BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

STATIC ANALYSIS USING FACEBOOK INFER FOCUSED ON DEADLOCK DETECTION

STATICKÁ ANALÝZA V NÁSTROJI FACEBOOK INFER ZAMĚŘENÁ NA DETEKCI UVÁZNUTÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **VLADIMÍR MARCIN**
AUTOR PRÁCE

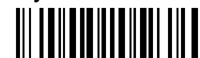
SUPERVISOR **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce



Student: **Marcin Vladimír**
Program: Informační technologie
Název: **Statická analýza v nástroji Facebook Infer zaměřená na detekci uváznutí**
Static Analysis Using Facebook Infer Focused on Deadlock Detection
Kategorie: Analýza a testování softwaru
Zadání:

1. Prostudujte principy statické analýzy založené na abstraktní interpretaci. Zvláštní pozornost věnujte přístupům zaměřeným na odhalování problémů v synchronizaci paralelních procesů.
2. Seznamte se s nástrojem Facebook Infer, jeho podporou pro abstraktní interpretaci a s existujícími analyzátory vytvořenými v prostředí Faceboook Infer.
3. V prostředí Facebook Infer navrhněte a naimplementujte analyzátor zaměřený na odhalování chyb typu uváznutí.
4. Experimentálně ověřte funkčnost vytvořeného analyzátoru na vhodně zvolených netriviálních programech.
5. Shrňte dosažené výsledky a diskutujte možnosti jejich dalšího rozvoje v budoucnu.

Literatura:

- Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, Springer-Verlag, 2005.
- Blackshear, S., O'Hearn, P.: Open-Sourcing RacerD: Fast Static Race Detection at Scale, 2017. Dostupné on-line: https://code.fb.com/android/open-sourcing-racerd-fast-static-race-detection-at-scale/.
- Atkey, R., Sannella, D.: ThreadSafe: Static Analysis for Java Concurrency, Electronic Communications of the EASST, 72, 2015.
- Bielik, P., Raychev, V., Vechev, M.T.: Scalable Race Detection for Android Applications, In: Proc. of OOPSLA'15, ACM, 2015.
- Engler, D.R., Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks, In: Proc. of SOSP'03, ACM, 2003.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a alespoň začátek návrhu z bodu 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Vojnar Tomáš, prof. Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstract

Static analysis has nowadays become one of the most popular ways of catching bugs early in the modern software. However, a frequent problem of static analysers, which are reasonably precise, is their scalability. Moreover, these which are efficient and scale (e.g.: Coverity, KlockWork, etc.) are often proprietary and difficult to openly evaluate or extend. An improvement to this state of practice is brought Facebook Infer, which offers an open-source framework for compositional and incremental static analysis. In this thesis, we present our Low-Level Deadlock Detector (L2D2) extending the capabilities of Infer. Our algorithm fits the compositional analysis, based on a context independent computation of a summary for each function, which results in its high scalability. We have implemented the algorithm and evaluated it on a benchmark consisting of real-life programs derived from the Debian GNU/Linux with in total 11.4 MLOC. While neither sound nor complete, our approach is effective in practice, finding all known deadlocks and giving false alarms in less than 4 % of the considered programs only.

Abstrakt

Statická analýza dnes patrí medzi najpopulárnejšie metódy na odhaľovanie chýb v modernom softvéri, no častým problémom dostatočne presných statických analyzátorov je ich škálovateľnosť. Mnohé efektívne analyzátory (napr.: Coverity, KlockWork, atď.) sú navyše proprietárne, čím sa ich ďalšia rozšíriteľnosť a použitie stávajú obťažnými. Pokrok v tejto oblasti prináša Facebook Infer, ktorý ponúka open-source framework na tvorbu kompozičných a inkrementálnych statických analýz. V tejto práci predstavujeme vlastný Low-Level Deadlock Detector (L2D2), ktorý rozširuje funkcionalitu Inferu. Náš algoritmus spĺňa princípy kompozičnej analýzy, založenej na kontextovo nezávislom výpočte súhrnu pre každú funkciu, čo má za následok jeho vysokú škálovateľnosť. Algoritmus sme implementovali a overili na sade príkladov z Debian GNU/Linux, ktorá pozostávala z 11.4 MLOC. Aj keď náš prístup nie je ani presný ani úplný, ukazuje sa ako efektívny. Okrem toho, že dokázal odhaliť všetky známe uviaznutia, hlásil falošné pozitíva v menej ako 4 % z testovaných programov.

Keywords

static analysis, abstract interpretation, Facebook Infer, deadlock detection, Infer.AI framework, L2D2

Kľúčové slová

statická analýza, abstraktná interpretácia, Facebook Infer, detekcia uviaznutia, Infer.AI framework, L2D2

Reference

MARCIN, Vladimír. *Static Analysis Using Facebook Infer Focused on Deadlock Detection.* Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

Rozšírený abstrakt

V dnešnej dobe sa čoraz väčšmi kladie dôraz na kvalitu softvéru, a to aj v prípade aplikácií, ktorých zlyhanie nepredstavuje kritické riziká. S nárastom komplexnosti počítačových programov sa manuálne odhaľovanie ich chýb stáva náročným až nemožným. Jedným z existujúcich prístupov na automatickú detekciu chýb je automatické testovanie a dynamická analýza. Nevýhoda týchto metód spočíva predovšetkým v tom, že kvôli nedostatočnému pokrytiu kódu testovaného programu nemusia odhaliť všetky prítomné chyby. Alternatívnym riešením je statická analýza, schopná objaviť aj chyby na zriedkavých cestách programom, ktoré nemusia byť skontrolované v priebehu testovania. Práve táto skutočnosť súvisí s veľkou nevýhodu statických analyzátorov, a síce so vznikom falošných chýb v dôsledku uvažovania neexistujúcich ciest v programe. Jedným z ďalších problémov statických analýz je ich škálovateľnosť, ktorá sa javí ako nevyhnutnosť pri testovaní veľkých a rýchlo sa meniacich projektov.

Spoločnosť Facebook nedávno ponúkla vlastné riešenie na efektívne odhaľovanie chýb a verifikáciu programov v podobe nástroja Infer — vysoko škálovateľný, kompozičný a inkrementálny framework na vytváranie interprocedurálnych, statických analyzátorov. Napriek tomu, že sa Infer stále vyvíja, je každodenne využívaný spoločnosťami ako Facebook, WhatsApp, Uber či Amazon, kde pomocou neho odhaľujú rôzne druhy chýb vrátane únikov pamäte (memory leaks), chýb súbehu (data races) a ďalších.

Keďže paralelné programy sa stávajú prevládajúcimi v dôsledku reality viacjadrových procesorov a vytváranie kvalitných paralelných programov sa stalo kriticky dôležitým. Táto práca sa preto zameriava na chyby vznikajúce práve v týchto programoch. Konkrétne sem sa rozhodli rozšíriť nástroj Infer o analýzu slúžiacu na detekciu uviaznutí (deadlocks), ktoré predstavujú pravdepodobne jednu z najčastejšie sa vyskytujúcich chýb v paralelných programoch.

