which defines the properties of the package. In the SPEC file are also included compilation commands. Extraction of the compilation commands would be relatively easy, but such compilation commands do not have to be, e.g., `make`, `ant`, etc. But it can be an arbitrary script within the package. The compilation command can be located in the script at a completely specific location, and thus Infer would not be able to extract it.
- **Use a generic compilation command.** – Another approach is to use a generic compilation command, e.g., `clang -c file.c`. However, this compilation needs at least prototypes of external functions which are located in header files. The path to these header files as well as their source files are often specified within the compilation command, e.g., through the `-I` option. So the absence of the library path specification can cause compilation problems. However, even if this modified compilation command succeeds, information about conditional compilation is lost, e.g., the `-D` option.
- **Use a compiler wrapper.** – As our final approach we have chosen to create wrappers for commonly used C/C++/Java compilers. When any of the wrappers are called, they sequentially run the capture phase of Infer and then call the original compiler. This way no information about the build process is lost.

The `install-infer.sh` script is also responsible for creating wrappers for C/C++ and Java compilers. Supported C/C++ compilers are `8cc`, `9cc`, `ack`, `c++`, `ccomp`, `chibicc`, `clang`, `cproc`, `g++`, `gcc`, `icc`, `icpc`, `lacc`, `lcc`, `openCC`, `opencc`, `pcc`, `scc`, `sdcc`, `tcc`, `vc`, `x86_64-redhat-linux-c++`, `x86_64-redhat-linux-g++`, `x86_64-redhat-linux-gcc` and `x86_64-redhat-linux-gcc-10`. The supported compiler for Java is `javac`. However, our plugin is aimed to analyse C/C++ packages and thus was not tested on packages written in Java.

There are two types of wrappers. For C/C++ and Java. They differentiate in a way they invoke the `infer capture` command. For C/C++ compiler wrappers must be passed the option `--force-integration cc`. This indicates to Infer, that the given compile command should be handed over to the Infer's internal C/C++ compiler – `clang`. For the Java compiler wrapper, the option `--force-integration javac` needs to be used, to hand the command over to the Infer's internal Java compiler – `javac`. This serves as a precaution for unknown compilers. If the `--force-integration` option is not passed, Infer may not be able to recognize the given compile command and the capture phase may fail.

Since other compilers may not be compatible with the Clang compiler, a command executed, e.g., with GCC may not be valid when executed with Infer's internal Clang. This will cause Infer's capture phase to fail. The plugin is built in a way that when this happens the whole analysis will not end with an error but keeps going and tries to analyse as many source files as possible. To reduce the incompatibility between compilers, some incompatible options are being removed when invoking the Infer's capture phase (the call to the original compiler remains unchanged). Some of the removed options are, e.g., `-fstack-clash-protection` or `-fwhole-program-vtables`. The full list of the filtered options can be found in the `install-infer.sh` script. The wrapper also filters out options linked with LTO (Link

Time Optimization) for two reasons. Firstly, these options have a different format in Clang and GCC. Secondly, LTO causes a deadlock when the compiler is not called directly but through a wrapper. The problem with LTO is very common since it is enabled by default in Fedora 33 and the newer versions (due to this fact the testing was moved from Fedora 29 to Fedora 33). The wrapper also skips capturing of some special files, e.g., `conftest.c` which is a file generated by Autoconf[1] scripts.

The wrapper also stores the compilation commands together with a list of files that were captured by Infer with the given compilation command in the file `/builddir/infer-ast-log` in the chroot. The list of freshly capture files is obtained from the Infer's database. Infer marks all the files captured by the last command. The information is stored as `0` or `1` in the column `freshly_captured` in the table `source_file` in the database `infer-out/results.db`. This information is later used to generate an AST which is used to filter out memory leaks that are considered intentional, more details in Section 3.4.

While running the Infer's capture phase, Infer modifies its database. The compiler wrapper also stores compilation commands with a list of files for AST generating. Both of these operations are considered critical because both are writing in a file. If two instances of either of the operations above were trying to write into the file and a *context switch* happened, then the file could end up in an inconsistent state. Because compilers (and therefore compiler wrappers) are often called in parallel, some precautions need to be taken to prevent inconsistencies.

Bash scripts do not offer a locking mechanism. Due to this fact, *mkdir locking* was used. This technique uses the atomicity of the `mkdir` operation (on some network filesystems, it may not be atomic). If `mkdir` tries to create a directory that already exists it fails. This can be used as a substitution for locks. Moreover, the script must also contain a mechanism that will avoid a deadlock when the process that has created the directory is terminated. This method was successfully tested on dozens of SRPM packages.

3.3 Filter Script

The `filter-infer.py` script is located in the `scripts/` directory in the csmock repository. The script implements heuristics from Section 3.4 and transforms the Infer's output format (see Listing 2.1) into the csmock's output format (see Listing 2.2). The script takes as input (on `stdin`) a list of defects in the JSON format (generated by Infer) and prints out a filtered and transformed list of defects in the csmock format. If the `--only-transform` option is passed when invoking the script, no defects will be removed. Individual filters can be disabled by command line options as well (see Section 3.1 or Appendix B). All filters, except the memory leak filter, are implemented as simple pattern matching rules and thus can be applied even outside of csmock. The memory leak filter uses the AST to prove certain properties of a variable on which the memory leak was reported.

[1] **Autoconf** is a tool for creating packages from source files.

3.4 Filtering False Positives

A large number of false positives is a well-known problem of static analysis tools [18, 21]. Due to the large number of reports which developers have to go through, the incredulity of developers toward static analysis tools increases. To lower the percentage of false positives in reported defects, we have been trying to discover common patterns in false positives reported by Infer, which can be later used for removing reported defects that are likely to be false positives. To discover such patterns a large number of defects must have been manually checked. More information about analysed and checked packages is given in Section 4. The following heuristics are based on the discovered patterns. These heuristics can also filter out true positives, but the percentage of filtered true positives is much lower than the percentage of filtered false positives as our later presented experiments show. In many cases, the heuristics did not filter out any true positives. More statistics in Section 4. If any of the below-mentioned filters runs into an error, the worst-case scenario is that the examined defect will not be filtered out.

3.4.1 An Uninit Filter

Our Uninit filter tries to identify defects `UNINITIALIZED_VALUE` which are likely to be false positive. The uninitialized value defects are reported by the Uninit plugin, described in Section 2.2. While manually checking Infer reports, it has been discovered, that Infer treats variables and arrays which are initialized in loops as uninitialized. Infer considers that the loop does not have to be executed at all (even if the loop has constant boundaries). Since almost every array is initialized in a loop, this behavior causes false positives to be reported on every array which is initialized in this way. From the Infer report, it is possible to find out whether the defects are reported on an array or not, as shown in Listing 3.1. If so, then the reported defect is filtered out.

```
1 crypt.c:232: error: Uninitialized Value
2   The value read from header[_] was never initialized.
```

Listing 3.1: An example of an Infer output about an uninitialized value defect reported in a `zip` package. The indication that the uninitialized variable is an array is highlighted in red.

3.4.2 A Bi-abduction Filter

Our Bi-abduction filter tries to identify defects `NULL_DEREFERENCE` and `RESOURCE_LEAK` which are likely to be false positive. Both defect types are reported by the Bi-abduction plugin, described in Section 2.2. The bi-abduction plugin according to our experience, often skips function calls. By manually going through reports produced for the analysed Linux packages, we have discovered that these skipped calls are mostly calls to user-defined exit functions (at least in the considered context). These functions are in majority of the cases used to exit, e.g., if the pointer is null. Infer therefore does not consider some constructions as safe since it will not see that the execution is exited for a null pointer. In the checked defects, the calls were skipped for two reasons. The first is that Infer has an empty specification of the called function (the function was not translated to SIL and Infer does not have a model). The second reason is that Infer was not able to resolve a function pointer. From

the Infer report, it is possible to find out whether any functions were skipped, as shown in Listing 3.2 and Listing 3.3. If so, then the reported defect is filtered out.

Another common pattern in false positives reported by the Bi-abduction plugin is a skipped switch case. In many cases, Infer is not able to determine the order in which switch cases will be executed. If this situation happens Infer assumes, that any order is possible. If one switch case contains an initialization of a pointer and another switch case contains a dereference of that pointer, this will automatically lead to a warning generation. This type of warning was not in any of the manually checked reports proven to be true positive. If the skipped switch case as shown in Listing 3.4 is detected, then the defect report is filtered out.

```
1 fileio.c:2072: error: Null Dereference
2   pointer ‘archive\_name‘ last assigned on line 2066 could be null and is
3   dereferenced by call to ‘strcpy()‘ at line 2072, column 5.
4
5 ...
6
7 fileio.c:2046:3: Skipping get_in_split_path():  empty list of specs
8
9 ...
```

Listing 3.2: An example of an Infer output about a null dereference defect reported in the zip package. The indication that there has been a skipped function call is highlighted in red.

```
1 lib/obstack.c:204: error: Null Dereference
2   pointer ‘new\_chunk‘ last assigned on line 200 could be null and is
3   dereferenced at line 204, column 3.
4
5 ...
6
7 lib/obstack.c:87:5: Skipping __function_pointer__():  unresolved function
8     pointer
9 ...
```

Listing 3.3: An example of an Infer output about a null dereference defect reported in the sed package. The indication that there has been a skipped function pointer dereference is highlighted in red.

3.4.3 An InferBO Filter

The InferBO filter tries to identify buffer overrun and integer overflow defects reported by the InferBO plugin (described in Section 2.2), which are likely to be false positives. The considered buffer overrun defect types are:

- `BUFFER_OVERRUN_L2`

```
lib/dfa.c:4153: error: Null Dereference
  pointer ‘mp‘ last assigned on line 4060 could be null and is
  dereferenced at line 4153, column 11.

...

lib/dfa.c:4074:9: Switch condition is false.  Skipping switch case

...
```

Listing 3.4: An example of an Infer output about a null dereference defect reported in the sed package. The indication that there has been a skipped switch case is highlighted in red.

- BUFFER_OVERRUN_L3
- BUFFER_OVERRUN_L4
- BUFFER_OVERRUN_L5
- BUFFER_OVERRUN_S2
- INFERBO_ALLOC_MAY_BE_NEGATIVE
- INFERBO_ALLOC_MAY_BE_BIG

The considered integer overflow defect types are:

- INTEGER_OVERFLOW_L2
- INTEGER_OVERFLOW_L5
- INTEGER_OVERFLOW_U5

Some of the mentioned defect types are explained in more detail in Section 2.2 and a full list with explanations of individual defect types can be found in [35]. The InferBO plugin, according to our experience, often over-approximates boundaries of possible values of a variable to $-\infty$ or $+\infty$. This usually happens on a global variable or variables that are assigned in loops with complex control conditions as described in more detail in Section 6.2. If the bound contains, e.g., $+\infty$, every action which increases the value of the variable will be reported as an integer overflow. This behavior leads to a large number of false positives as verified by a manual check. Since the information about the boundaries of the variable on which the defect was reported can be found in Infer reports as shown in Listing 3.5, these defects can be filtered out.

```
sed/execute.c:1623: error: Integer Overflow L1
  ([-oo, 0] - 1):unsigned64 by call to ‘debug_print_line‘.
```

Listing 3.5: An example of an Infer output about an integer overflow defect reported in the sed package. The indication that there has been a $+\infty$ or $-\infty$ is highlighted in red.

3.4.4 A Memory Leak Filter

Our memory leak filter tries to identify `MEMORY_LEAK` defects reported by the Pulse checker (described in Section 2.2), which are likely to be false positives. Unlike the filters mentioned above, which all work with the Infer output only, the memory leak filter also needs an AST generated from the Clang compiler. Because of that, this filter cannot be easily applied outside of csmock.

While manually going through reported memory leaks, we have discovered that many of these memory leaks are true positives but are intentional. Very often, developers intentionally leave memory freeing on the operating system, which must free all the memory that the program has allocated after the end of the program. In some cases, this can lead to better program performance or more readable and shorter code. In some cases, it may not be clear if a developer leaves the memory freeing on the operating system intentionally. To solve this, we take as intentional memory leaks those whose address (the value returned from, e.g., `malloc`) is still reachable at the end of the program or at the end of the variable's scope. The goal of this filter is to filter out memory leaks that are intentional by our measures.

To determine if the memory is still reachable, the filter uses a generated AST and an Infer report. Our heuristic to differentiate reachable memory from unreachable memory is based on the fact that if the variable's value which holds the memory address is not overwritten before the end of the program or before the variable's end of the scope, the memory address remains intact. Since it is only a heuristic, there are cases in which the principle does not work, e.g., due to accessing variables through pointers. If some pointer points to a monitored variable, the value in the variable can be changed without explicitly using the variable in the code.

The filter takes as an input memory leaks reported by Infer. For each defect, the filter generates the AST of the file in which the defect was found. The compile command and the list of compiled files are stored in `/builddir/infer-ast-log` (in chroot) which was generated during the build of the analysed package by compiler wrappers (see Section 3.2). For each memory leak, the filter looks for the variable into which the memory address was assigned. Then it checks for assignments where the variable is on the left side, between the last use of the variable's value (this information is taken from the Infer report as shown in Listing 3.6) and the end of the function. This approach identifies only a subset of the intentional memory leaks. It is because the filter checks for occurrences of assignment only in the syntactic order and not semantic (for example, loops can jump backward).

For generating the ASTs a three-step approach is used:

- **Using unmodified arguments of the compile command.** – Only the first argument (the compiler binary) is substituted for `clang` and immediately after that `-cc1` and `-ast-dump` (in the same order) are added. If this approach succeeds, no information is lost (e.g., in conditional builds). However, while experimenting with the AST generation, we have found that the `-cc1` option is incompatible with many other options, and therefore the command may fail even before the source code parsing begins.

- **Removing the `-c` option.** – The `-c` option, used to only compile the given source files, is very common. However, it is also incompatible with the `-cc1` option. Since we are only interested in AST generation and not binary generating, the failure of the command is irrelevant. As long as this command generates AST, this approach is successful. This approach is used if the first one fails to generate AST.
- **Removing all options except macros and includes.** – If even the second approach fails, we try to remove every option from the original compile command, except arguments starting with `-D` or `-I`. These options are compatible with the `-cc1` option and are used for the definition of macros (used in conditional compilation) and library path including. This way we will not lose information about conditional compilation.

```
1 lib/malloca.c:65: error: Memory Leak
2   memory dynamically allocated at line 52 by call to ‘malloc‘, is not
      freed after the last access at line 65, column 11.
3
4 ...
5
6 lib/malloca.c:65:11:  memory becomes unreachable here
```

Listing 3.6: An example of an Infer output about a memory leak defect reported in the sed package. The indication of the position of the last usage of the memory address is highlighted in red. The brief trace explanation `memory becomes unreachable here` is not accurate because the positional information points to the last known usage of the memory address value instead.

However, there are also cases in which all three above mentioned approaches fail or generate a partial AST only. The partial generation can happen, e.g., in cases where some external declarations are missing, but Clang is still able to produce parts of the AST, which it was able to parse without an error. If some parts of AST are missing, it is a problem only if the inspected part of the code has some AST parts missing. The Clang compiler generates its AST to `stdout`. However, even if Clang exits with an erroneous return code, the AST could have been generated. So the success of the generation cannot be checked using the return code of the given command. Instead, an occurrence of a `FunctionDecl` expression must be checked. The `FunctionDecl` in the Clang AST syntax indicates the start of a function declaration. There are also source files that do not have any function declaration, e.g., files with constant definitions, but since every memory leak reported by Infer is reported in a function and we only generate ASTs from a file in which this function is located, these exceptions cannot happen in our case. When the filter runs into an error (caused by missing parts of AST, the whole AST missing, or any other error) the worst-case scenario is that the examined defect is not filtered out.

3.4.5 Lowering the Severity of a Dead Store

The dead store defects alone cannot cause any serious problems, except minor performance issues. Due to this fact, the severity of `DEAD_STORE` defects reported by the Liveness plugin (described in Section 2.2) has been lowered from `ERROR` to `WARNING`. This will help to decrease the number of defects when developers filter only errors in the csmock output.