V súčasnosti je známych viacero typov statických analyzátorov, ktoré sa sústreďujú na odhaľovanie uviaznutí, no žiaden z nich (aspoň podľa našich vedomostí), okrem analyzátora Starvation (implementovaného v samotnom Inferi), nespĺňa princípy kompozičnej analýzy používanej vo Facebooku. Problémom Starvation analyzátora je fakt, že bol vyvinutý primárne pre jazyky Java a C++, tým pádom nepodporuje nízkoúrovňové zamykanie, ktoré ponúka napríklad knižnica `Pthreads`.

Aby sme túto situáciu vylepšili, predstavujeme náš Low-Level Deadlock Detector (L2D2), založený na novej metóde odhaľovania uviaznutí, ktorá zvláda aj nízkoúrovňové zamykanie. Náš prístup používa kompozičnú analýzu (presadzovanú v nástroji Infer) založenú na počítaní súhrnov (summaries) pre každú funkciu v analyzovanom programe, a to nezávisle od jej kontextu. Súhrn môže byť chápaný ako štruktúra obsahujúca relevantné dáta potrebné na odhalenie detekovaného problému, čo v praxi znamená, že každá analýza používa vlastný typ súhrnu na odhalenie konkrétneho problému. Práve tento prístup robí naše riešenie vysoko škálovateľným a tiež umožňuje inkrementálnu analýzu, vďaka ktorej je L2D2 použiteľný pri každodennej práci programátora.

Nami navrhnutú metódu sme úspešne implementovali v nástroji Infer a experimentálne overili na sade príkladov z Debian GNU/Linux pozostávajúcej z 11.4 MLOC. Táto sada bola použitá aj na experimentálne overenie nástroja Cprover, s'ktorým sme sa porovnávali. Vo výsledku náš analyzátor odhalil všetky známe uviaznutia (rovnako ako Cprover) s menej ako 4 % falošných pozitív, čo je o 7.5 % menej ako v prípade nástroja Cprover. Taktiež sme úspešne preukázali jeho škálovateľnosť, keďže na kontrolu testovacej sady potreboval menej ako 1 % času v porovnaní s Cproverom.

Static Analysis Using Facebook Infer Focused on Deadlock Detection

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of prof. Ing. Tomáš Vojnar, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

....
Vladimír Marcin
May 15, 2019

Acknowledgements

I would like to thank my supervisor Tomáš Vojnar for his support both during designing the L2D2 analyser and in the critical time of writing this thesis. Further, I would like to thank Nikos Gorogiannis and Sam Blackshear from Infer team at Facebook for helpful discussions about the development of our analyser. Lastly, I thank for the support received from the H2020 ECSEL project Aquas.

Contents

1 Introduction **2**

2 Preliminaries **5**
2.1 Static Analysis 5
2.2 Facebook Infer 7
2.3 Existing Solutions of Deadlock Detection 11

3 A Modular Low-Level Deadlock Detector **13**
3.1 Design Principles 13
3.2 Deadlock Detection 14
3.2.1 An Illustrating Example 14
3.2.2 Algorithm Details 17
3.3 Reporting Possible Deadlocks 20
3.4 Reducing the Number of False Alarms 21

4 Implementation and Experiments **23**
4.1 Analyser Plugin 23
4.2 Usage 26
4.3 Experimental Evaluation 26
4.3.1 Basic Examples 26
4.3.2 Cprover Test-Suite 27

5 Conclusion **28**

Bibliography **29**

A Storage Medium **32**

Chapter 1

Introduction

"Testing is about defect detection, **Quality Assurance** is about defect prevention."

Amir Ghahrai

In the spirit of Amir Ghahrai's claim, nowadays, the demand for quality assurance is growing even for non-safety-critical applications. Large companies like Mozilla, Facebook, Spotify, and others are increasingly focusing on the verification area, trying to reduce the number of bugs in their products before they are deployed, ideally, already during development.

With the increasing complexity of computer programs, manual error detection becomes impossible and that is the reason why many studies have been made to detect errors automatically. Existing approaches to automatic errors detection can be divided into two main categories, namely, automated testing & dynamic analysis and static analysis (which is the focus of this thesis). Ideally, both of these approaches are used since they both have various advantages and disadvantages.

The first of the mentioned approaches, automated testing & dynamic analysis, has the task to detect real or potential problems by executing an analysed program. The advantage of this approach is that no source code is needed for testing, although testing with the source code available is usually better. However, if we want to test the program during its development, this approach becomes sometimes more complicated: one has to write unit tests (or use some advanced tool for their at least partially automated generation). If larger pieces of code are missing, this approach may become infeasible (at least for some kinds of tests). Another disadvantage is the possibility of not revealing some issues due to insufficient test coverage.

On the other hand, static analysis (at least in some of its forms) can be faster and does not require complete runnable code. Such analyses are easier to use during development. Moreover, static analysis is capable of discovering bugs even in rare execution traces that need not be spot during testing — though, on the other hand, this can lead to generation of false alarms due to considering non-existent paths in the program. Furthermore, the use of the tools such as Coverity and FindBugs in the industry provides evidence that static analysers meet the scalability requirement, which is a necessity to handle large and fast-changing codebases [1].

One way how to achieve scalability is compositional and incremental static analysis as supported, e.g., in the Infer[1] tool developed by Facebook. This tool is still under further development, and it is in everyday use in Facebook (and several other companies, such as Uber, WhatsApp, Amazon, and others) and it already provides many checkers for various kinds of bugs. It is Facebook Infer that is the subject of the thesis that aims at contributing some new checker into the suit of the already available ones.

Since concurrent programs are becoming prevalent due to the reality of multi-core hardware and since writing good quality concurrent programs has become critically important, the focus of this thesis is in particular on finding problems in these programs. Another motivation is that detecting errors in such programs is very difficult because of the non-deterministic behaviour of concurrently running threads due to which concurrent errors need not show up even when repeated tests are run.

To create a non-trivial multi-threaded program, programmers must use some sort of synchronisation between threads. However, most of them think sequentially and easily make mistakes when designing the needed synchronisation. Incorrect synchronisation then results in various kinds of errors, including, e.g., data races and deadlocks as two of the probably most prominent concurrency related errors. In this thesis, we will deal with the second mentioned kind of errors, that is, deadlocks.

A *deadlock* is a situation where each thread of some set of threads waits for a resource that is owned by some thread from the same set which will not release it before getting the resource that it is itself asking for. Figure 1.1 shows a deadlock between two threads and its *lock graph*, which is defined by the order in which threads access the locks. A cycle in this graph denotes a deadlock.

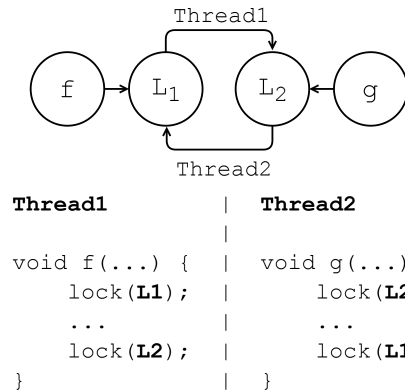


Figure 1.1: A deadlock and its lock graph.

There exist various static analyses for deadlocks, which are discussed later on, but none (to the best of our knowledge) fitting the computation loop of Infer, except the Starvation checker implemented in Infer itself. A problem of the Starvation checker is that it is developed primarily for Java/C++ locks and does not handle low-level locks as used, e.g., in C/Pthreads.

To improve on this situation, in this thesis, we present our Low-Level Deadlock Detector (L2D2) which is based on a new method for deadlock detection that handles low-level locking and that fits the *compositional analysis* style common to Facebook Infer. This approach makes our solution highly scalable and also allows for incremental analysis which makes it useful in everyday development. It also requires no additional help from the programmer to detect bugs (no annotations). Our method was only weakly inspired by the general principles and some heuristics of some other analysers (especially [8]).

We have already successfully implemented our method in Infer and experimentally show the effectiveness on the benchmark consisting of 11.4 MLOC derived from Debian GNU/Linux distribution, which was used to experimentally evaluate the Cprover tool presented in [16].

[1] https://fbinfer.com/

Some parts of this thesis were taken from the article [10], published at the Excel@FIT'19 conference, presenting the preliminary results of this work. Our work was there awarded a *Prize of Jíři Kunovský*[2] and also an award of an expert committee composed of FIT[3] academics.

Structure of this paper. The rest of this thesis is structured as follows. **Chapter 2** introduces the reader to the theory of abstract interpretation and provides an overview of the Infer tool. Also, it describes some of the existing solutions which use various approaches to deadlock detection. After that, a complete design of the L2D2 analyser is presented (**Chapter 3**). Further, **Chapter 4** summarises the current state of implementation, followed by its experimental evaluation. Finally, **Chapter 5** addresses directions for future research and concludes this thesis.

[2]The Jíři Kunovský Prize is awarded to five works that receive the most votes from the professional public during the conference.

[3]**Faculty of Information Technology** of Brno University of Technology

Chapter 2

Preliminaries

2.1 Static Analysis

Static analysis is one of formal verification methods (or, at least, it can be — if its soundness is not sacrificed to scalability). Formal verification of a program consists of verifying whether the semantics of the program ("what the program actually does") meets the pre-determined specification ("what the program should do"). Except for static analysis, formal verification can also be performed by methods like theorem proving or model checking.

In short, static analysis collects information about the program behaviour based on its source code without executing the program at all, or at least without executing it under its original semantics as in dynamic analysis or (basic) model checking. Obtained information can be used to find potential errors in the code but also, e.g., for optimisation, code generation, etc.

The area of static analysis is very extensive and includes many different approaches such as data flow analysis, constraint-based analysis, type-based analysis, and abstract interpretation, which is the approach used in this work.

Abstract Interpretation

The description of Abstract Interpretation is mostly inspired by [4, 7] and web article called "Abstract Interpretation in a Nutshell"[1]. All of these sources were written by the author of the original paper [6].

The concrete semantics of a program gives the set of all possible executions of the program in all possible execution environments. When a semantic analysis of programs is to be automated, in the general case, the answers can only be partial or approximate since concrete, precise information is in general not computable within finite time and memory (see Rice's theorem [21] and the halting problem [22]).

Abstract Interpretation [6] is a method for approximating the semantics of programs which can be used to gain information about them in order to provide sound answers to questions about their run-time behaviours. From a practical point of view, the purpose of abstract interpretation is to design automatic program analysis tools for statically determining dynamic properties of programs. To achieve this, abstract interpretation defines an *abstract semantics* of a program that is a superset of the *concrete semantics* of the program. If an execution of a program is represented by a curve showing an evaluation of the vector *f(t)* of the values of the input, state, and output variables of the program as

[1] https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html

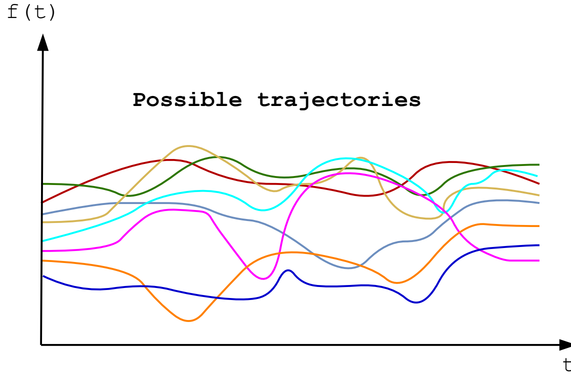


Figure 2.1: Concrete semantics.

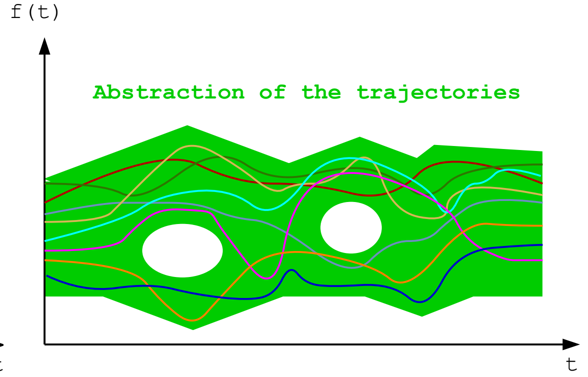


Figure 2.2: Abstract semantics.

a function of time t, the concrete semantics can be represented as in Figure 2.1. An example of a corresponding abstract semantics is shown in Figure 2.2. One can see that the abstract semantics covers all possible executions, therefore if the abstract semantics meets the pre-determined specification, then the concrete semantics meets it too. However, the consequence of the overapproximation of the possible executions is that inexisting executions are considered, some of which may be erroneous, which may lead to false alarms (also called *false positives*). Another case of the false alarms are the so-called *false negatives*. These errors arise as a result of insufficient coverage of a concrete semantics (abstract semantics do not covers all possible cases of the concrete semantics).

Concrete and abstract semantics are defined on suitable concrete and abstract lattice-based domains. To establish the correspondence between these domains, abstract interpretation uses a *Galois connection*[2] $(D, \sqsubseteq) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} (D^*, \sqsubseteq^*)$ which links a concrete domain $(D, \sqsubseteq)$ with an abstract domain $(D^*, \sqsubseteq^*)$ by a pair of monotone functions — the so-called *abstraction* and *concretisation*, denoted α and γ, respectively. Under such a connection, a concrete program property $p \in D$ is approximated by any abstract program property $p^* \in D^*$ such that $p \sqsubseteq \gamma(p^*)$ and has a best/most precise abstraction $\alpha(p) \in D^*$.

Example 1 ([5]): Answering a concrete question in the abstract

The concrete question, "Is there a partial trace in X which has s, s' and s'' as initial, intermediate and final states?" can be replaced by the abstract question "Is there a pair $\langle s, s'' \rangle$ in $\alpha(X)$?". If there is no such pair in $\alpha(X)$, then there is no such partial trace in $\gamma(\alpha(X))$ whence none in X since $X \subseteq \gamma(\alpha(X))$. However, if there is such a pair in $\alpha(X)$, then we cannot conclude that there is such a trace in X since this trace might be in $\gamma(\alpha(X))$ but not in X.