However, in some cases, these may indicate more serious problems, e.g., the in hostname package (see Section 4.1.1). To increase the usefulness of dead store reports in such cases in the future, it is intended to increase the severity to `ERROR` of dead stores which can lead to suspicious behavior. One of such cases is when the dead store is reported on a return value of a function. This indicates that the return value has been forgotten to check and it can potentially cause other errors. These possible errors should be reported by other plugins, but since Infer is not a sound static analyser it is not guaranteed.

3.5 Future Work on the Infer Plugin

The development of our Infer plugin for csmock does not end with this thesis. The plugin as well as the false positive filtering will remain in active development. For the plugin, it is planned to:

- Reduce impacts of incompatibility between Infer's internal Clang compiler and other used compilers, e.g., by identifying commonly used command line options not recognized by Infer's internal Clang.
- Test the plugin on a wider range of SRPM packages to discover potential problems.
- Create an RPM package with Infer to simplify the installation into chroot as well as on operating systems such as Fedora or CentOS.
- Improve AST generation (to lower the number of incorrectly generated ASTs) as hinted on the page of the AST linting framework in Infer[1] and compare this approach with the existing one.
- Improve the performance of compiler wrappers by parallelization of the build itself and the Infer's capture phase.

For false positive filtering, it is planned to raise the severity of dead stores which are reported on return values of functions. The memory leak filter can be further improved by deleting all the reported memory leaks when the software does not contain any `free` and similar functions. This indicates that memory freeing is left on the operating system. It is also possible to improve the memory leak filter by modifying the search for assignments to a given variable so that not only the syntactic order of statements is checked but also the semantic order.

To find any more filtering heuristics, it is necessary to manually identify more false positives from Infer reports, especially on rare defect types such as resource leaks. Manually going through hundreds of reported defects is very time-consuming, and moreover, it is not certain if any more common patterns in false positives will be found to produce new heuristics.

Therefore, we have decided to use a machine learning-based approach instead. When applying *machine learning* to false positive filtering it is possible to sort the reported defects by their probability to be true positive. This way no defects will be removed and thus no true positives will be left out. And yet the sorting will improve the usefulness of the report because developers can start checking the reported defects from the most promising ones.

[1] **AST Linting Framework** on Infer web – https://fbinfer.com/docs/checker-linters

On the development of this approach, we will be cooperating with researchers from IBM Research and RedHat. The approach is based on [28] from IBM Research. The goal is to include this machine learning-based *false positive sorter* into the Infer plugin for csmock and apply such a modified plugin on SRPM packages. Further, we will use the heuristics as additional features for machine learning.

Chapter 4

Experiments on Software Packages for Fedora

This chapter discusses results that Infer plugin for csmock achieved on selected SRPM packages for the Fedora operating system. It also discusses the results of the heuristics we proposed for filtering false positives in Section 3.4.

4.1 Results of Experiments

The experiments were run on a machine with the AMD Ryzen 3 3200U Mobile Processor (2 cores, 2.6 GHz base frequency), the AMD Raven2 graphics card and 5,8 GiB RAM. Versions of the operating system used for the testing were Fedora 33 and Fedora 29. At first, Fedora 29 was used, then the testing was moved to Fedora 33 due to the fact that LTO is enabled by default on Fedora 33 and newer versions (see Section 3.2). The tested version of Infer was 1.0.0[2]. All source packages have been downloaded from official repositories (every tested package is open-sourced). The source files of the packages can be found at Fedora Project sites[3]. Default versions of the packages were used for the analysis. All the packages were designed for the `x86_64` architecture. Each report was manually checked against the source code. The possible defects that were reported are sorted according to their type and truthfulness:

- **FP (False Positive)** – the reported defect cannot happen in a real-life execution (see Section 2.1) of the program.
- **Likely FP** – the reported defect is highly likely to be a FP, but we were not able to prove it. This is the case, when Infer reports a possible null dereference on a function parameter, but the function's documentation states, that the parameter will not ever have a null value, and possible values of the parameter cannot be determined, by going through the source code, e.g., because the value depends on the user input.
- **Unknown** – we were not able to determine whether the defect is a FP or a TP.
- **Intentional TP** – the reported defect can happen in a real-life execution, but it is intentional. Dead stores are common intentional defects. A dead variable is, e.g., left

[2]**Infer v1.0.0 release** – https://github.com/facebook/infer/releases/tag/v1.0.0.
[3]**SRPM packages download** – https://src.fedoraproject.org/.

in the code for some future use. Another common case is an intentional underflow which will not cause any problems. An example is shown in Listing 4.1 where, after the `while` loop, the variable `i` will underflow. However, the variable `i` is not used after the loop, so this construction is safe. Other very often kind of intentional defect is that of memory leaks (see Section 3.4.4).

- **TP (True Positive)** – the reported defect is true and can happen in a real-life execution of the program.

```
1  size_t i;
2  while(i--){
3      /* commands */
4  }
```

Listing 4.1: An example of an intentional variable underflow in C.

For experiments, we chose SRPM packages that contain commonly used system tools written in C/C++ such as `make`, `zip`, `hostname` `grep`, etc. The only exception is the cswrap package which is the only part of csmock written C and was therefore analysed. As the commonly used system tools have been evolving for decades, many errors have been uncovered and fixed, so there is less chance of detecting a serious error. Although this fact may indicate that more modern software will have more errors, this may not be true. Static analysis is typically part of the development process of modern software, so many errors are uncovered and fixed during the development. However, with newer software, there are more defects in terms of functionality (the program does something different than it should), which is debugged over time, but the kind of light-weight static analysis considered in this work is not able to detect this.

For each package, the number of LoC was obtained. The LoC is used to calculate statistics and classify the software by size [1]. LoC was calculated as the sum of LoC in all the `.c` and `.h` files (including comments) in the analysed packages. However, some source files may not have been analysed, which results in an inaccurate number of LoC. Some of the possible reasons why Infer could not analyse some files are (sorted by relevance):

- The file was not used in the build. The file is, e.g., architecture-specific.
- The Infer's capture phase failed due to compiler incompatibility (see Section 3.2).
- Some parts of the code were omitted due to conditional compilation.
- The analysis stopped after reaching a certain number of reported defects.
- An analysis timeouted for a given function.
- An analysis got into an inconsistent state when analysing.

The following statistics were observed and calculated for each analysed SRPM package:

- The number of reported defects of each defect type (memory leak, null dereference, etc.).

Table 4.1: The analysis and filtering statistics of the hostname package. The FPs per TP statistic is calculated as $(FP + LikelyFP + IntentionalFP)/TP$.

	FP	Likely-FP	Unknown	Int. TP	TP	LoC/defect	FP/TP
All	2	0	0	1	7	55.3	0.43
Filtered	2	0	0	0	0	69.13	0.14
Filtered [%]	100	0	0	0	0	–	–

- The number of FPs, Likely FPs, Unknown, Intentional TPs, and TPs defects.
- The number of filtered defects of each defect type.
- The number of filtered FPs, Likely FPs, Unknown, Intentional TPs, and TPs defects.
- The percentage of filtered defects in each category (FP, Likely FP, etc.).
- The number of LoC per defect before and after the filtering.
- The number of FPs per TP before and after the filtering. This is calculated as $(FP + LikelyFP + IntentionalFP)/TP$, because even if Intentional TP can happen in a real-life execution, it will not cause any problems. Since developers know about these defects, it is unnecessary or harmful, to report them, and so they should be filtered out.

4.1.1 Hostname Package

The hostname package provides the `hostname` utility, which is used to:

- display the system's DNS[1] name,
- display or set the system's hostname, or
- display or set the system's NIS[2] domain name.

The analysed version of the hostname package was 3.23. As this tool is very widespread and has been under development since 1994 (according to the copyright in the package), it is properly user-tested. The tool belongs to the net-tools project, which also contains tools such as `arp`, `ifconfig`, and `netstat`. Other tools belonging to this project can be found on the project's website[3]. The codebase is very small, it contains only a single source file with a length of 553 LoC, which is considered a micro-sized project.

The hostname package contains a fairly large number of reported defects for its size (see Table 4.1). This is mainly due to a large number of memory leaks because the memory is not being freed and it is up to OS to free the allocated memory. This technique is often used when the freeing operation is time-expensive, e.g., when deleting a doubly-linked list. For this reason, the mentioned memory leaks were not reported. The difference between an intentional TP memory leak and a TP memory leak is explained in Section 3.4.4.

[1] **Domain Name System (DNS)**.
[2] **Network Information Service (NIS)**.
[3] **Net-tools project** website – https://sourceforge.net/projects/net-tools/.

Table 4.2: The analysis and filtering statistics of the zip package. The FPs per TP statistic is calculated as $(FP + LikelyFP + IntentionalFP)/TP$.

	FP	Likely-FP	Unknown	Int. TP	TP	LoC/defect	FP/TP
All	106	0	28	8	28	436.36	4.07
Filtered	85	0	27	0	0	1,279	1.03
Filtered [%]	80.19	0	96.42	0	0	–	–

Most of the reported memory leaks do not have the required characteristics to be classified as an intentional TP by our standards. However, a manual check revealed that they are indeed intentional. This fact shows that a better definition of intentional memory leaks needs to be proposed. But it is problematic to propose a definition that will not classify all the reported memory leaks as intentional.

All the reported defects sorted by type and truthfulness can be seen in Figure C.1. The results of FP filtering (see Section 3.4) can be seen in Figure C.2. The statistics are shown in Table 4.1. After filtering, 100 % of FPs were removed, without removing any TPs. The LoCs per defect ratio was increased from 55.3 to 69.13 and the FPs per TP ratio was decreased from 0.43 to 0.14.

4.1.2 Zip Package

The zip package provides the `zip` utility, which is a compression and file packaging utility. For uncompression/unpackaging the `unzip` utility is used. The `unzip` utility is provided by a different package and thus was not analysed. The analysed version of the zip package was 3.0. As this tool is very widespread and has been under development since 1990 (according to the copyright in the package), it is properly user-tested. The tool belongs to the Info-ZIP project[1], which also contains tools such as `unzip`, `WiZ` or `MacZip`. The zip package contains 74,182 LoC, which is considered a medium-sized project.

The TPs we found were first compared with the latest publicly available (currently developed version Zip 3.1e is private) beta version Zip 3.1c[2] of the zip package, to verify that the TPs were still not fixed and that the end provider (Fedora) did not modify these packages before including them in their distribution (if so, these TPs could have been introduced by Fedora developers). After verification, all the TP null dereferences, memory leaks, and selected dead stores were reported[3] directly to Info-ZIP developers.

A dead store does not cause any error itself, but it is an indicator of a suspicious part of the code. Indeed, the reported dead stores showed an unchecked return value of a function, which in case of failure, led to a dereference of an uninitialized pointer, which, in most cases would end up with a program crash. Such cases only confirm the suitability of creating a dead store filter, which will emphasize these cases (see Section 3.5).

All the reported defects sorted by type and truthfulness can be seen in Figure C.3. The

[1] **Info-ZIP project** website – https://sourceforge.net/projects/infozip/.

[2] **Zip beta 3.1c** sources – https://sourceforge.net/projects/infozip/files/unreleased%20Betas/Zip%20betas/zip31c.zip/download.

[3] Defects reported to Info-ZIP developers – https://sourceforge.net/p/infozip/bugs/64/.

Table 4.3: The analysis and filtering statistics of the cswrap package. The FPs per TP statistic is calculated as $(FP + LikelyFP + IntentionalFP)/TP$.

	FP	Likely-FP	Unknown	Int. TP	TP	LoC/defect	FP/TP
All	1	0	0	0	1	931	1
Filtered	1	0	0	0	0	1,862	0
Filtered [%]	100	0	0	0	0	–	–

results of FP filtering (see Section 3.4) can be seen in Figure C.4. The statistics are shown in Table 4.2. After filtering, 80.19 % of FPs were removed and 96.42 % of Unknown defects were removed without removing any TPs. The LoCs per defect ratio was increased from 436.36 to 1,279, and the FPs per TP ratio was decreased from 4.07 to 1.03.

4.1.3 Cswrap Package

The cswrap package provides a generic compiler wrapper (see Section 3.2) used in csmock. Cswrap is being developed by Kamil Dudka from RedHat. Cswrap is relatively modern software so static analysers have already been applied on the cswrap. The cswrap package contains 1,862 LoC, which is considered a micro-sized project.

The TP we found, was a memory leak. Since this memory leak was reported on a variable into which memory was allocated only once (the allocation is not, e.g., in a loop, or often called function), then this memory leak does not pose a risk and therefore, was not reported to developers.

All the reported defects sorted by type and truthfulness can be seen in Figure C.5. The results of FP filtering (see Section 3.4) can be seen in Figure C.6. The statistics are shown in Table 4.3. After filtering, 100 % of FPs were removed without removing any TPs. The LoCs per defect ratio was increased from 931 to 1,862 and the FPs per TP ratio was decreased from 1 to 0.

4.1.4 Grep Package

The grep package provides the GNU implementation of the `grep` utility, which is used for regular expression matching in a text. As this tool is very widespread and has been under development since 1992 (according to the copyright in the package), it is properly user-tested. The tool belongs to the GNU project, which also contains tools such as `make` (see Section 4.1.5), `time`, `diff`, `nano`, `gzip` and many more. The list of all the tools can be found on the GNU website[1]. The grep package contains 123,307 LoC, which is considered a medium-sized project and also the biggest analysed SRPM package in this thesis.

All the reported defects sorted by type and truthfulness can be seen in Figure C.7. The results of FP filtering (see Section 3.4) can be seen in Figure C.8. The statistics are shown in Table 4.4. After filtering, 53.06 % of FPs were removed and 100 % of Likely FPs were removed without removing any TPs. The LoCs per defect ratio was increased from 1,868.29 to 3,736.58 and the FPs per TP ratio was decreased from 29 to 5.64.

[1] **GNU project** website – https://www.gnu.org/.

Table 4.4: The analysis and filtering statistics of the grep package. The FPs per TP statistic is calculated as $(FP + LikelyFP + IntentionalFP)/TP$.

	FP	Likely-FP	Unknown	Int. TP	TP	LoC/defect	FP/TP
All	49	7	3	6	1	1,868.29	29
Filtered	26	7	0	0	0	3,736.58	5.64
Filtered [%]	53.06	100	0	0	0	–	–

Table 4.5: The analysis and filtering statistics of the make package. The FPs per TP statistic is calculated as $(FP + LikelyFP + IntentionalFP)/TP$.

	FP	Likely-FP	Unknown	Int. TP	TP	LoC/defect	FP/TP
All	37	22	10	3	11	686.65	5.64
Filtered	14	1	7	0	0	934.3	4.27
Filtered [%]	37.84	4.55	70	0	0	–	–

4.1.5 Make Package

The make package provides the GNU implementation of the `make` utility, which is used to generate executables and other non-source files of a program from the program's source files. The analysed version was 4.3. As this tool is very widespread and has been under development since 1998 (according to the copyright in the package), it is properly user-tested. The tool belongs to the GNU project (see Section 4.1.4). The make package contains 56,992 LoC, which is considered a medium-sized project.

The TP null references we found are possible under circumstances where the input string to be parsed does not contain '('. The null dereference can happen on a return value from a function `strchr` which looks for an occurrence of the '(' in the given string. The `strchr` function can return null if the character was not found. The string represents internal statements in the program and should always contain '('. However, there may be an error in the generation of this statement, and '(' is not generated, in which case a null reference would certainly occur. Unfortunately, it was not possible to determine if the missing '(' can occur. However, we think that having these unchecked function calls in the code is still at least a bad practice. Before reporting the null dereferences to the developers, these reports need to be further investigated.

The rest of the TPs we found are dead stores that did not lead to any serious problems, so they were not reported. Often, dead store errors occurred when using conditional compilation where the concerned variable was initialized before a series of conditional blocks to shorten the code (instead of initialization in each block separately). If during preprocessing, all blocks using the given variable were removed, then the initialization of this variable is reported as a dead store. These types of dead stores could be considered Intentional TPs, but since this was not mentioned in the code, they are referred to as TPs.