Each program's statement has assigned a corresponding concrete and abstract transformer that represents the effect of the statement on a concrete and abstract context. These transformers are modelled as monotone functions: $f_a : D^* \to D^*$ for the abstract transformer and $f_c : D \to D$ for the concrete transformer. So, if we consider as a concrete domain a set of integers and as an abstract domain a set of even and odd numbers, for

[2] A monotone Galois connection between two partially ordered sets $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ consists of two monotone functions: $f : A \to B$ and $g : B \to A$ such that $\forall a \in A \wedge \forall b \in B : f(a) \sqsubseteq_B b \iff a \sqsubseteq_A g(b)$.

the `x++` statement, the transformers may look like:

$$f_c(x) = x + 1$$

$$f_a(x) = \begin{cases} \textit{even} & \texttt{if } x \texttt{ is } \textit{odd} \\ \textit{odd} & \texttt{if } x \texttt{ is } \textit{even} \end{cases}$$

In a more general formulation of abstract interpretation [7], the requirement of dealing with a Galois connection is lifted, and the analysis is defined in terms of a concretisation (or, dually, abstraction) function only, which, however, excludes the possibility of defining best abstract transformers. A consequence of using the more general setting is that there is no easy way of comparing the precision of abstractions.

The program analysis is then performed by iterating the abstract transformers over the *control flow graph*[3] (CFG). It is possible that there are multiple paths leading to a single program point, therefore abstract interpretation defines an operator for *accumulation* of abstract values computed for that point via all program paths. This operator is called *join*, and usually denoted $\circ$. Moreover, when the abstract domain is large or infinite, *widening* $\triangledown$ and *narrowing* $\triangle$ operators should be used to tune the cost/precision compromise. The main property of the widening operator is that for any infinite sequence of abstract values $x_0, x_1, x_2, \dots$, the sequence $y_0, y_1, y_2, \dots$ where $y_0 = x_0$ and $y_{i+1} = y_i \triangledown x_{i+1}$ eventually stabilises. The widening operator is an overapproximation of the join operator and is used at loop junctions where it is generally not guaranteed that, without widening, the analysis will terminate in acceptable or even finite time. As an example, imagine a cycle in which each repetition increases the array's index by one. It will lead to states with this variable having values of $\{0\}, \{0, 1\}, \{0, 1, 2\}$ etc. until the range of the type of variable is reached. So, a widening operator can define that if a *fixed point*[4] is not reached after, e.g., ten iterations through the cycle, the value of our variable will be set to some value that guarantees termination, e.g., ∞ (generally *top value*[5] of the given abstract domain). In summary, the widening operation decreases the precision of the computation to make it faster. Sometimes, a narrowing operator may be used after the widening operation to refine its efect (the narrowing operator may be sometimes missing).

2.2 Facebook Infer

Infer is an open-source static analysis tool, written in OCaml, which is used for analysing Objective C, Java and C/C++ code bases at Facebook on daily basis. Initially, Infer was based on an academic work about Separation Logic [19], [20], which attempted to make algorithms for reasoning about memory safety of programs with pointers scalable. Since then it has evolved into an analysis framework supporting a variety of sub-analysis, including ones for data races [2] (RacerD), for buffer overruns (Inferbo), and for other specialised properties.

Infer is notable mainly for its ability to perform in-depth interprocedural analysis, and yet still scales to large code bases. Other popular open-source tools such as FindBugs and Clang Static Analyser are also scalable, but in comparison to Infer, their reasoning

[3] A control flow graph is a graph representation of all paths that might be traversed through a program during its execution.

[4] A fixed point is reached when all values in the given state before cycle execution and afterwards are the same.

[5] The top value of an abstract domain is the most imprecise value, which covers all possible values.

is typically limited to a single function/file. There are also many research tools, which offer interprocedural analysis but require a sophisticated whole-program analysis, which may be precise, but sadly cannot scale to millions of lines of code.

Infer scales by using a technique called *compositional program analysis*, where the result of the whole program analysis is computed from the analysis results of the individual parts of the program — the so-called *summaries*. A compositional analyser computes a summary of each function independently from its context and then uses the summary in all its call sites. A consequence of that is that each function is analysed once only, making the analysis scale. Another consequence of compositionality is that Infer can work incrementally, which means that after a change in the program, it analyses only the parts affected by the change instead of the entire program. These properties are an advantage especially for large code bases, where a complete re-analysis of the project after each change would be infeasible. This is especially true when the analysis is to be used within live development when programmers are mostly willing to correct the bugs [3].

The Infer.AI Framework

Infer.AI[6] (Abstract Interpretation) is a framework which provides an API to Infer's backend compositional analysis infrastructure, based on abstract interpretation. It makes the development process of new analysers much easier, because the developer can focus mostly on the design of the new analysis instead of dealing with the backend of abstract interpretation.

A block diagram of a simplified architecture of the framework can be seen in Figure 2.3. The architecture consists of three main components, which are the frontend, scheduler and abstract interpreter.

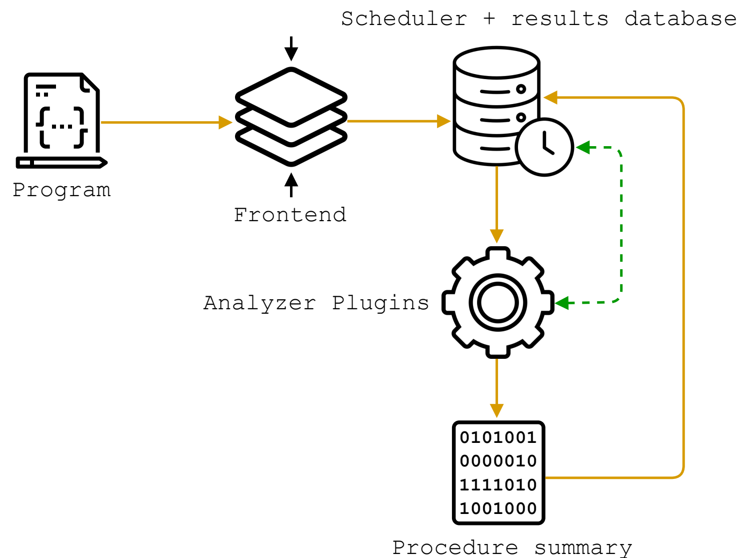


Figure 2.3: The architecture of the Infer.AI framework.

Frontend. The task of the frontend component is to compile the analysed program from its source language to the Smallfoot Intermediate Language (SIL) which is the intermediate

[6] https://fbinfer.com/docs/absint-framework.html

language used by Infer during the analysis. The frontend provides an output in the form of a CFG for each function in the analysed program, where each of its nodes consists of a list of SIL instructions. In SIL there are four main instructions:

- `LOAD` – loads the value from the heap into a temporary identifier. The heap is represented by an address expression, which can denote the address of a simple program variable, array or structure. In the case of a structural type, the `LOAD` instruction also provides an offset within the structure.
- `STORE` – stores the value of an expression into the heap. The heap representation is the same as in the case of `LOAD`, and the value is represented by an expression consisting of constants and temporary identifiers created by the `LOAD` instruction.
- `CALL` – represents a function call and provides information about the return value, return type, list of function's parameters (also their values and types), and some call flags like, e.g., `is_virtual, is_interface, ...`
- `PRUNE` – splits the CFG into two new branches based on a boolean expression. It also provides information about the source of the pruning (a conditional, a ternary operator, a cycle, etc.).

Infer also provides an abstraction over SIL, which is called as the High-level Intermediate Language (HIL). It consists only from three instructions (`CALL`, `ASSIGN` — abstraction of `STORE`, and `ASSUME` — abstraction of `PRUNE`), but still is sufficient for many analyses, including the one presented in this thesis.

Analyzer Plugin. Another component of the Infer.AI framework is the abstract interpreter which has to be instantiated by every analyser implemented in Infer. This is the way a new analyser plugin is created. The plugin then takes as its input the CFG of the analysed function and the *transfer functions* module (of a specific analyser), which defines abstract transformers for SIL instructions. The effect of these transfer functions is then applied to an *abstract state* of the analyser, which is tied to its *abstract domain*.

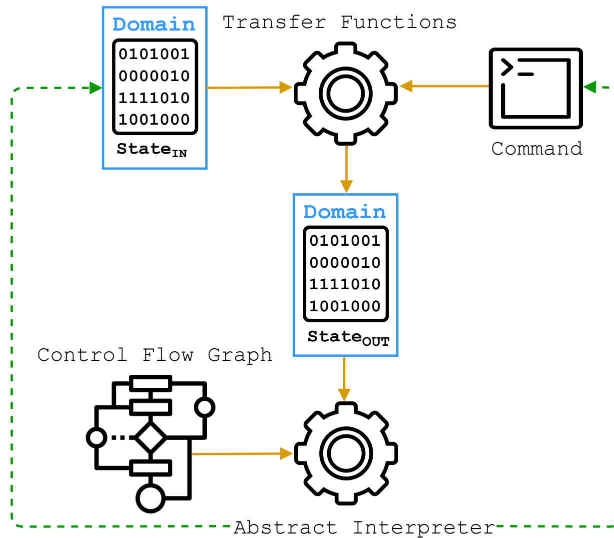


Figure 2.4: Intraprocedural analysis flow.

A basic building block of the INFER's analysis approach is an *intraprocedural* analysis that is illustrated in Figure 2.4. The main component is the already mentioned abstract interpreter, which manages the entire analysis flow. At the beginning of the analysis, it selects the first instruction from the CFG and applies it to an initial abstract state using the appropriate transformer. Next, it takes the output state of the transformer and applies the next instruction to it. In case of conditionals and loops, the role of the abstract interpreter is to call the join and widen operation, respectively. This repeats until all instructions from the given CFG are processed, and a fixpoint is reached. The result is a summary of the analysed function.

The intraprocedural analysis is lifted to an *interprocedural* one by adding a new abstract transformer into the transfer functions module that is responsible for handling function calls under the given analysis. Then, if a call of a user-defined function appears in the analysed code, the abstract interpreter uses the given transformer and instantiates the summary of this function at actual callsite.

Summary. INFER represents its summaries as specifications in a program logic. In more detail, a specification is a pair of a *precondition* and a *postcondition* known from the Hoare logic [13] which uses Hoare triples of the form $\{pre\}$ $code$ $\{post\}$ where pre is a precondition, $post$ is a postcondition, and $code$ is a program part (one function in case of INFER). The interpretation of a $(pre, post)$ pair is that if the logical property pre is fulfilled before executing a function, $post$ will be fulfilled afterwards. As an example, we could imagine a function which closes a resource r given to it as a parameter:

$$\{r \mapsto open\} \quad close_resource(r) \quad \{r \mapsto closed\}.$$

Scheduler and results database. Another main component of the INFER.AI architecture is the scheduler. Its role is to determine the order in which the functions are analysed. Once the analysis of a function is ended, INFER stores a summary of the function to the results database such that it can be used repeatedly at different call sites. The analysis of individual functions cannot be arbitrary if an interprocedural analysis is considered. The reason is that during the interprocedural analysis the abstract interpreter needs to have a valid summary for each function that is called by the currently analysed one. To handle this issue INFER.AI uses a *call graph*[7] to ensure that functions will be analysed in a suitable order.

An example of such a call graph can be seen in Figure 2.5. Using this figure, we can illustrate the order of analysis in INFER and its incrementality. The underlying analyser starts with the sink nodes[8] $\mathtt{F}_5$ and $\mathtt{F}_6$ and then proceeds towards the root $\mathtt{F}_{\mathtt{MAIN}}$ while respecting the dependencies represented by the edges. Each subsequent code change then triggers a re-analysis of the directly affected functions only as well as a re-analysis of all the functions up the call chain. For example, if function $\mathtt{F}_6$ were changed, INFER would have to re-analyse functions $\mathtt{F}_3$, $\mathtt{F}_1$, and $\mathtt{F}_{\mathtt{MAIN}}$ only. However, a change in function $\mathtt{F}_2$ affects only $\mathtt{F}_{\mathtt{MAIN}}$, and the summaries of all other functions stay untouched, which brings scalability to rapidly changing codebases. This approach also brings one more advantage to the analysis. Thanks to it, independent functions can be analysed simultaneously, which is another reason why INFER scales so well.

[7] A call graph is a CFG, which represents calling relationships between functions in a program.

[8] A sink node in a graph is a node such that no edge emerges out of it.

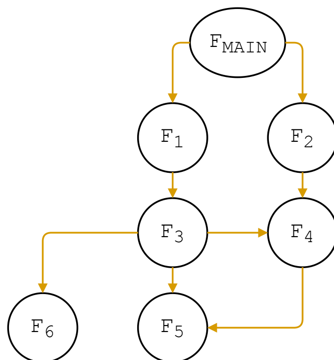


Figure 2.5: A call graph.

2.3 Existing Solutions of Deadlock Detection

Nowadays, there are many tools used for deadlock detection in multi-threaded programs. A common deficiency of a large number of them is that they are unsound and/or incomplete (produce many false positives/negatives) or they are precise but their requirements on time and resources are unacceptable in case of large codebases.

This section lists some of the most popular existing analysers using different approaches to deadlock detection.

Dynamic Tools. There are many analysers that detect deadlocks dynamically. An advantage of such tools is that they see concrete executions which allows them to reduce the number of false alarms. On the other hand, they may fail to detect specific errors due to insufficient test coverage. Moreover, while performing a single test run of a given system may seem quite scalable, the scalability of such tools is reduced by (1) a need to repeat the runs many times to mitigate the non-determinism and/or (2) the slow-down generated by monitoring code or (3) techniques such as noise injection [9] used to increase the achieved coverage of the possible thread interleavings.

Dynamic approaches such as Visual Thread [11] and GoodLock [12] use deadlock prediction to detect a deadlock. These tools make predictions about an exponential number of permutations of a single execution history. Essentially, both tools monitor the lock acquisition history by creating a dynamic lock-order graph, followed by checking the graph for existence of deadlock candidates by searching for a cycle in it. A handicap of these approaches is that they may produce a high rate of false positives.

Another group of dynamic analysers are tools like DeadlockFuzzer [15] that verify the candidates produced by a deadlock predictor by re-executing the program. They efficiently try out different schedules to see if they can recreate the deadlock, but the additional runs come with no guarantee of success.

Like any dynamic analysers, these approaches cannot be applied to open programs and without test input data. This makes it impossible to use them during development unless suitable test suites and models of the missing parts of the code are created. Moreover, creating test suites that would suitably exercise various thread interleavings is not easy (nor common).

Static Tools. One group of static analysers are those that use annotations to detect deadlocks. This group includes analysers such as Warlock [23] or ESC [17]. The main disadvantage of these analysers is that the aforementioned annotations must be supplied by programmers, which represents an unnecessary burden and increases the time needed to verify the software.