All the reported defects sorted by type and truthfulness can be seen in Figure C.9. The results of FP filtering (see Section 3.4) can be seen in Figure C.10. The statistics are shown in Table 4.5. After filtering, 37.84 of % FPs were removed, 4.55 % of Likely FPs were removed and 70 % of Unknown defects were removed without removing any TPs. The LoCs

Table 4.6: The analysis and filtering statistics of the mlocate package. The FPs per TP statistic is calculated as $(FP + LikelyFP + IntentionalFP)/TP$.

	FP	Likely-FP	Unknown	Int. TP	TP	LoC/defect	FP/TP
All	10	0	0	0	5	1,089.73	2
Filtered	3	0	0	0	0	1,362.17	1.4
Filtered [%]	30	0	0	0	0	–	–

per defect ratio was increased from 686.65 to 934.3 and the FPs per TP ratio was decreased from 5.64 to 4.27.

4.1.6 Mlocate Package

The mlocate package provides `locate` and `updatedb` utilities. The `locate` tool is used to search file names using a pattern in the name database. The `updatedb` tool is used to update the name database. This package does not provide an implementation belonging to GNU findutils[1]. However, from a user's point of view, they should be identical. They differ mainly in the implementation of the name database. When GNU `updatedb` updates the database, the whole database is rewritten. For this newer implementation, the timestamps of the files and folders are maintained in the database, so only the changed files are reloaded [31]. The analysed version was 0.26. The mlocate package has been under development since 2005 (according to the copyright in the package) by Red Hat, Inc. The mlocate package contains 16,346 LoC, which is considered a small-sized project.

The results of the analysis revealed only a series of TP dead stores, but none of them pointed to a more serious problem, so they were not reported to the developers.

All the reported defects sorted by type and truthfulness can be seen in Figure C.11. The results of FP filtering (see Section 3.4) can be seen in Figure C.12. The statistics are shown in Table 4.6. After filtering, 30 % of FPs were removed without removing any TPs. The LoCs per defect ratio was increased from 1,089.73 to 1,362.17 and the FPs per TP ratio was decreased from 2 to 1.4.

4.1.7 Less Package

The less package provides the `less` utility. The `less` utility is used to reading files one page at a time with the ability to scroll up and down. It is similar to the `more` utility which allows to view a file and scroll down. As this tool is very widespread and has been under development since 1984 (according to the copyright in the package), it is properly user-tested. The tool belongs to the GNU project (see Section 4.1.4). The less package contains 28,776 LoC, which is considered a medium-sized project.

In this package, two TPs and two FNs[2] null dereferences were found. Since the null dereference TPs were unchecked values returned from `calloc`, we have checked for more occurrences of missing null check and we have discovered two FNs. The defects were reported as

[1] **GNU findutils** website – https://www.gnu.org/software/findutils/.
[2] **False Negative (FN)** – a defect missed by the static analyser (see Section 2.1).

Table 4.7: The analysis and filtering statistics of the less package. The FPs per TP statistic is calculated as $(FP + LikelyFP + IntentionalFP)/TP$.

	FP	Likely-FP	Unknown	Int. TP	TP	LoC/defect	FP/TP
All	55	1	13	0	2	405.3	28
Filtered	35	1	7	0	0	1,027.71	10
Filtered [%]	63.64	100	53.85	0	0	–	–

Table 4.8: The analysis and filtering statistics of the sed package. The FPs per TP statistic is calculated as $(FP + LikelyFP + IntentionalFP)/TP$.

	FP	Likely-FP	Unknown	Int. TP	TP	LoC/defect	FP/TP
All	20	7	1	4	3	2,947.54	10.3
Filtered	1	7	0	0	0	3,820.89	7.7
Filtered [%]	5	100	0	0	0	–	–

a repository issue[1] and were already fixed.

All the reported defects sorted by type and truthfulness can be seen in Figure C.13. The results of FP filtering (see Section 3.4) can be seen in Figure C.14. The statistics are shown in Table 4.7. After filtering, 63.64 % of FPs were removed, 100 % of Likely FPs were removed and 53.85 % of Unknown defects were removed without removing any TPs. The LoCs per defect ratio was increased from 405.3 to 1,027.71 and the FPs per TP ratio was decreased from 28 to 10.

4.1.8 Sed Package

The sed package provides the `sed` utility. It is a non-interactive command line text editor. The `sed` utility is often used for filtering, searching, and modifying text streams. The analysed version was 4.8. As this tool is very widespread and has been under development since 1989 (according to the copyright in the package), it is properly user-tested. The tool belongs to the GNU project (see Section 4.1.4). The sed package contains 103,163 LoC, which is considered a medium-sized project.

The results of the analysis revealed only a series of TP dead stores and unreachable code, but none of them could cause any serious problems, so they were not reported to the developers.

All the reported defects sorted by type and truthfulness can be seen in Figure C.15. The results of FP filtering (see Section 3.4) can be seen in Figure C.16. The statistics are shown in Table 4.8. After filtering, 5 % of FPs were removed and 100 % of Likely FPs were removed without removing any TPs. The LoCs per defect ratio was increased from 2,947.54 to 3,820.89 and the FPs per TP ratio was decreased from 10.3 to 7.7.

4.1.9 Tree Package

The tree package provides the `tree` utility. The utility is used to display a tree view of directories and files within. The printed files have color according to their type. The anal-

[1] Reported defects in Less package – https://github.com/gwsw/less/issues/151.

Table 4.9: The analysis and filtering statistics of the tree package. The FPs per TP statistic is calculated as $(FP + LikelyFP + IntentionalFP)/TP$.

	FP	Likely-FP	Unknown	Int. TP	TP	LoC/defect	FP/TP
All	5	0	0	0	17	177.86	0.29
Filtered	2	0	0	0	2	217.39	0.2
Filtered [%]	40	0	0	0	11.76	–	–

ysed version was 1.8.0. As this tool is very widespread and has been under development since 1996 (according to the copyright in the package), it is properly user-tested. The `tree` tool is being developed by Steve Baker. The tree package contains 3,913 LoC, which is considered a micro-sized project.

In the tree package, 2 TPs null dereferences, 12 TPs memory leaks, and 7 FNs null dereferences were found. In the project, a custom wrapper over `malloc`, which checks the return value for null is correctly implemented. However, this wrapper was not used in many cases. Some of these cases were reported by Infer and others were manually discovered. Memory leaks are reported due to the fact that a `realloc` wrapper is incorrectly implemented. The wrapper does not correctly handle situations when `realloc` fails. If a `realloc` fails to allocate new memory, the old memory is not freed by `realloc` but should explicitly be freed by the developer. Since no active public repository was found, the found defects were reported via email, but we haven't received any feedback yet.

All the reported defects sorted by type and truthfulness can be seen in Figure C.17. The results of FP filtering (see Section 3.4) can be seen in Figure C.18. The statistics are shown in Table 4.9. After filtering, 40 % of FPs were removed while removing 11.76 % of TPs. The LoCs per defect ratio was still increased from 177.86 to 217.39 and the FPs per TP ratio was decreased from 0.29 to 0.2.

4.1.10 Psmisc Package

The psmisc package provides utilities for managing Linux processes. The utilities in the package are:

- `fuser` – identifies processes that are using files or sockets.
- `killall` – kills processes given by name.
- `peekfd` – shows the data traveling over a file descriptor.
- `pstree` – shows currently running processes as a tree.
- `prtstat` – prints the contents of `/proc/<pid>/stat`.

The analysed version was 23.3. As these tools are very widespread and has been under development since 1993 (according to the copyright in the package), they are properly user-tested. The psmisc package contains 6,810 LoC, which is considered a micro-sized project.

Table 4.10: The analysis and filtering statistics of the psmisc package. The FPs per TP statistic is calculated as $(FP + LikelyFP + IntentionalFP)/TP$.

	FP	Likely-FP	Unknown	Int. TP	TP	LoC/defect	FP/TP
All	11	0	0	0	4	454	2.75
Filtered	5	0	0	0	0	681	1.5
Filtered [%]	45.45	0	0	0	0	–	–

Table 4.11: The overall analysis and filtering statistics on the SRPM packages. The FPs per TP statistic is calculated as $(FP + LikelyFP + IntentionalFP)/TP$.

	FP	Likely-FP	Unknown	Int. TP	TP	LoC/defect	FP/TP
All	296	37	55	22	79	850.52	4.49
Filtered	174	16	41	0	2	1785	2.14
Filtered [%]	58.78	43.24	74.55	0	2.53	–	–

The results of the analysis revealed only a series of TP dead stores, but none of them pointed to a more serious problem, so they were not reported to the developers.

All the reported defects sorted by type and truthfulness can be seen in Figure C.19. The results of FP filtering (see Section 3.4) can be seen in Figure C.20. The statistics are shown in Table 4.10. After filtering, 45.45 % of FPs were removed without removing any TPs. The LoCs per defect ratio was increased from 454 to 681 and the FPs per TP ratio was decreased from 2.75 to 1.5.

4.2 Overall Statistics

One of the main goals of this thesis was to decrease the number of false positives in Infer reports, to reduce the effort of developers while manually going through reported defects. Plain Infer has FPs per TP ratio on the analysed SRPM packages 4.49. Once our filter is enabled (see Section 3.4), the FPs per TP ratio is reduced to 2.14. This means that over 52 % of unuseful (FP, Likely FP, and Intentional TP) reported defects were filtered out. Since we were using heuristics, we have also filtered some TPs. But the overall FPs per TP ratio was reduced anyway, so it is acceptable if we miss some of the real defects. The overall LoCs per defect ratio was increased from 850.52 to 1,785. All the SRPM packages had a total of 415,905 LoC. Overall statistics of reported and filtered defects can be found in Table 4.11.

Some of the defects considered as TPs are still disputable as real true positives. We, however, did not see a way how to further narrow down the definition of the TPs in such a way that no cases of bugs with possible real negative consequences were left. We believe that such a further reduction would already have to be specific for the given codebase where the developers would have to specify, which defect they do not want to see on their own responsibility.

Since Infer is not a sound analyser, some false negatives were found when analysing the SRPM packages. FNs were not included in the statistics because their numbers do not necessarily correspond to the real numbers. These FNs were found because some null

dereferences in the given programs were reported under certain conditions, and we checked for all possible other locations where null dereferences could occur under the same conditions. All the FNs were reported to the developers of the given packages. These FNs can be caused by not analysing some parts of the code in which the error occurs, this can be caused by various reasons [33], for example:

- **An analysis failure** – if Infer gets into a state in which the monitored properties become inconsistent, the analysis of the current function is terminated.
- **An analysis timeout** – Infer has an internally set analysis timeout, and if it expires, the analysis for the current function will be terminated, otherwise, the analysis may not even finish.
- **Analysis stops** – the analysis may stop after reporting a certain number of defects in a function or a file.

Other possible reasons why Infer was not able to find certain defects depend on the specific plugin. This can be, for example, an incorrect approximation.

Chapter 5

Infer Wrapper for Incremental Analysis

This chapter describes the design and implementation of a new wrapper of Infer to be applied in the analysis phase whose goal is to improve the incremental analysis currently implemented in Infer. The principles behind the Infer's incremental analysis are described in Section 2.2.

In particular, while experimenting with an incremental analysis in Infer, we have discovered that Infer analyses wrongly some functions while running in the incremental mode. In particular, it analyses some functions unnecessarily and, what is worse, skips analysis of some function that should be re-analysed (for more information about the experiment, see Appendix D). Based on that observation, we have proposed a way that corrects the Infer's behavior. The proposed solution is to create a wrapper over Infer's analysis phase, which identifies a list of files that should be re-analysed. The wrapper decreases the incrementality from the level of functions to the level of files because no way was found to set Infer not to analyse a specific function.

5.1 Design

The proposed wrapper starts with a list of changed files and produces a list of files into which defects could have been propagated from the changes. The information about which files have been changed is obtained from the Infer's database. Infer internally marks every freshly captured file as changed. However, the build system is the one responsible for determining which files should be re-compiled and thus captured and marked as modified. So if the build system always re-compiles the whole code, then every single file from the codebase is marked as changed. However, almost every modern build system uses timestamps to determine which files do not have to be re-compiled.

After obtaining the list of changed files, the wrapper constructs a *file dependency graph*, which is very similar to a call graph (see Figure 2.1), except the nodes, do not represent functions but files instead. A file `X` is linked by an edge with a file `Y` if `X` contains a function that calls some function from `Y`. The information needed to construct the graph is also obtained from the Infer's database. To construct a file dependency graph, a list of all the project's files is needed together with a list of functions in each file and each function's

callees. The graph is constructed starting from the root file – the file that contains the `main()` function.

From the constructed file dependency graph, the wrapper obtains a list of files to analyse. The list consists of changed files and all of their parents all the way up to the root file. The list is then handed over to the Infer's analysis phase to perform the analysis only on the files from the produced list.

5.2 Implementation

Currently, the wrapper works only on C and C++ programs with exactly one `main()` function. The wrapper works on the current newest version of Infer – v1.1.0. The wrapper uses the `sqlite3` utility to query Infer's database. The files are loaded from the `source_files` table together with information about which files have been changed – the column `freshly_captured`. The list of callees is extracted from the binary data from the column `callees` from the table `procedures`.

After all the data are loaded, the dependency graph is generated by the *recursive algorithm* shown in Algorithm 5.1.

Algorithm 5.1: File dependency graph generation written in pseudo-python language.

```
1  allFiles.children ← [];
2  allFiles.parents ← [];
3  allFiles.isProcessed ← False;
4  currFile ← mainFile;
5  def createDependencyGraph(allFiles, currFile):
6      if currFile ∉ allFiles then
7          return;
8      end
9      for currFunc ∈ currFile.callees do
10         if currFunc.hasFile ∧ currFunc.file ∉ currFile.children then
11             currFile.children.Add(currFunc.file);
12             if currFile ∉ currFunc.file.parents then
13                 currFunc.file.parents.Add(currFile);
14             end
15             currFile.isProcessed ← True;
16             if currFunc.file.isNotProcessed then
17                 createDependencyGraph(allFiles, currFunc.file);
18             end
19         end
20     end
21     currFile.isProcessed ← True;
22     return;
23 end
```

After the file dependency graph is constructed, a list of files to analyse can be obtained by traveling from the changed files to the root of the graph and adding all the nodes along the way, as shown in Algorithm 5.2.

Algorithm 5.2: Obtaining the list of files to analyse from the file dependency graph.

```
1  def obtainFilesToAnalyse(changedFiles):
2      for file ∈ changedFiles do
3          for parent ∈ file.parents do
4              if parent ∉ changedFiles then
5                  changedFiles.Add(parent);
6              end
7          end
8      end
9      return changedFiles;
10 end
```

The wrapper stores all the files, which will not be analysed in the `.inferconfig` file in the current directory. The `.inferconfig` uses JSON format as shown in Listing 5.1. With the `skip-analysis-in-path` option of Infer [19, 37], we can set Infer not to analyse files with given paths. Unfortunately, no way was found to set Infer not to analyse certain functions, because of that, the incremental analysis is lowered from the level of functions to the level of files. However, even if some files are not analysed this way, the produced report will contain reported defects from all the previous analyses. This can be omitted by passing the `--incremental-report` option to the Infer wrapper. This option will remove all the reports reported in the files that were not analysed, by passing the `report-blacklist-path-regex` option to the Infer's analysis phase. An example of the `.inferconfig` file generated by the Infer wrapper is shown in Listing 5.1.

The `skip-analysis-in-path` option does not work only on the analysis phase but also on the capture phase – preventing given files from being captured. Due to this fact, the `.inferconfig` file must be deleted before running the next capture phase. Otherwise, some files need not be captured at all and thus not analysed. The user manual, together with an experiment on incremental analysis are described in Appendix D.

```
1 {
2     "skip-analysis-in-path": [ "callee.c", "constant.c" ],
3     "report-blacklist-path-regex": [ "callee.c", "constant.c" ]
4 }
```

Listing 5.1: An example of the generated .inferconfig file by the wrapper for incremental analysis.

Chapter 6

Experiments with Plain Infer

Apart from our work on the Infer plugin for csmock and apart from applying Infer through this plugin to SRPM packages, we have also applied Infer directly on multiple software projects. The purpose of these analyses was to find errors in the analysed software and also to examine which kinds of software Infer handles well and which not. In addition to default and non-default plugins of Infer coming from Facebook, plugins developed at Brno University of Technology were also used, namely:

- **L2D2 (Low Level Deadlock Detector)** [25] – a plugin for detecting low-level deadlocks in C or C++ programs.
- **Atomer** [17] – a plugin for detecting *atomicity violations* in C/C++ or Java programs.

6.1 Analysed Software

The analysed projects were a mixture of open-source and proprietary software. Because of that, some projects mentioned below do not contain information about their repositories. The analysed projects were:

- **Follow the Gap (CTU[1])** – an open-source[2] library, which provides a modified version of the *Follow the Gap algorithm*. The library was developed at the Czech Technical University in Prague. This project is written in C++ with the Make build system.
- **AQUAS Space Use Case code** – a multithreaded application for the platform Gaisler GR12RC with the Leon 3 dual-core processor. This platform is certified by ESA (European Space Agency) for flight purposes. One thread handles telemetry commands, another one does memory scrubbing. The application is a part of real satellite software. This application is written in C with the Make build system. Since this application is aimed to run on a Leon processor a cross-compiler `sparc-rtems-gcc` was used. The `sparc-rtems-gcc` compiler is not compatible with Infer's internal Clang. To overcome the incompatibility, many adjustments were made in cross-compiler's libraries, compile commands, and source files. The source files were

[1] **Czech Technical University in Prague (CTU)**.
[2] **Follow the Gap library** sources – https://github.com/CTU-IIG/f1t-ftg.

analysed separately because Infer was not able to the needed extract compile commands from the `Makefile`.

- **KCF tracker (CTU)** – open-source[1] software for *tracking the bounding box of an object* based on *kernelized correlation filters*. This software modifies the original version[2] to be able to run on the NVIDIA TX2 board and track rotating objects. This version has also an improved efficiency due to a parallelized algorithm. The project was developed at the Czech Technical University in Prague and it is written in C++ with a Make build system.

- **fast-DDS** – an open-source[3] implementation of the RTPS (Real Time Publish Subscribe) mechanism of DDS (Data Distribution Service). The RTPS protocol provides *publisher-subscriber communication* over unreliable communication protocols, such as UDP (User Datagram Protocol). The analysed versions were 1.9.3, 1.9.4, and 1.8.1. This project is written in C/C++ with the CMake build system.

- **fast-CDR** – an open-source[4] library that implements two *serialization mechanisms*. The first one is the standard CDR (Common Data Representation) serialization mechanism and the second, is a modified version of the standard with improved performance. The analysis was performed on version 1.0.12. The library is used in fast-DDS software and it is written in C++ with the CMake build system.

- **GNU coreutils** – a set of basic file, shell, and text manipulation utilities for the GNU operating system. All the utilities are open-sourced[5]. GNU coreutils provide utilities like `ls`, `rm`, `cat`, `mkdir`, and many others. The majority of utilities are written in C with the Make build system.

- **Gnulib** – an open-source[6] library for common GNU code, used by various GNU packages. The library was analysed as a part of the GNU coreutils analysis. The library is written in C with the Make build system.

- **Music** – a simple project for generating *digital signatures* based on an *elliptic curve digital signature algorithm*. This project is written in C with the Make build system. This project inspired us to analyse the OpenSSL library.

- **OpenSSL** – an open-source[7] cryptographic library that implements the SSL/TLS (Secure Sockets Layer/Transport Layer Security) protocol. The analysis was performed on version 1.1.1. The majority of the library is written in C with the Make build system.

- **mmwave** – software for *traffic monitoring* from the CAMEA company. This software is being developed on the Windows operating system and since Infer is only for Linux, many adjustments were made in preparation for the Infer analysis. The Make build system was using `.cmd` scripts. Since `.cmd` scripts are used in Windows,

[1] **KFC tracker** sources – https://github.com/CTU-IIG/kcf.
[2] An original version of the KCF tracker **KCF tracker** – https://github.com/vojirt/kcf.
[3] **fast-DDS** sources – https://github.com/eProsima/Fast-DDS.
[4] **fast-CDR** sources – https://github.com/eProsima/Fast-CDR.
[5] **GNU coreutils** sources – https://github.com/coreutils/coreutils.
[6] **Gnulib** sources – https://git.savannah.gnu.org/gitweb/?p=gnulib.git.
[7] **OpenSSL** sources – https://github.com/openssl/openssl.

the provided `Makefile` could not be used. The source files were analysed separately instead. Options for the needed compile commands (mostly -D for conditional compilation) were extracted from the `Makefile`. Hundreds of adjustments were made in libraries and source files to overcome compilation errors because the provided toolkit with libraries was also for the Windows operating system. The software was written in the C language.

- **Open122** – an open-source CCSDS (Consultative Committee for Space Data Systems) 122.0 codec. CCSDS 122.0 is a standard for lossy-to-lossless image data compression. The codec is written in C with the Make build system.
- **Arrowhead Framework** – an open-source[1] framework for developing enterprise Java applications. The framework is written in Java with the Maven build system. The version for analysis was 4.1.3. Infer was able to extract compilation commands from Maven, but the framework was written for a higher javac version than Infer's internal javac. This resulted in compilation errors. The newer javac version supported special class constructions. To resolve this problem a majority of classes would have to be rewritten and because this would be very time-consuming, the analysis was not performed.
- **Face Detector (CAMEA)** – an application for detecting human faces from the CAMEA company. The application is written in C with the Make build system.
- **Make and Model Recognition**[2] **(CAMEA)** – software for vehicle classification from the CAMEA company. The software is written in C with the Make build system.
- **CAMEA drivers** – various drivers from the CAMEA company were analysed. For example, a driver for DMA (Direct Memory Access). All of these drivers were aimed to be compiled with the GCC compiler, which supports assembly code directly inserted into source files. Infer's internal Clang does not support this. So the inserted assembly instructions have been removed to successfully run Infer's capture phase. All of these drivers were written in C with the Make build system.

The results of the Infer's analyses on the above mentioned projects have been reported to their developers (except for Gnulib, GNU Coreutils, fast-DDS, fast-CDR, and OpenSSL). Only a minority of the reported defects were checked, the rest were left to the developers of the respective projects. None of the checked reports indicated a serious bug. Nevertheless, these projects provided a range of experience with deploying Infer to different projects.

6.2 Software Not Suitable for Analysis by Infer

As we have already said, Infer can analyse software written in C/C++/Objective C, C#, or Java. The thesis aims at analysing C and C++ code (since the majority of UNIX utilities are written in C). However, even if the analysed software is written in one of the supported languages, that does not guarantee that Infer will be able to successfully handle such software. However, even if Infer can run an analysis on some software, that does not mean that the software is suitable for Infer, e.g., Infer can have a very high percentage of

[1] **Arrowhead Framework** sources – https://github.com/eclipse-arrowhead/core-java-spring.

[2] **Make and Model Recognition** website – https://www.camea.cz/en/its/vehicle-identification/make-and-model-recognition/.

FPs, the analysis can be time expensive or a lot of FNs can occur. One of the goals of the thesis was thus to determine the types of projects on which Infer has such problems. The following sections are based on experiences with analyses of the above-mentioned software as well as on SRPM packages.

Incompatible Compiler

Compiler incompatibility has proven to be the biggest obstacle when analysing software with Infer. To successfully finish the capture phase, Infer needs a working compilation command (see Section 2.2). However, the compilation command needs to be compatible with Infer's internal compiler (Clang for C/C++ and javac for Java). In a majority of analyses, the Clang compiler was not used as the project's compiler for C/C++. This resulted in invalid compilation commands. The only way to solve this problem is to manually adjust the compilation commands (removing incompatible options) and source files/libraries (removing/modifying incompatible constructions).

Unsupported Build System

Infer supports many build systems[1]. If some software uses a build system unknown to Infer, then the software will be hard to analyse. If the build system is very similar to the supported ones, then it is possible to tell Infer to behave to the used build system as to the one Infer recognizes. This can be done by using the `--force-integration command` command line option where `command` is one of the supported build systems.

However, even if the used build system is supported, that does not guarantee a successful capture phase. For example, if the Make script calls an external Shell script, Infer will not be able to capture compile commands from the Shell script. The simpler Make (or any other build system) script, the higher chance to successfully capture compile commands.

The problem with an unsupported build system or an unsuccessful capture phase on the supported one can be bypassed by creating a compiler wrapper as described in Section 3.2. This way Infer does not rely on a build system, but only on compatibility between Infer's internal compiler and the compiler which is used to compile the source codes (note that the compiler incompatibility problem also exists while letting Infer extract compile commands from the build script, so the wrapper technique does not introduce any new compatibility problems). Some issues need to be resolved while using a compiler wrapper. The problems and their solutions are listed in Section 3.2.

Global Variables with InferBO

The InferBO plugin (see Section 2.2) is not able to correctly handle global variables. Currently, if a global variable is used in a function, InferBO over-approximates this variable to $[-\infty, +\infty]$ for signed variables and $[0, +\infty]$ for unsigned variables. A similar over-approximation can occur even in loops with complex conditions. This over-approximation results in a lot of FPs since almost every operation on such over-approximated bounds will result in an overflow/underflow or accessing outside of the allocated buffer. The reports which over-approximate the bounds that much are currently filtered out by our InferBO filter (see Section 2.2).

[1] Supported build systems by Infer – https://fbinfer.com/docs/analyzing-apps-or-projects.

Complex Software

The more complex the software to be analysed is, the more time is needed to complete the analysis. Every checker has different time costs for analysing different code structures. Since Infer checkers can have their own timeouts for analysis, then the bigger the time cost, the bigger the chance for a timeout to stop the analysis on a function or a file [33]. Stopping the analysis before it finishes means that some parts of the code will not be analysed at all and FNs can occur. A general advice is to divide code into short functions. If a function has thousands of LoC, then the timeout will more likely stop the analysis before it is finished.

Large Functions

Some of Infer analyses are very memory expensive. The longer the analysed function is, the more memory the analysis typically needs. This problem can be solved by increasing RAM, e.g., through swap files or by simply increasing memory on a virtual machine. However, if the machine runs out of RAM during analysis, then the results are lost, and the analysis needs to be re-run. As hinted in [15], the needed RAM can be decreased by lowering the number of parallel jobs by the `-j N` option. The huge amount of needed RAM can be also caused by the InferBO plugin, a more detailed explanation, and possible solution is described in Section 2.2. However, a general advice is to divide the code into smaller functions (up to several hundreds LoC).

Conditional Compilation

Since Infer performs the analysis on a preprocessed code, some parts of the code enclosed in `#if`, `#ifdef`, or `#ifndef` directives – so-called *conditional blocks* – could have been removed. The removal of such blocks depends on defined macros (in a source file or by passing -D options to the compiler). This causes that the analysis was not performed on the whole code. It also causes problems in manual report checking. Infer does not keep track of which conditional blocks were skipped. This makes it hard to reproduce the same analysis results in a different environment. It is also harder to manually check if the report is indeed a real defect because developers need to know exactly which conditional blocks were used. To alleviate this problem, it is good to keep track of the used compilation commands, for example with a compiler wrapper as mentioned in Section 3.2.

Chapter 7

Conclusion

In this thesis, we have proposed solutions to mitigate three major negative features of the Facebook Infer static analyser, mainly for the context of using it to analyse Linux utilities shipped as SRPM packages. The negative features are the difficulty to deploy Infer on the given project, high numbers of false reports, and the time and space requirements.

To simplify the deployment of the Facebook Infer tool, an Infer plugin for the csmock tool was proposed to automatically run analyses on SRPM packages for the Fedora and CentOS operating systems. The plugin has been successfully tested on several SRPM packages for Fedora, that contain more than 400 KLoC in total. When manually reviewing the reports, errors were found in utilities such as `zip`, `less`, and `tree`. These errors have been reported to the developers and, in the case of the `less` tool, have already been fixed in the latest version of this tool.

To reduce the number of false reports, we have proposed a heuristic filter that removes reports that are likely false according to experience that we obtained by analysing Infer's reports. The filter has been integrated into the Infer plugin for the csmock tool and tested on several SRPM packages for Fedora. On the analysed packages, the filter was able to remove approximately 60 % of the false reports with a loss of only 2.5 % of the true errors.

While Facebook Infer implicitly provides a support for incremental analysis, which greatly decreases the time and space requirements when deploying the tool in continuous integration, our experiments revealed several shortcomings. To address these shortcomings, we have proposed a wrapper over Infer that identifies parts of the code that may have been affected by the changes instead of relying on Infer to select such code. Our wrapper somewhat reduces the level of incrementality (from functions to files), but unlike the original solution it – to the best of our knowledge – correctly recognizes code that needs to be re-analysed.

Our plans for the future aim mainly at further improvements of dealing with false alarms. Manual reviewing of Infer's reports is very time-consuming, and it is not guaranteed that it will be possible to propose further heuristics based on this review. Our future work will therefore not focus on filtering but on sorting. Specifically, we plan to use machine learning to sort the reports by the probability to be a real bug. Our previously proposed heuristics will be used as features for machine learning along with other metrics of analysis and source codes. In this area, we will be collaborating with IBM Research and Red Hat (based on the agreement at the online meeting that took place in April 2021).

Bibliography

[1] Aguilar, J., Sanchez, M., Fernandez, C., Rocha, E., Martinez, D. et al. The Size of Software Projects Developed by Mexican Companies. In: *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'14)*. Las Vegas, NV, USA: [b.n.], July 2014, p. 36:1–36:5.

[2] Berdine, J., Calcagno, C. and O'Hearn, P. W. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In: *Proceedings of the 4th International Symposium on Formal Methods for components and Objects (FMCO)*. Amsterdam, Netherlands: Springer, Berlin, Germany, November 2005, p. 115–137. DOI: 10.1007/11804192_6. ISBN 978-3-540-36750-5.

[3] Blackshear, S., Gorogiannis, N., O'Hearn, P. W. and Sergey, I. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* New York, NY, USA: Association for Computing Machinery, New York, NY, USA. october 2018, vol. 2, OOPSLA. DOI: 10.1145/3276514.

[4] Brown, M., Williams, C., Vidal, S. et al.. *Mock - Linux Man Page* [online]. [cit. 2021-05-11]. Available at: https://linux.die.net/man/1/mock.

[5] Bygde, S. *Static WCET Analysis Based on Abstract Interpretation and Counting of Elements*. Västerås, 2010. Bachelor's thesis. Mälardalen University,.

[6] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P. et al. Moving Fast with Software Verification. In: *Proceedings of the NASA Formal Methods Symposium 2015 (NFM)*. Pasadena, CA, USA: Springer, Berlin, Germany, April 2015, p. 3–11. DOI: 10.1007/11804192_6. ISBN 978-3-540-36750-5.

[7] Cook, B. Formal Reasoning About the Security of Amazon Web Services. In: *Proceedings of the International Conference on Computer Aided Verification 2018 (CAV)*. Oxford, UK: Springer, Berlin, Germany, July 2018, p. 38–47. DOI: 10.1007/978-3-319-96145-3_3. ISBN 978-3-319-96145-3.

[8] Cousot, P. *Abstract Interpretation in a Nutshell* [online]. January 2010 [cit. 2021-05-10]. Available at: https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html.

[9] Distefano, D., Fähndrich, M., Logozzo, F. and O'Hearn, P. W. Scaling static analyses at Facebook. *Communications of the ACM*. 2019, vol. 62, no. 8, p. 62–70. DOI: 10.1145/3338112. ISSN 1557-7317.

[10] Dudka, K. *Csmock - Linux Man Page*. Brno, Czech Republic: Red Hat Czech s.r.o.

[11] Dudka, K. Fully Automated Static Analysis of Fedora Packages. In: *Flock*. Prague, Czech Republic: [b.n.], August 2014. Available at: https://kdudka.fedorapeople.org/static-analysis-flock2014.pdf.

[12] Dudka, K. Static Analysis and Formal Verification at Red Hat. In: *13th Alpine Verification Meeting (AVM'19)*. Brno, Czech Republic: [b.n.], September 2019. Available at: https://kdudka.fedorapeople.org/avm19.pdf.