Another group consists of tools using dataflow analysis. This group includes the RacerX analyser introduced by Engler and Ashcraft in [8]. It performs a flow-sensitive and contex-sensitive interprocedural analysis of C programs to compute a static lock-order graph, by computing so-called *locksets*, i.e., sets of currently held locks, and reports a possible deadlock in case of a cycle in it. This approach scales quite well but produces false alarms due to the used approximations. Moreover, the approach is based on classical forward analysis that differs from the analysis loop of Infer.

Williams et al. [24] present an algorithm for deadlock detection in Java libraries. Their detector uses interprocedural, context and flow sensitive analysis to create a lock-order graph. To build the graph, the algorithm iterates over procedures in a library building a lock-order graph for each of them. Then the graphs of all procedures are combined into a single graph for an entire library. Combining information into one graph allows one to represent any calling pattern of library's methods across any number of threads. However, it considers infeasible paths and impossible alias relationships, resulting in false positives. Moreover, the algorithm is not designed as compatible with the common analysis loop of Facebook Infer.

The Chord static analyser presented in [18] reduces the number of false alarms obtained by a pure data-flow interprocedural analysis using a novel combination of static analyses each of which approximates a necessary condition for a deadlock. The algorithm then reports only deadlocks that fulfil all of the necessary conditions. However, since this approach combines six analyses, the algorithm becomes a bit heavy-weight and does not scale well. Another limitation is that they may fail to report some real deadlocks due to using a may-alias pointer analysis instead of a must-alias analysis (when deciding whether some reentrant lock is locked repeatedly and hence its locking can be ignored). They also report some false positives due to limitations of their thread-escape analysis.

The Cprover tool implements the approach introduced by Kroening et al. in [16], which also uses a combination of multiple analysis to create a sound (i.e., misses no deadlocks) static deadlock analysis for C/Pthreads. They also build a lock-order graph and search for a cycle in it to detect deadlocks. The biggest limitation of Cprover is the pointer analysis used, which takes most of the analysis time (making the analysis less scalable), and it is also a source of false alarms. Further, as with the above approaches, the approach of [16] does not meet the principle of compositional analysis used in Infer.

According to our knowledge there is only one deadlock detector which uses a compositional approach in the spirit of analyses used in Infer, and that is the Starvation[9] checker, already mentioned in Chapter 1. It detects deadlocks by deriving lock dependencies for each function, followed by checking whether some other function uses the locks in an inverse order. The problem of this analyser is that it uses a heuristic based on the class of the lock to determine the functions whose summaries should be checked for an inverse lock dependency. So, if a program uses low-level locks (e.g. C/Pthreads), which do not provide this information, the analyser will not detect any deadlocks on it.

[9] https://fbinfer.com/docs/checkers-bug-types.html

Chapter 3

A Modular Low-Level Deadlock Detector

This chapter presents the design of our L2D2 analyser. First, we sketch the basic ideas of our analysis. As the next step, we introduce the algorithm of our analyser in more detail and demonstrate its use on a simple example. After that, we describe an algorithm for reporting deadlocks, and at the end of this chapter, we will talk about some techniques which we use to reduce false positives.

3.1 Design Principles

The design of L2D2 is based on the following principles:

(1) Interprocedural analysis: to be able to find deadlocks between multiple functions.

(2) Compositional analysis: each function is analysed independently from its context.

(3) Process locks sequentially while reasoning conservatively about all possible interleavings between threads.

(4) Represent locks by access paths: the analyser should not perform a more detailed alias analysis.

The first two decisions are motivated by the need to run L2D2 on a real-world, fast-changing, large code basis and still meet the scalability requirement. Since most known deadlocks occurring in real-world code are interprocedural, we have also decided to design an analysis which can handle this kind of bugs. As already mentioned, one way to achieve the desired scalability of interprocedural analysis is compositionality [3]. This is the approach that we also strive to use.

The other two decisions came about as a compromise between scalability and precision. The third decision is motivated by the large number of possible interleavings between threads. For example, the number of interleavings between two threads where $thread_1$ has N instructions and $thread_2$ has M instructions is the binomial coefficient $\binom{N+M}{N}$. As the number of instructions increases, the number of interleavings will increase rapidly, making it impossible to explore all interleavings by brute force. The decision to reason about any thread interleavings is based on programming intuition. When writing multi-threaded programs, programmers usually do not think about different possible interleavings in too many details and they use synchronisation conservatively. Hence, if the analysis assumes

that any interleaving is possible, this will often not be a too drastic overapproximation (at least in common programs). However, this over-approximation may of course lead to false positives because our analyser may consider cases that may not occur.

The last decision is motivated by the fact that, according to our knowledge, there is no sufficiently precise alias analysis that works compositionally and at scale. The principle of using the access paths has been taken from an existing analyser for data race detection (RacerD [2]) which is already implemented in Infer and it uses this principle to report races between syntactically identical *access paths* [14]. The access paths represent heap locations via the paths used to access them (see Figure 3.1).

`variables`	`x` $\in$ `Var`
`field names`	`f` $\in$ `Field`
`access paths`	$\pi \in$ `Path ::= Var x Field`* `|` π `:= x.f`$_1$`....f`$_n$

Figure 3.1: An access path is represented by a base variable `Var` followed by a sequence of fields `Field`.

According to the authors of [2], using the syntactic equality of access paths is a reasonably effective way to say (in an underapproximate fashion) that heap accesses touch the same address. Also, by using access paths, they have been able to detect many errors in real-world programs (especially in Facebook codebases), proving that the use of access paths can reveal real errors. That is why we have decided to use this principle to represent locks in our analysis.

However, many projects provide their own wrappers for functions like lock/unlock when using low-level locks. For instance, Listing 1 shows a lock wrapper from the VLC project which uses the C/Pthreads library. Using just the access paths in this case will fail and result in many false alarms because all of the locks that would be locked by this wrapper would be named as "`p`". To deal with this problem, all of the function's formal parameters are replaced by the actual ones at the concrete call site in our analyser.

```
1 void vlc_mutex_lock (vlc_mutex_t *p) {
2     int val = pthread_mutex_lock(p);
3     VLC_THREAD_ASSERT("locking mutex");
4 }
```

Listing 1: Lock wrapper.

3.2 Deadlock Detection

We divide our explanation of L2D2 into two parts. First, we will show the main idea of our analyser and describe the entities we use for our compositional deadlock detection (Subsection 3.2.1), and the second part is devoted to a detailed description of an algorithm used to calculate these entities (Subsection 3.2.2).

3.2.1 An Illustrating Example

To help ground the following discussion, the principle of L2D2 will be illustrated by the example in Listing 2.

Workflow of L2D2. L2D2 works in two phases. In the first phase, it computes a summary for each function by looking for lock and unlock events present in the function. An example of a lock and unlock event is illustrated in our example on lines 5 and 8. If a call of a user-defined function appears in the analysed code during the analysis, like at line 7 of our example, the analyser is provided with a summary of the function if available. Otherwise, the function is analysed on demand (which effectively leads to analysing the code along the call tree, starting at its leaves, as usual in INFER). The summary is then applied to an *abstract state* at the call site. Hence, in our example, the summary of `foo` will be applied to the abstract state of `thread1`.