[13] Ebalard, A., Mouy, P. and Benadjila, R. Journey to a RTE-free X.509 parser. In: *Proceedings of the Symposium sur la sécurité des technologies de l'information et des communications(SSTIC)*. Rennes, France: [b.n.], June 2019, p. 171–200. ISBN 78-2-9551333-4-7.

[14] Farell, M. L. *C++11s take on retain cycles* [online], 12. December 2014 [cit. 2021-05-11]. Available at: https://vertostudio.com/gamedev/?p=298.

[15] Gorogiannis, N. *Running out of memory on a large project* [online]. March 2019 [cit. 2021-05-13]. Available at: https://github.com/facebook/infer/issues/1072.

[16] Grigore, R. *What would be needed to extend this tool to more languages?* [online]. December 2020 [cit. 2021-05-10]. Available at: https://github.com/facebook/infer/issues/1368.

[17] Harmim, D. *Static Analysis Using Facebook Infer to Find Atomicity Violations*. Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology.

[18] Johnson, B., Song, Y., Murphy Hill, E. and Bowdidge, R. Why don't software developers use static analysis tools to find bugs? In: *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, New York, NY, USA, May 2013, p. 672–681. DOI: 10.1109/ICSE.2013.6606613. ISSN 1558-1225.

[19] Kotulski, A. *How to include project folders or exclude dependency folders during analysis of iOS project?* [online]. May 2016 [cit. 2021-05-16]. Available at: https://github.com/facebook/infer/issues/364.

[20] Kwangkeun, Y. *Inferbo: Infer-based buffer overrun analyzer* [online], 6. February 2017 [cit. 2021-05-10]. Available at: https://research.fb.com/blog/2017/02/inferbo-infer-based-buffer-overrun-analyzer/.

[21] Muske, T. B., Baid, A. and Sanas, T. Review efforts reduction by partitioning of static analysis warnings. In: *Proceedings of the 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Eindhoven, Netherlands: IEEE, New York, NY, USA, September 2013, p. 106–115. DOI: 10.1109/SCAM.2013.6648191.

[22] Rijnard van Tonder and Claire Le Goues. Static automated program repair for heap properties. In: *ICSE '18: Proceedings of the 40th International Conference on Software Engineering*. Gothenburg, Sweden: Association for Computing Machinery, New York, NY, USA, May 2018, p. 151–162. DOI: 10.1145/3180155.3180250. ISBN 978-1-4503-5638-1.

[23] Rob van Glabbeek. *Rice's theorem* [online]. March 2002 [cit. 2021-05-10]. Available at: http://kilby.stanford.edu/~rvg/154/handouts/Rice.html.

[24] Villiard, J. and Gorogiannis, N. *Infer is not deterministic* [online]. June 2019 [cit. 2021-05-10]. Available at: https://github.com/facebook/infer/issues/1110.

[25] Vladimír, M. *Static Analysis Using Facebook Infer Focused on Deadlock Detection*. Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology.

[26] Zheng, Y. *Out of memory when the bufferoverrun checker enabled* [online]. April 2020 [cit. 2021-05-10]. Available at: https://github.com/facebook/infer/issues/1246.

[27] Zheng, Y. *Parallelism gives inconsistent results* [online]. March 2020 [cit. 2021-05-10]. Available at: https://github.com/facebook/infer/issues/1239.

[28] Zheng, Y., Pujar, S., Lewis, B., Buratti, L., Epstein, E. et al. D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Madrid, Spain: IEEE, New York, NY, USA, May 2021, p. 111–120. DOI: 10.1109/ICSE-SEIP52600.2021.00020. ISBN 978-1-6654-3869-8.

[29] *ARC vs. GC* [online]. [cit. 2021-05-11]. Available at: https://docs.elementscompiler.com/Concepts/ARCvsGC/.

[30] *Constructors* [online]. [cit. 2021-05-11]. Available at: https://isocpp.org/wiki/faq/ctors#static-init-order.

[31] *Debian – Details of package mlocate in jessie* [online]. [cit. 2021-05-12]. Available at: https://packages.debian.org/jessie/mlocate.

[32] *Benign Data-Races* [online]. Oracle Corporation, 2010 [cit. 2021-05-10]. Available at: https://docs.oracle.com/cd/E19205-01/820-0619/gecqt/index.html.

[33] *Infer | Need help?* [online]. February 2015 [cit. 2021-05-12]. Infer v1.1.0. Available at: https://fbinfer.com/docs/support/#why-infer-doesnt-find-a-particular-bug.

[34] *Infer Static Analyzer* [online]. February 2015 [cit. 2021-05-10]. Available at: https://fbinfer.com/.

[35] *List of all issue types* [online]. February 2015 [cit. 2021-05-10]. Infer v1.1.0. Available at: https://fbinfer.com/docs/all-issue-types/.

[36] *Separation logic and bi-abduction* [online]. February 2015 [cit. 2021-05-10]. Infer v1.1.0. Available at: https://fbinfer.com/docs/separation-logic-and-bi-abduction/.

[37] *[ObjC] Skip analysis on file or part of File* [online]. April 2017 [cit. 2021-05-18]. Available at: https://github.com/facebook/infer/issues/633.

[38] *Fragments | Android Developers* [online]. AndroidDev, March 2020 [cit. 2021-05-10]. Available at: https://developer.android.com/guide/fragments.

[39] *Chroot* [online]. May 2021 [cit. 2021-05-11]. Available at: https://wiki.archlinux.org/title/Chroot.

[40] *Infer Analyze Manual* [online]. Facebook Inc., March 2021 [cit. 2021-05-10]. Infer v1.1.0. Available at: https://fbinfer.com/docs/man-infer-analyze.

[41] *Infer Capture Manual* [online]. Facebook Inc., March 2021 [cit. 2021-05-10]. Infer v1.1.0. Available at: https://fbinfer.com/docs/man-infer-capture/.

Appendix A

Contents of the Included Storage Media

The attached memory media have the following structure:

- `csmock/` – Source codes of csmock with the Infer plugin.
- `incremental-analysis/` – Source codes of the Infer wrapper for the incremental analysis. In this directory, there is also an experiment that demonstrates shortcomings in Infer's incremental analysis.
- `packages-results/` – This directory contains results and Infer's intermediate files of every analysed SRPM package listed in this thesis. Every subdirectory have the following structure:
  - `all.csdiff` – A list of all the defects reported by Infer in the csdiff format.
  - `filtered.csdiff` – A list of defects that were not removed by our FP filter. The list is in the csdiff format.
  - `infer-out/` – A directory that contains Infer's intermediate files. On this directory, the analysis can be re-run, e.g., with different plugins.
  - `removed.csdiff` – A list of defects that were removed by our FP filter. The list is in the csdiff format.
  - `report.txt` – A list of all the defects reported by Infer in the text format. Each defect is labeled according to its truthfulness.
- `text/` – Source codes of this thesis for `OverLeaf`.
- `README.md` – A description of contents of the included storage media in the `Markdown` format.
- `xberan46.pdf` – The PDF version of this thesis.

Appendix B

Installation and User Manual of Csmock with the Infer Plugin

The Infer plugin is not included in the csmock's official repository and thus cannot be installed as an RPM package via DNF. The csmock with Infer plugin needs to be installed from sources instead. The following installation instructions are appliable to Fedora and CentOS operating systems.

The following packages need to be installed before installation of csmock:

- `make`, `cmake`, `mock`, `cswrap`, `cscppc`, `csdiff`, and `python3`.

Run the following command to install all the dependencies:

```
sudo dnf install --assumeyes make cmake mock cswrap cscppc csdiff python3
```

User must be added to mock group before running csmock. This can be done by running the command:

```
sudo usermod -a -G mock USER
```

Source files can be downloaded from the repository with the following command:

```
git clone https://github.com/TomasBeranek/csmock
```

Source files are also stored in the included storage media, see Appendix A.

Installation of csmock can be done by running the following commands:

```
cd csmock
sudo make install
```

To download the binary version of Infer visit https://github.com/facebook/infer/releases or use the following command (the plugin was tested with Infer v1.0.0):

```
wget https://github.com/facebook/infer/releases/download/\
v1.0.0/infer-linux64-v1.0.0.tar.xz
```

```
sudo mv infer-linux64-v1.0.0.tar.xz /opt
   # or use "--infer-archive-path $PWD/infer-linux64-v1.0.0.tar.xz"
   # when invoking csmock
```

To analyze a package with csmock with enabled Infer plugin use the following command:

```
csmock -t infer package.src.rpm
```

Infer plugin currently supports the following command-line options:

`--infer-analyze-add-flag` – appends the given flag (except `-o`), when calling `infer analyze`. The option can be used multiple times.

`--infer-archive-path` – the plugin uses the given `.tar.xz` archive with the binary release of Infer. Default location is `/opt/infer-linux*.tar.xz`.

`--no-infer-filter` – disables false positive filter. The Infer output will be only transformed to csmock output.

`--no-infer-biabduction-filter` – disables Bi-abduction filter.

`--no-infer-inferbo-filter` – disables InferBO filter.

`--no-infer-uninit-filter` – disables Uninit filter.

`--no-infer-memory-leak-filter` – disables memory leak filter.

`--no-infer-dead-store-filter` – disables dead store filter.

To analyze a package with csmock with enabled Infer plugin with the archive in a different location than `/opt` use the following command:

```
csmock --infer-archive-path PATH -t infer package.src.rpm
```

To analyze a package with csmock with enabled Infer plugin without false positive filtering use the following command:

```
csmock --no-infer-filter -t infer package.src.rpm
```

To invoke csmock with multiple analyzers use the following command:

```
csmock -t infer -t cppcheck -t bandit package.src.rpm
```

Csmock produces a `package.tar.xf` archive with the results from all the analysers. After extracting the archive, the results produced by Infer in csdiff format are stored in `package/debug/uni-results/infer-results.err`.

Appendix C

Detailed Results of the Analysis on SRPM packages

This appendix contains graphs with the results of the analyses on SRPM packages. The analyzed packages were – hostname, zip, cswrap, grep, make, mlocate, less, sed, tree, and psmisc. The packages are described in more detail in Section 4. For each package there are generated two graphs:

- A graph with all issues sorted by issue type and truthfulness.
- A graph with all the removed issues sorted by issue type and truthfulness.

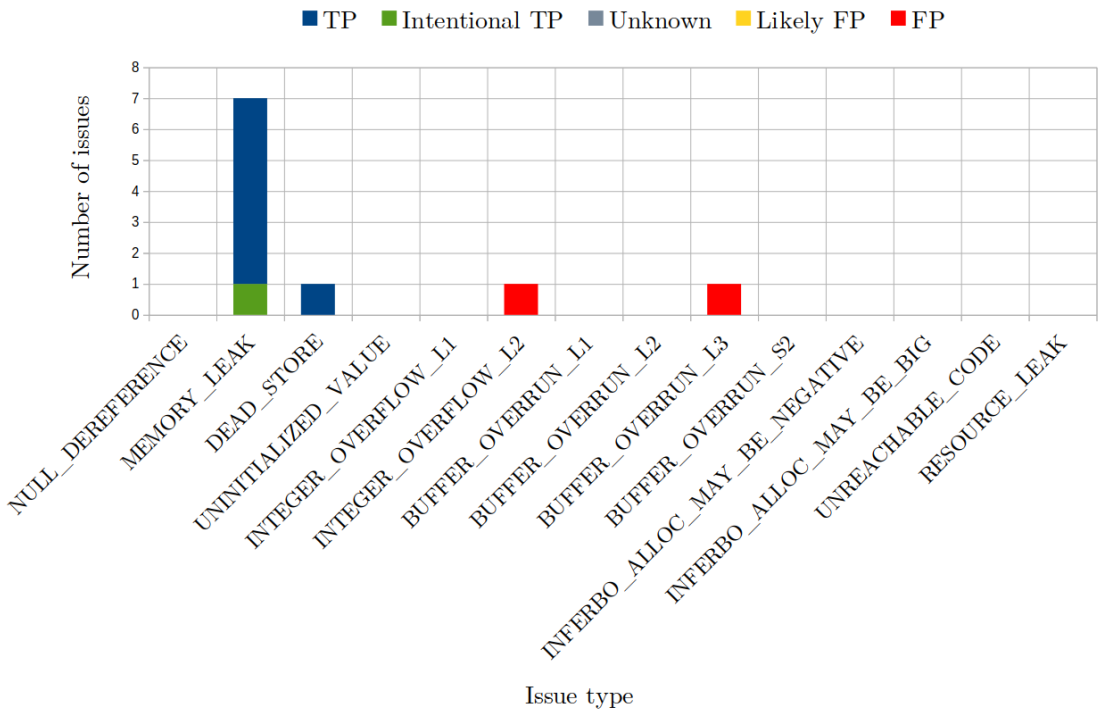

Figure C.1: All issues reported by Infer on the hostname package. Each issue was manually inspected and categorized according to its truthfulness.

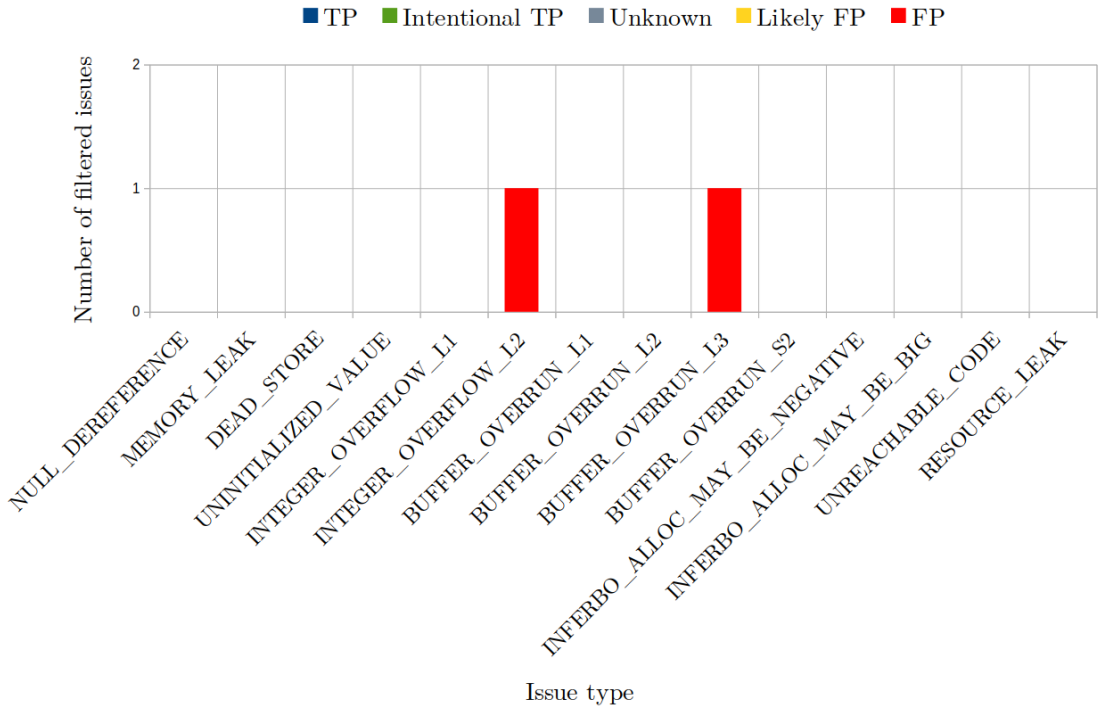

Figure C.2: Issues filtered by heuristics for filtering on the results of hostname package. Each filtered issue is categorized by its truthfulness.

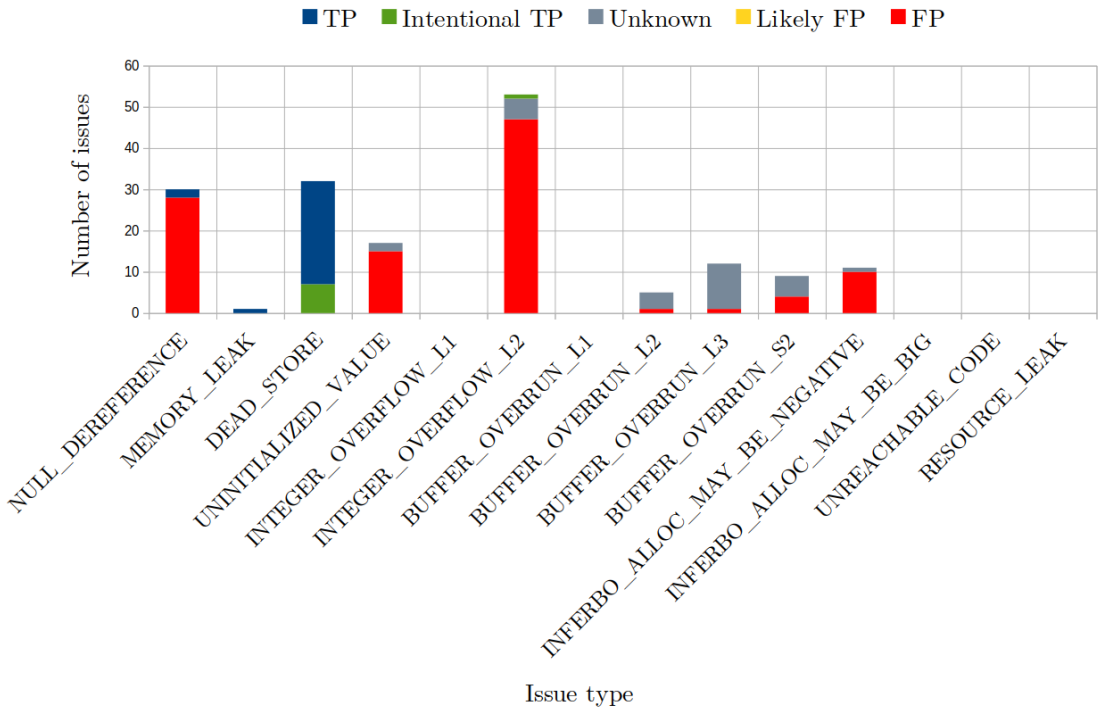

Figure C.3: All the issues reported by Infer on the zip package. Each issue was manually inspected and categorized according to its truthfulness.

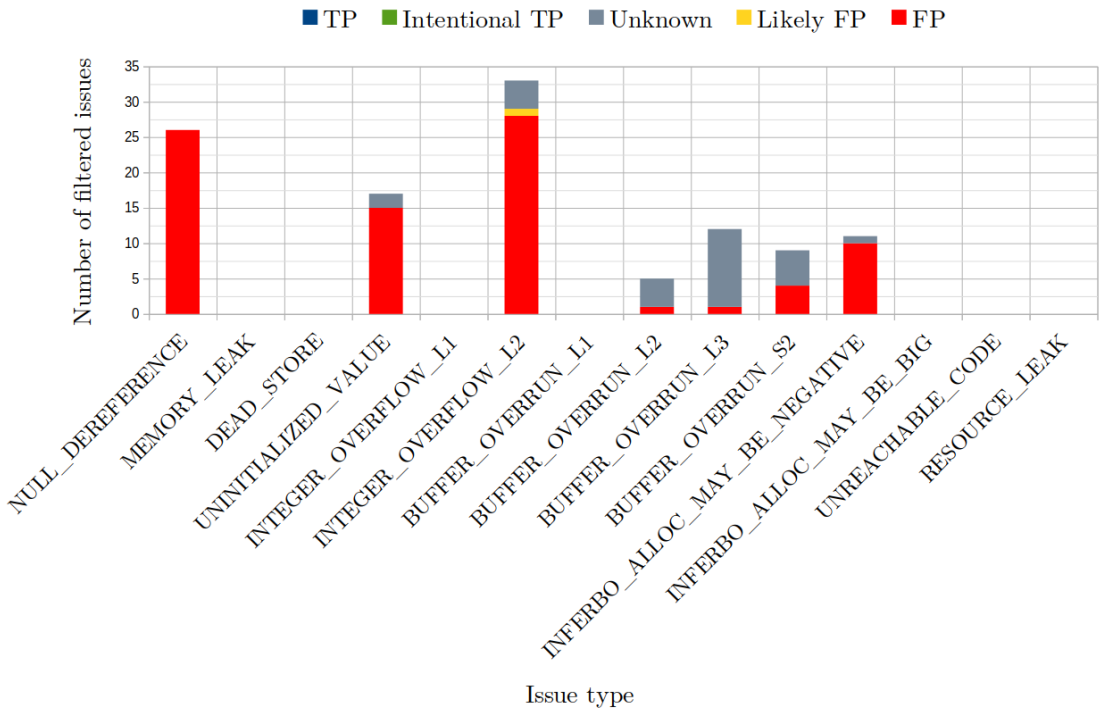

Figure C.4: Issues filtered by heuristics for filtering on the results of zip package. Each filtered issue is categorized by its truthfulness.

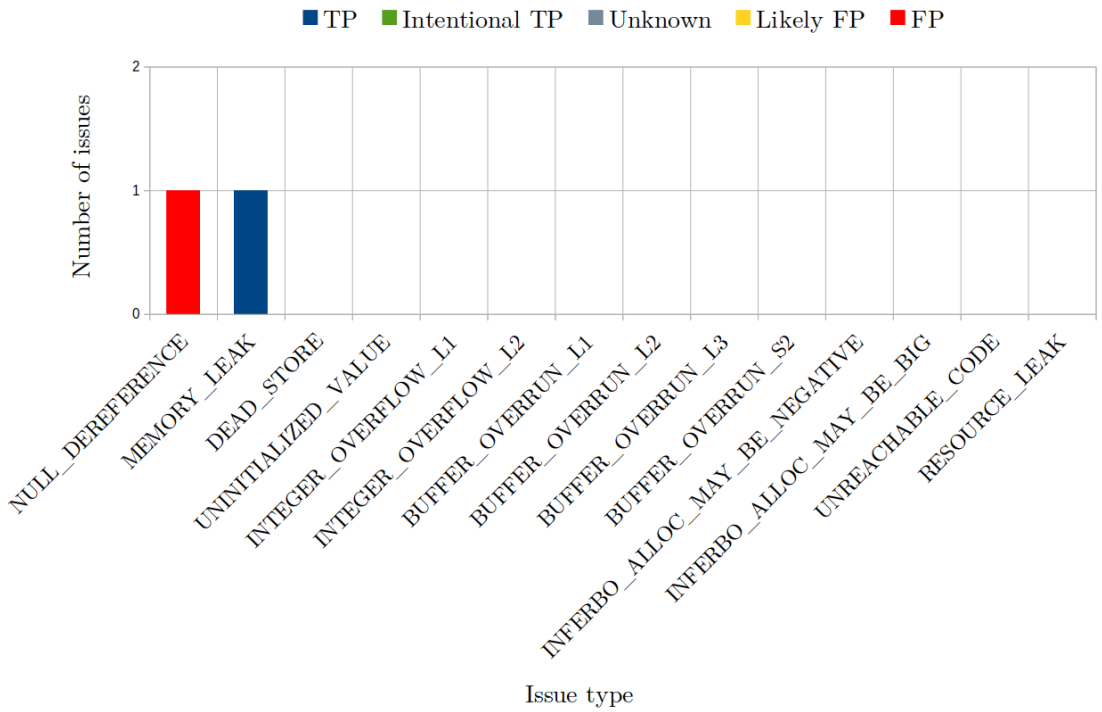

Figure C.5: All the issues reported by Infer on the cswrap package. Each issue was manually inspected and categorized according to its truthfulness.

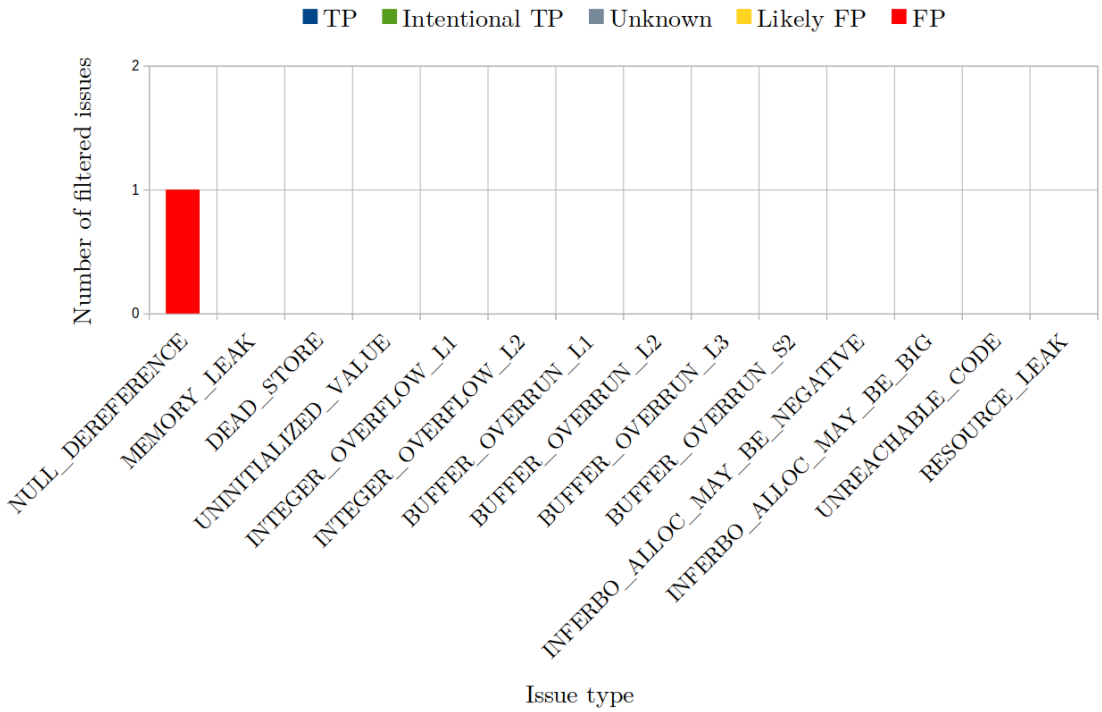

Figure C.6: Issues filtered by heuristics for filtering on the results of cswrap package. Each filtered issue is categorized by its truthfulness.

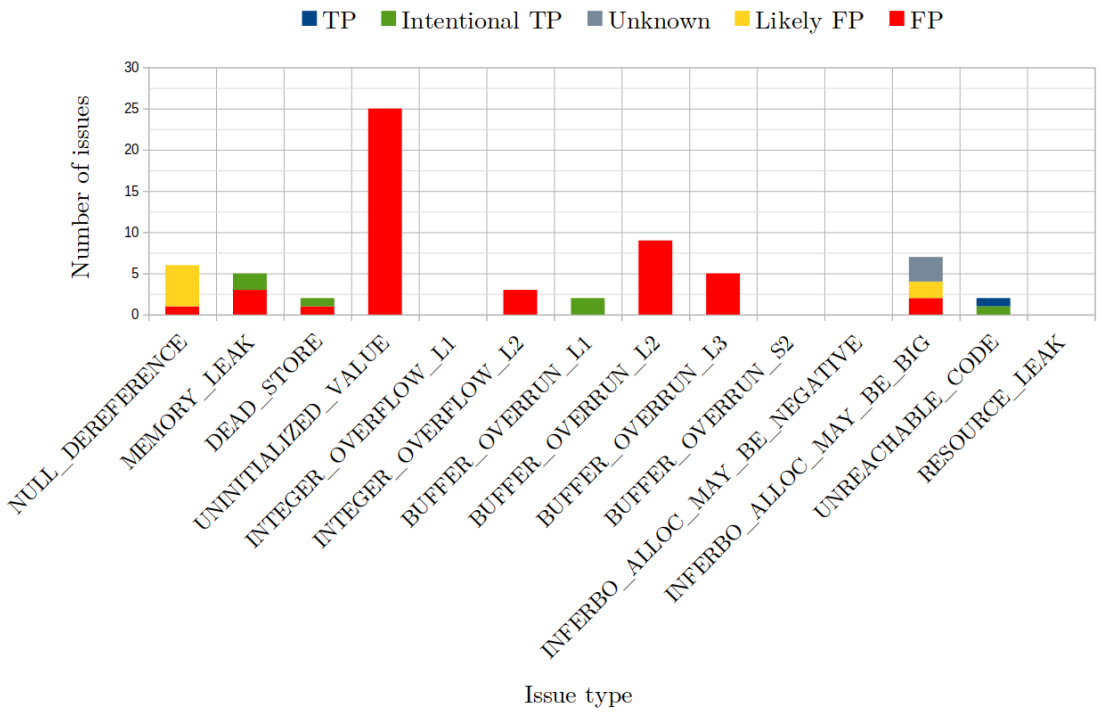

Figure C.7: All the issues reported by Infer on the grep package. Each issue was manually inspected and categorized according to its truthfulness.

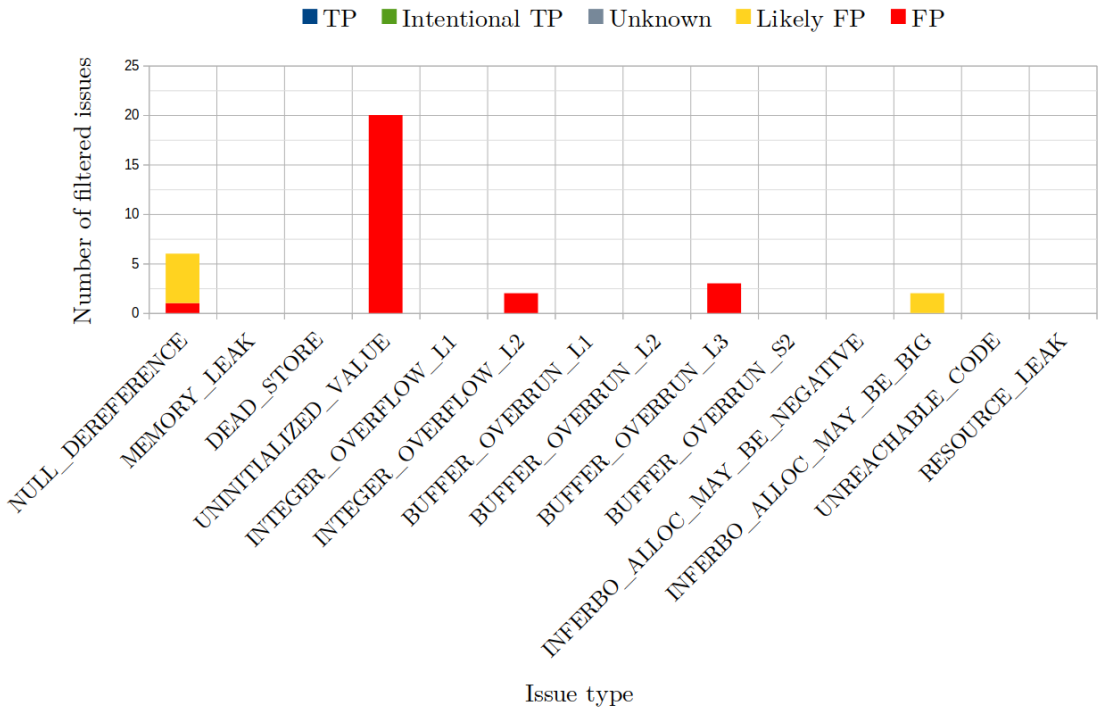

Figure C.8: Issues filtered by heuristics for filtering on the results of the grep package. Each filtered issue is categorized by its truthfulness.