In the second phase, L2D2 looks through all the computed summaries of the analysed program and concentrates on so-called *dependencies* that are part of the summaries and represent a possible locking behaviour of an analysed program. L2D2 interprets the obtained set of dependencies as a relation, computes its transitive closure, and reports a deadlock if some lock depends on itself in the transitive closure.

```
1  void foo() {
2      pthread_mutex_lock(&L2);
3  }
4  void *thread1(...) {
5      pthread_mutex_lock(&L1);
6      ...
7      foo();
8      pthread_mutex_unlock(&L1);
9  }
10 void *thread2(...) {
11     pthread_mutex_lock(&L2);
12     ...
13     pthread_mutex_lock(&L1);
14 }
```

Listing 2: A simple example illustrating a deadlock between two global locks in the C language using the POSIX threads execution model.

Structure of Function Summaries. To detect potential deadlocks, we need to record information that will allow us to answer the following questions:

(1) What is the state of the locks used in the analysed program at a given program point?

(2) Could a cyclic dependency on pending lock requests occur?

To answer Question (1), we compute sets *lockset* and *unlockset*, which contain the currently locked and the currently unlocked locks, respectively. These sets are also a part of the postconditions of functions and record what locks are locked/unlocked upon returning from a function. Further, we also compute sets *locked* and *unlocked* that serve as a precondition for a given function and contain locks that should be locked or unlocked before calling this function.

Each summary contains also a set of *dependencies* using which we can answer Question (2). The dependencies record that some lock got locked at a moment when another lock was still held. For example, if lock `L2` is in the current lockset (which means it is currently

locked) and lock `L1` has just been acquired the dependency `L2`→`L1` will be emitted. This exact situation can be seen in Listing 2 on line 13.

To be able to create these dependencies interprocedurally, we had to add two more sets to the summary to solve the following issues:

(1) What if some of the locks which were acquired in a callee were also released there?

(2) What if the lock from the lockset of a caller was unlocked in the callee before another lock was locked there?

Both of this situations are illustrated in Listing 3. The first case is represented by locks `L3` and `L1` in function `g`, which are unlocked at the end of this function (lines 6 and 7). In this case, these locks will not be in the set of the currently held locks (lockset), and we have no information that they were locked. As a result, we would not create any dependencies, which could lead to false negatives (we could miss some real errors). Therefore, we added a *wereLocked* set to the summary which contains all the locks that were locked in the function. Thus, the wereLocked set for function `g` in our example will look like this: `wereLocked = {L1, L3}`. So now, if we call function `g`, as on line 11 in our example, we have information about which locks were locked there and we can create dependencies with the already acquired lock `L2`.

However, using only this set would create a non-existent dependency `L2`→`L1` since lock `L2` was unlocked before locking lock `L1` (line 3). This situation represents the second of the problems that arise when creating dependencies between multiple functions. In order to avoid this problem, we create dependencies of the `unlock`→`lock` type in the summaries, that can be used to safely determine the order of operations in the callee. This finally ensures that the only newly created dependency in our example will be `L2`→`L3`. These newly created dependencies are stored in a set called *order*.

```
1  void g() {
2      pthread_mutex_lock(&L3);
3      pthread_mutex_unlock(&L2);
4      pthread_mutex_lock(&L1);
5      ...
6      pthread_mutex_unlock(&L1);
7      pthread_mutex_unlock(&L3);
8  }
9  void f() {
10     pthread_mutex_lock(&L2);
11     g();
12 }
```

Listing 3: A motivation example showing the problems with creating lock dependencies between multiple functions.

To sum it up, the summary is a 6-tuple consisting of sets *locked*, *unlocked*, *lockset*, *unlockset*, *wereLocked*, *order*, and *dependencies*. Out of these sets, sets *locked* and *unlocked* form a precondition for a given function and the remaining four sets represent its postcondition. An example of how the summaries for the functions in Listing 2 looked like is shown in Listing 4 (for simplicity, there are only sets that are not empty in the listing).

```
1  foo()
2    PRECONDITION:
3      unlocked = { L2 }
4    POSTCONDITION:
5      lockset = { L2 }, wereLocked = { L2 }
6  thread1(...)
7    PRECONDITION:
8      unlocked = { L1, L2 }
9    POSTCONDITION:
10     lockset = { L2 }, unlockset = { L1 }, wereLocked = { L1, L2 },
11     dependencies = { L1->L2 }
12 thread2(...)
13   PRECONDITION:
14     unlocked = { L1, L2 }
15   POSTCONDITION:
16     lockset = { L1, L2 }, wereLocked = { L1, L2 },
17     dependencies = { L2->L1 }
```

Listing 4: Summaries of the functions from Listing 2.

3.2.2 Algorithm Details

The algorithm for the summary computation is given in Listing 5. Throughout the following explanation we use `L` to denote the most recently locked/unlocked lock. In our explanation, we also use the `state` variable which represents the abstract state in the specific program point of the analysed function and has the same type as the above described summary.

As we said before, L2D2 looks for lock/unlock events in the function being analysed and also tracks calls of user-defined functions. Upon encountering each of these statements, it calls the corresponding abstract transformer which updates the abstract state. So, together, we have three transformers, and in our explanation we will describe each of them separately in the following:

(1) The **acquire** transformer is called upon every lock acquisition and takes an abstract state `state` and a currently acquired lock `L` as the input. First, it updates the precondition for the analysed function `f` by asking if the encountered lock operation is the first operation with that lock in the function `f` (line 3). If this condition is true (the lock is not in the set `locked` nor `unlocked` yet), the lock `L` is added to the `unlocked` set. Intuitively, this reflects the fact that the lock should be unlocked before calling `f` — otherwise, we would encounter double locking[1]. Since this set contains locks unlocked before calling the analysed function, local locks are not added to it. Subsequently, the acquisition itself takes place, which, in our case, means that the lock `L` is added to the `lockset` and removed from the `unlockset`, if it is there. The acquired lock is also added to the set of all locks which were locked by the function `f` (line 8). Since this set is used to create interprocedural dependencies, we only add locks that are not local to it. The final part of the transformer is an extraction of new dependencies and order edges. To extract `dependecies`, we iterate over every

[1]In the basic algorithm of Listing 5, the sets locked/unlocked are maintained, but not used. We will show how they can be used in Section 3.4.

lock in the current `lockset`, emiting the constraint produced by the current acquisition. The extraction of `order` edges works the same, but instead of iterating over the current lockset, it iterates over the `unlockset`.

(2) The **release** transformer works similarly like the acquire one, but it is called upon every lock release. Initially, it adjusts the precondition of the analysed function (lines 18 and 19). Then it releases the lock `L`, which, in our case, means that the lock is removed from the `lockset` and added to the `unlockset`.

(3) The **integrate_summary** transformer applies a summary of a `callee` (denoting the name of a called function) to the abstract state. L2D2 first finds the `summary` of the `callee` and updates it by replacing formal parameters with the actual ones provided that some of the locks were passed to the `callee` as parameters. We also check that all the locks that should be locked/unlocked before calling the `callee` are present in the current `lockset` or `unlockset`, respectively. If they are not, it means that they must be locked/unlocked even before the currently analysed function, and so we update its precondition (lines 27–30). Next, we edit the set of currently held locks (`lockset`) by first adding the locks acquired in the `callee` and then removing all locks which were released there (line 31). The set of currently released locks (`unlockset`) is updated by removing locks acquired in the callee and then adding locks which were released there (line 32). The decision to compute the lockset and unlockset as described comes from the observation that if some lock will appear at the same time in both the lockset and unlockset, then we consider the lock to be unlocked to decrease the number of possible false alarms (indeed, the analysis must have caused the situation by the over-approximation used — in reality, it is not possible to have a lock both locked and unlocked). An example of a situation where a lock may appear in both sets is shown in Listing 7, which will be discussed later on. Before the last step, it is necessary to extend the `wereLocked` set by adding locks acquired in the `callee`. Finally, the last step is to add new `dependencies` between the currently held locks and locks acquired in the callee. This is accomplished by iterating over all the locks in the current `lockset` and generating a dependency with locks in the `wereLocked` set of `callee` for each of them. However, before adding a newly created dependency `X`→`Y`, we still need to verify whether the lock `X` from the current lockset was not unlocked before locking the lock `Y` in the `callee` (line 38).