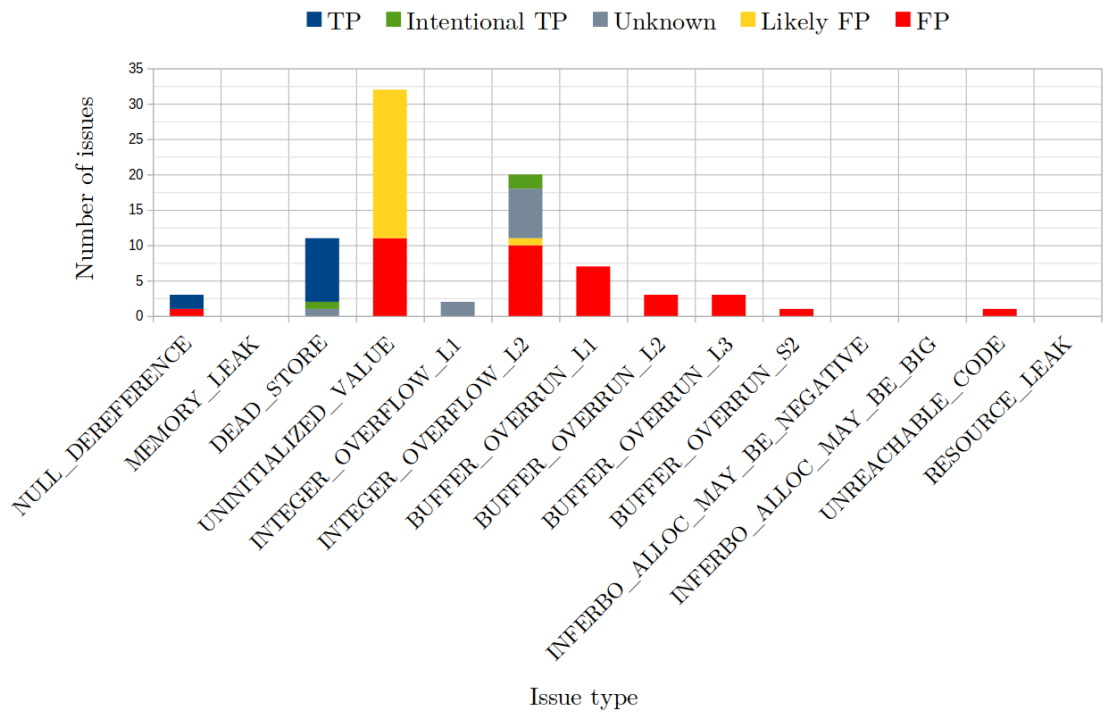

Figure C.9: All the issues reported by Infer on the make package. Each issue was manually inspected and categorized according to its truthfulness.

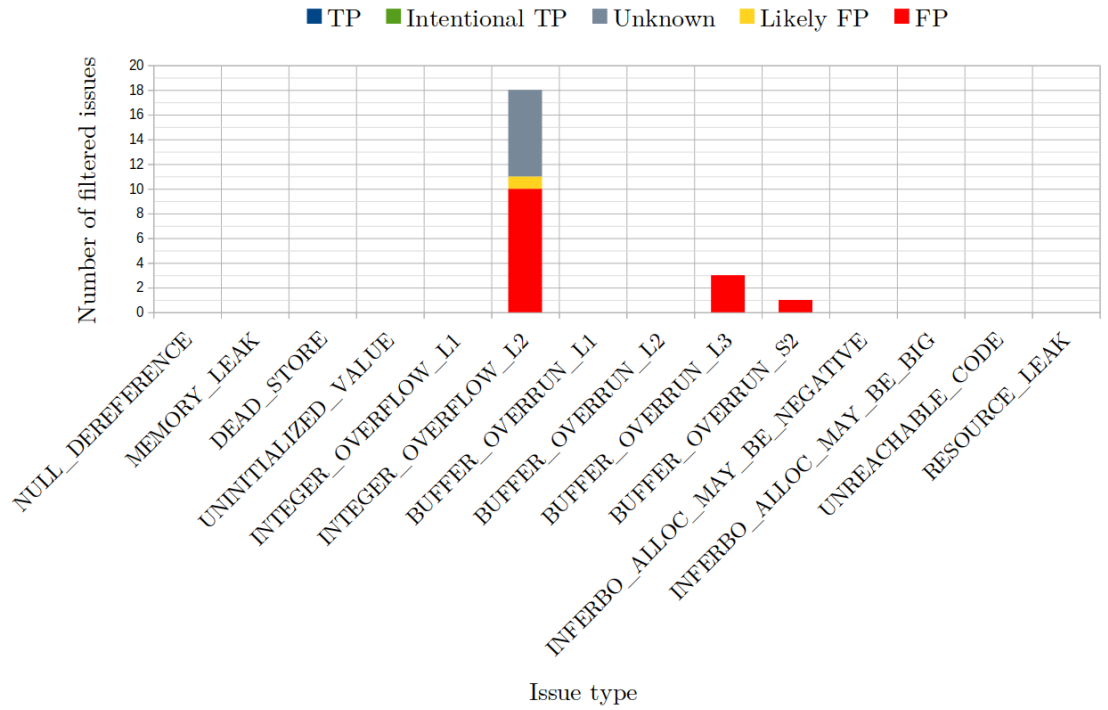

Figure C.10: Issues filtered by heuristics for filtering on the results of the make package. Each filtered issue is categorized by its truthfulness.

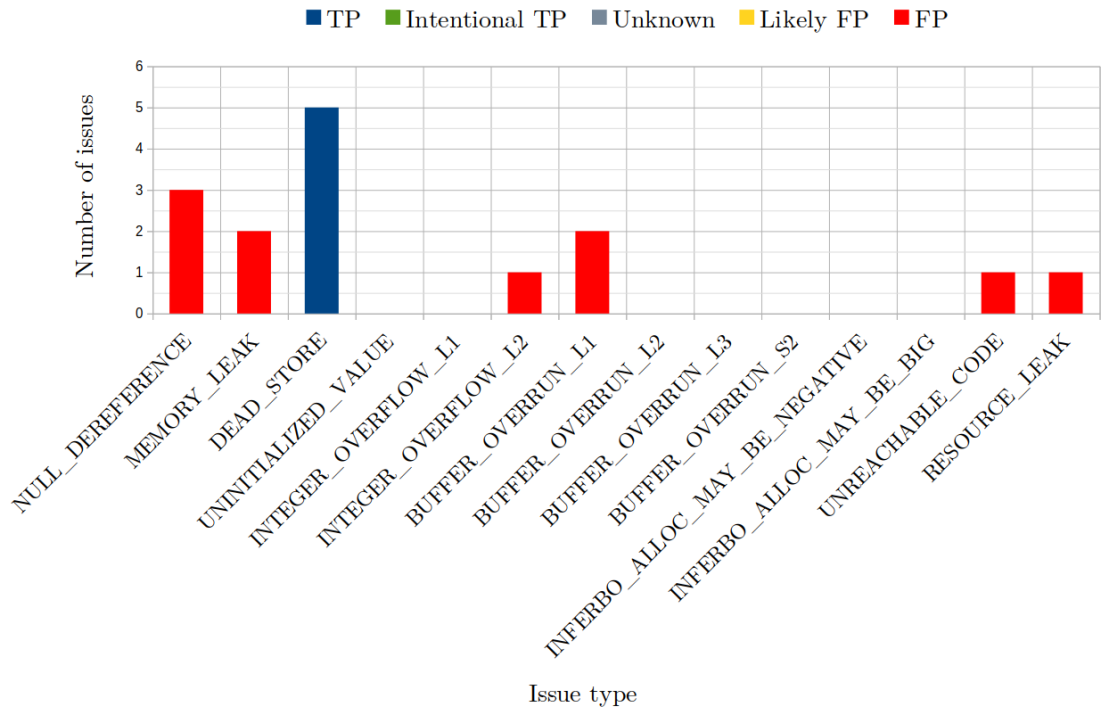

Figure C.11: All the issues reported by Infer on the mlocate package. Each issue was manually inspected and categorized according to its truthfulness.

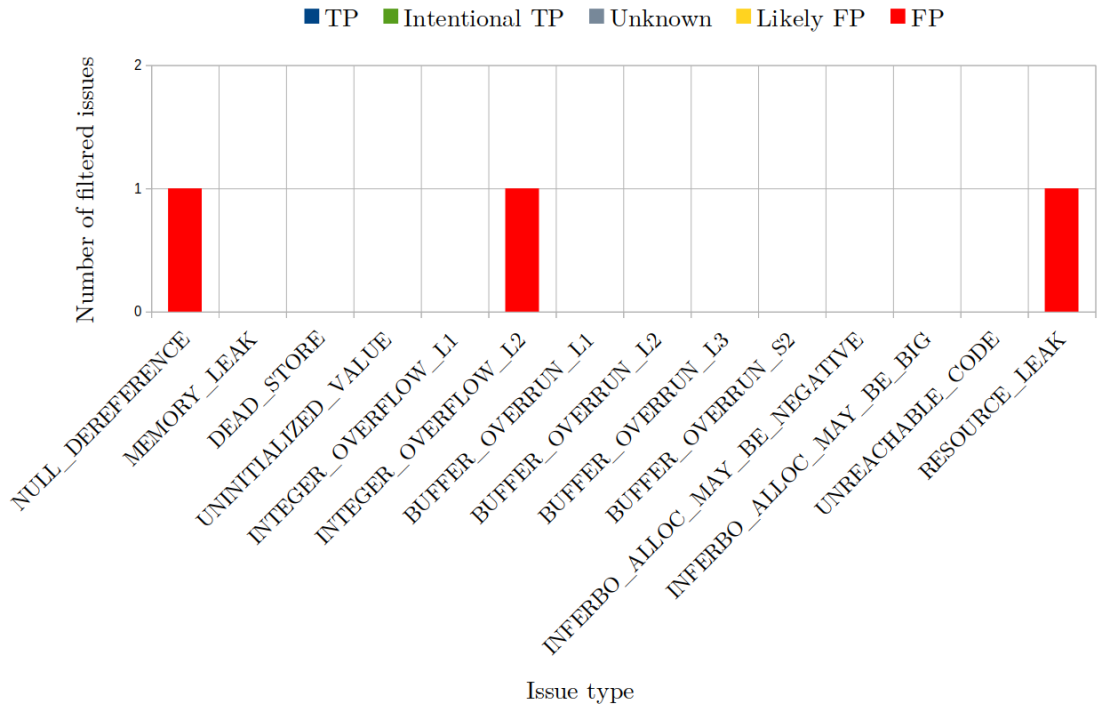

Figure C.12: Issues filtered by heuristics for filtering on the results of the mlocate package. Each filtered issue is categorized by its truthfulness.

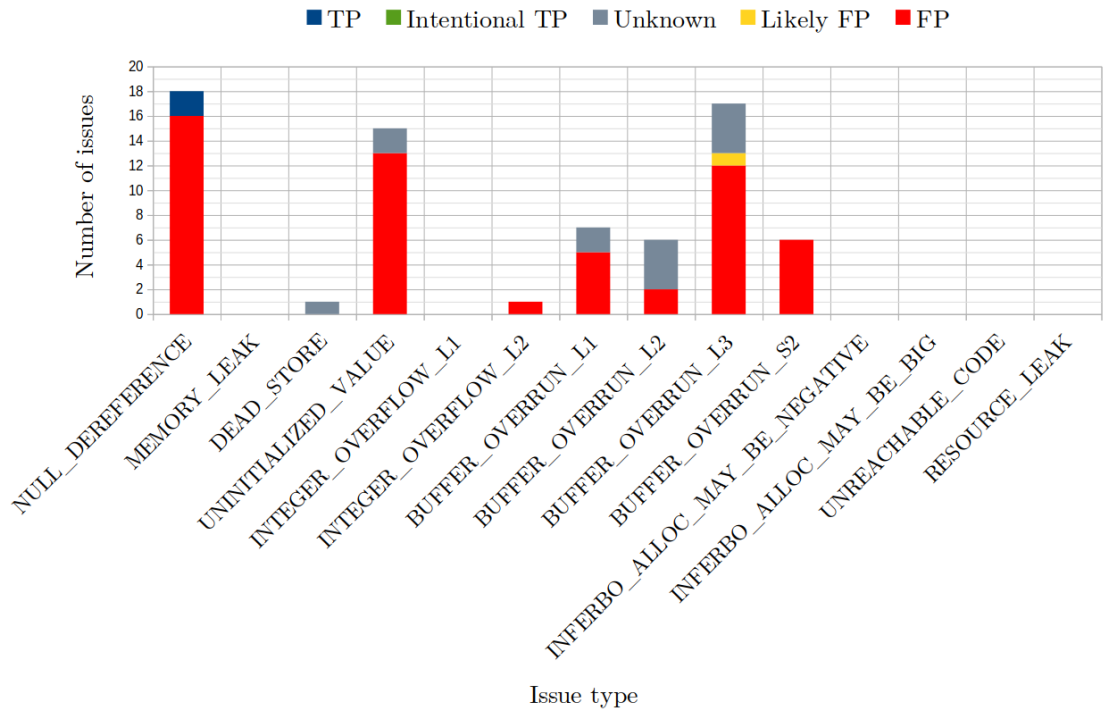

Figure C.13: All the issues reported by Infer on the less package. Each issue was manually inspected and categorized according to its truthfulness.

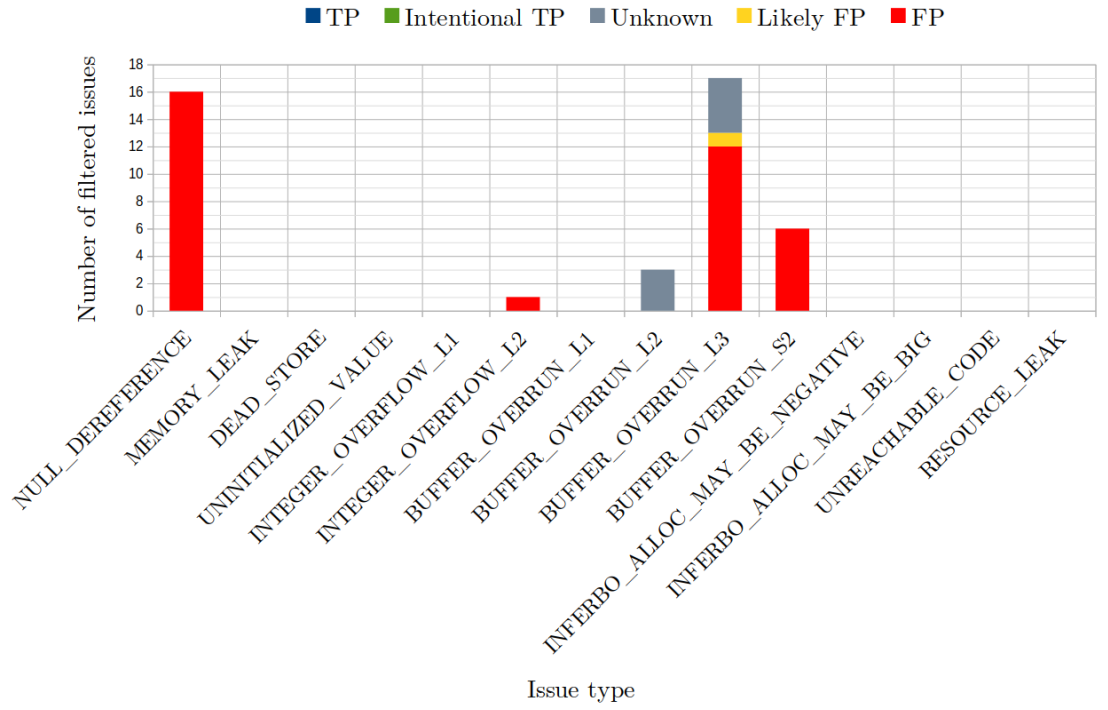

Figure C.14: Issues filtered by heuristics for filtering on the results of the less package. Each filtered issue is categorized by its truthfulness.

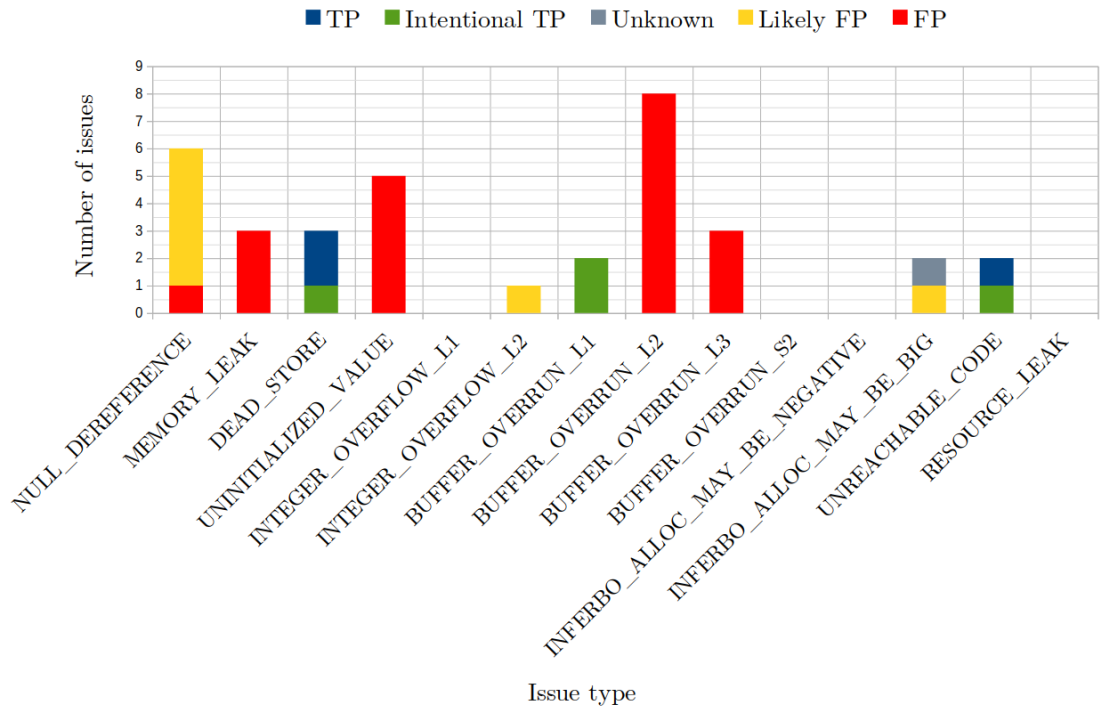

Figure C.15: All the issues reported by Infer on the sed package. Each issue was manually inspected and categorized according to its truthfulness.

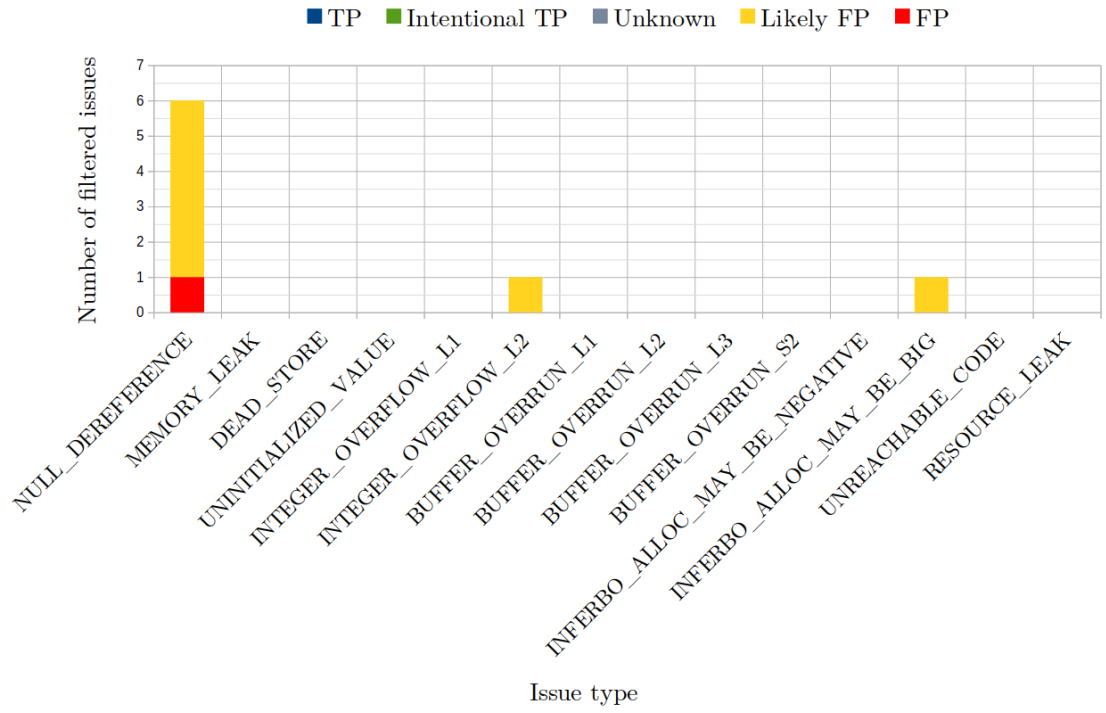

Figure C.16: Issues filtered by heuristics for filtering on the results of the sed package. Each filtered issue is categorized by its truthfulness.

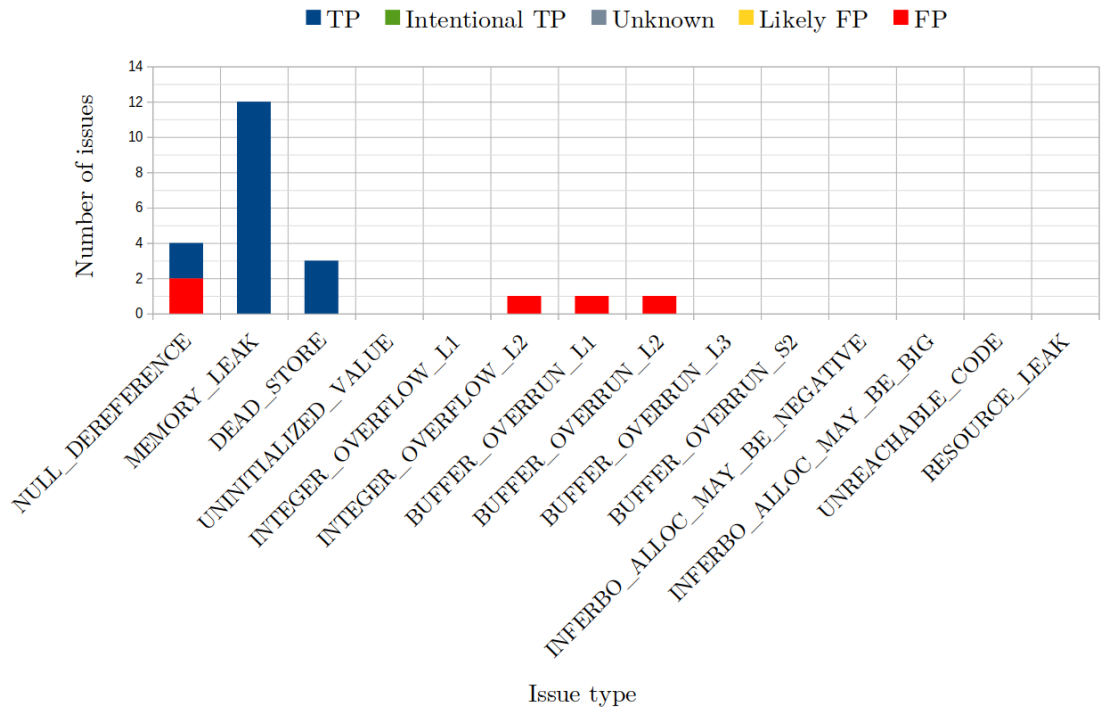

Figure C.17: All the issues reported by Infer on the tree package. Each issue was manually inspected and categorized according to its truthfulness.

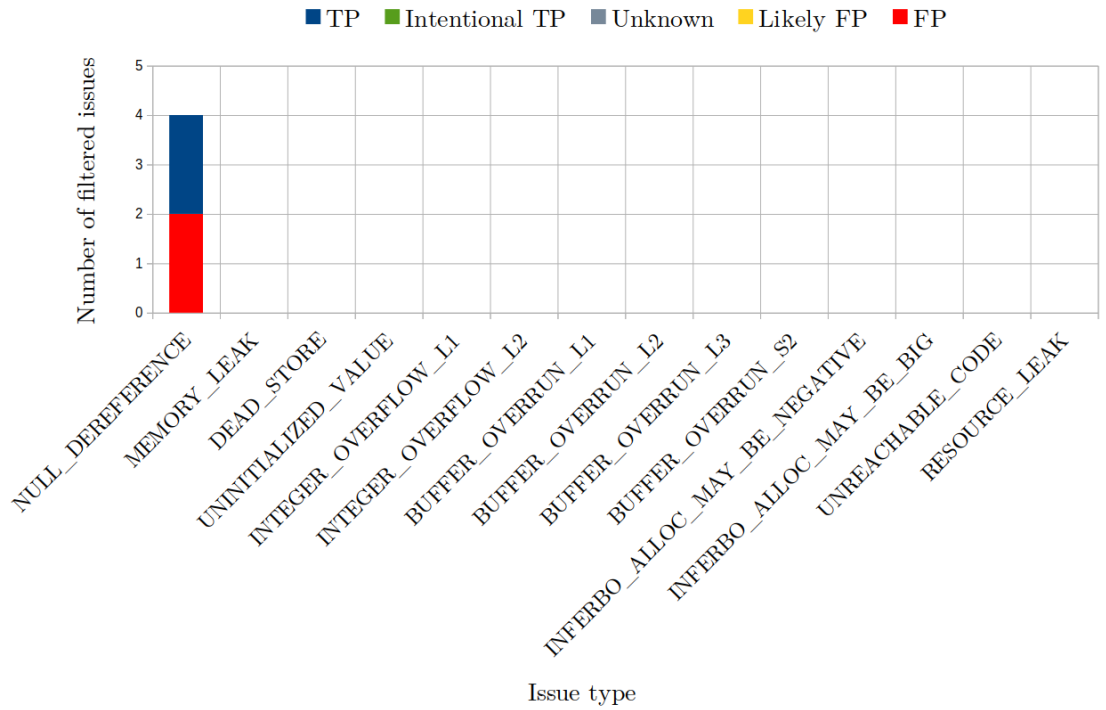

Figure C.18: Issues filtered by heuristics for filtering on the results of the tree package. Each filtered issue is categorized by its truthfulness.

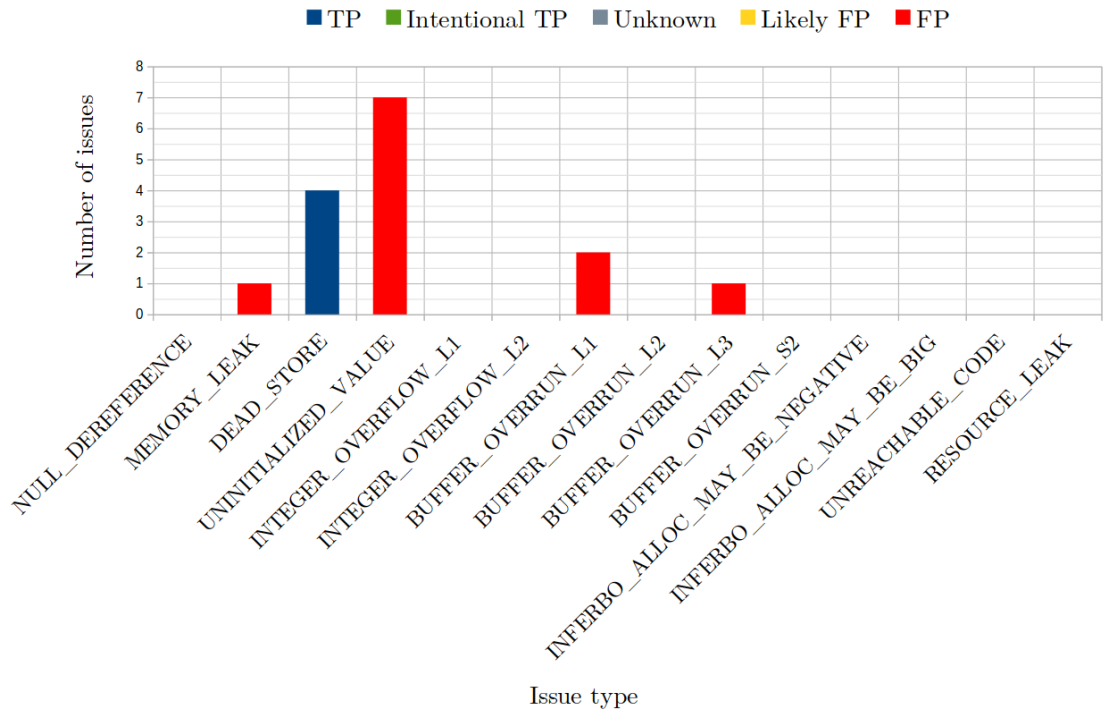

Figure C.19: All the issues reported by Infer on the psmisc package. Each issue was manually inspected and categorized according to its truthfulness.

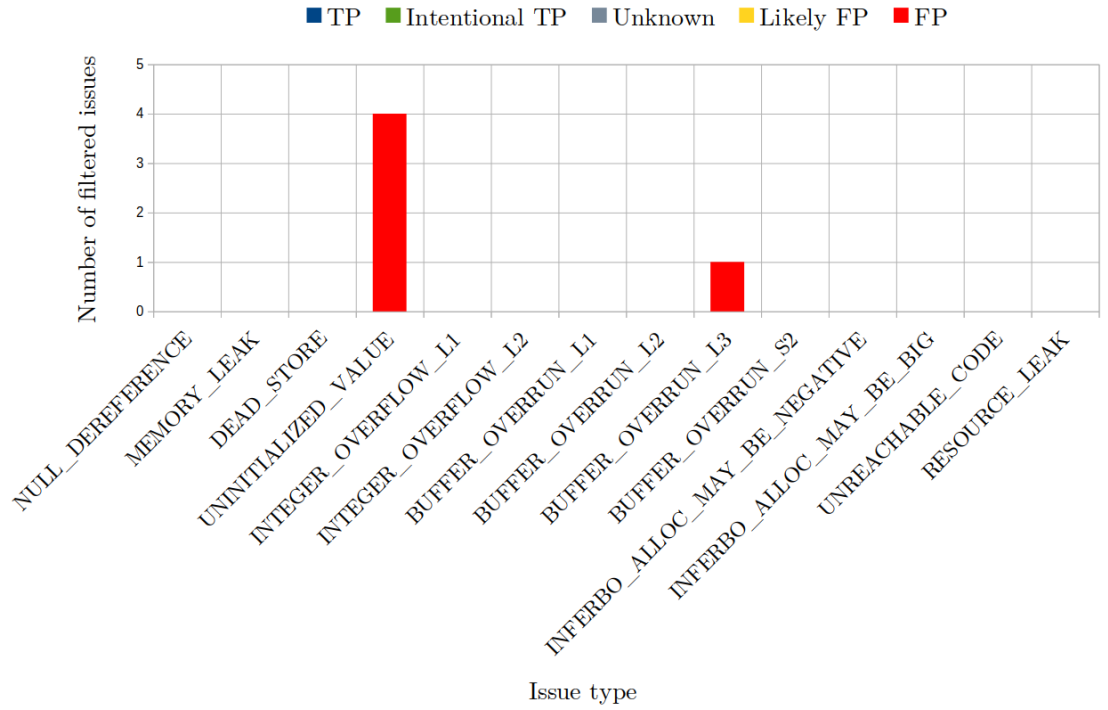

Figure C.20: Issues filtered by heuristics for filtering on the results of the psmisc package. Each filtered issue is categorized by its truthfulness.

Appendix D

User Manual for the Infer Wrapper for Incremental Analysis

The Infer wrapper consists of a single Python script. The script and the experiment is stored in the included storage media, see Appendix A. Alternatively, the script and the experiment can be downloaded from the repository on GitHub (https://github.com/TomasBeranek/infer-wrapper-for-incremental-analysis).

The project must be first captured by the Infer's capture phase in a reactive mode:

```
infer capture --reactive -- BUILD_CMD.
```

The wrapper is called instead of the Infer's binary, when invoking the `infer analyze` command, for example:

```
infer-inc analyze --pulse --bufferoverun
```

To omit the reported issues from files that were not analysed, add the `--incremental-report` option anywhere in the command, for example:

```
infer-inc --incremental-report analyze --pulse --bufferoverun
```

The experiment is described in `README.txt` in the `experiment` directory. The repository is stored in the `incremental-analysis` directory in the included storage media, see Appendix A.