At run time, our analyser also has to deal with combining states along confluent program paths (e.g., `if` statements). For this purpose, a join operator is defined, which takes two different abstract states as an input and combines them to produce an output state. Since we are interested in locking patterns along any possible path, our join operator is simply the union of incoming states' values for all the sets in the summaries. The widening operation is defined simply as the join operation as we are working with finite-domain summaries and we do not need any accumulation at loop points.

After processing all the instructions in the analysed function, all local locks are removed from the lockset and unlockset, as they are destroyed after the function is completed. This will prevent local locks from being propagated to other functions. The modified last abstract state (without any local locks) is declared as the summary of the analysed function and is applicable in all contexts where this function is called.

```
1  let acquire ( L, state ) {
2   (* Is it a first operation with the lock? *)
3   if (L ∉ state.locked && L ∉ state.unlocked && not(is_local(L))) then
4       add L state.unlocked
5   add L state.lockset
6   remove L state.unlockset
7   if ( not(is_local(L)) ) then
8       add L state.wereLocked
9
10  for X in state.lockset
11      add X->L state.dependencies
12  for Y in state.unlockset
13      add Y->L state.order
14 }
15
16 let release ( L, state ) {
17  (* Is it a first operation with the lock? *)
18  if (L ∉ state.locked && L ∉ state.unlocked && not(is_local(L))) then
19      add L state.locked
20  remove L state.lockset
21  add L state.unlockset
22 }
23
24 let integrate_summary ( callee, state ) {
25  summary:= read_summary callee
26  replace_formals_with_actuals summary
27  if (∃Lock: Lock ∈ summary.unlocked && Lock ∉ state.unlockset) then
28      add Lock state.unlocked
29  if (∃Lock: Lock ∈ summary.locked && Lock ∉ state.lockset) then
30      add Lock state.locked
31  state.lockset:= (state.lockset ∪ summary.lockset) \ summary.unlockset
32  state.unlockset:= (state.unlockset \ summary.lockset) ∪ summary.unlockset
33  state.wereLocked:= state.wereLocked ∪ summary.wereLocked
34
35  for X in state.lockset
36      for Y in summary.wereLocked
37          (* if X was not unlocked before Y was locked *)
38          if( X->Y ∉ summary.order ) then
39              add X->Y state.dependencies
40 }
```

Listing 5: The algorithm used for summary computation in L2D2.

3.3 Reporting Possible Deadlocks

In this section we describe the core deadlock checking algorithm. We begin by noting that our algorithm currently reports deadlocks between two locks only. Deadlocks between more than two locks are also possible and it is easy to detect them using our approach. A Problem arises in reporting: we are unable to tell between which locks the deadlock occurs. However, empirical evidence from bug databases such as http://issues.apache.org, shows that the vast majority of deadlocks involve two locks only (in fact, we could not find any deadlock that involves more than two locks).

The deadlock detection itself takes place after the summaries for all functions in the analysed program are calculated. L2D2 then merges all of the emitted locking dependency constraints into one set. This set is interpreted as a relation, and its transitive closure is computed. If any lock depends on itself in the transitive closure, our analyser will find dependencies that have caused the deadlock and will report a deadlock between the locks of these dependencies[2]. Every deadlock found by our analyser is reported twice — at the beginning of each of the two conflicting locking sequences.

If we run L2D2 on our example in Listing 2, it will report a possible deadlock due to the cyclic dependency between `L1` and `L2` that arises if thread 1 holds `L1` and waits on `L2` and thread 2 holds `L2` and waits on `L1` (see Listing 6). This is caused by dependencies `L1`→`L2` and `L2`→`L1` in the summaries of `thread1` and `thread2`.

```
Found 2 issues
pthr.c:21: error: DEADLOCK
 Deadlock between: lock L1 on line 21 -> lock L2 on line 17, in "thread1"
                   lock L2 on line 28 -> lock L1 on line 29, in "thread2"
  19.
  20.   void *thread1() {
  21. >     pthread_mutex_lock(&L1);
  22.       foo();
  23.
pthr.c:28: error: DEADLOCK
 Deadlock between: lock L2 on line 28 -> lock L1 on line 29, in "thread2"
                   lock L1 on line 21 -> lock L2 on line 17, in "thread1"
  26.
  27.   void *thread2() {
  28. >     pthread_mutex_lock(&L2);
  29.       pthread_mutex_lock(&L1);
  30.   }
Summary of the reports
  DEADLOCK: 2
```

Listing 6: The output of L2D2 in case we run it on the example from Listing 2.

[2] A deadlock is represented in the transitive closure R^* as a lock dependency (a, a). Then, in the case of a deadlock between two locks, we look for the lock $b : (a, b) \in R^* \wedge (b, a) \in R^*$, and report the dependencies. However, if a deadlock occurs between, three and more locks, the current algorithm is unable to find the path that caused the dependency (a, a), e.g., $a \to b, b \to c, c \to a$.

3.4 Reducing the Number of False Alarms

Like many static analysers, our tool also reports false positives. In our case, they are caused by false locking dependencies created during the summary computation. Wrong dependencies are caused by invalid locksets, and the main reasons for it are imprecision in dealing with conditionals (all outcomes are considered as possible), function calls (missing context), and lock aliasing. False positives reduce the usability of the tool because verifying that a report is false can be tedious. This section describes technique we use to reduce the number of false positives.

Errors caused by the lack of context and aliasing of locks are not subject of our optimisations as we would violate the principles described in Section 3.1. Specifically, by adding some context, we would lose the compositionality, and alias analysis would incur overhead which reduce the scalability. That is why, the technique used focuses on errors caused by infeasible sets of paths through the program. It is this group of errors that, according to our observations, produced the greatest number of mistakes in locksets. Almost all false dependencies arise from situations like the one in Listing 7.

```
1 void bar(int x) {
2     if(x)
3         pthread_mutex_lock(&L);
4     ...
5     if(x)
6         pthread_mutex_unlock(&L);
7 }
```

Listing 7: An example of data-dependent locking operations. Since L2D2 does not perform a path-sensitive analysis, it will believe that there are four paths through the `bar` function, one of which locks `L`, but does not release it. This invalid path will result in a wrong lockset containing lock `L` which causes a prompt generation of spurious dependencies.

The method we use seeks to eliminate these errors by pruning the lockset on paths that contain a *locking error*. By a locking error, we mean the situation of double locking or double unlocking. If such an error occurs during the analysis, we can look at it from two perspectives. First, it can really be a locking error, which means the system is in an inconsistent state. Secondly, this may be the case when our analysis has made a mistake and the subsequent reports are neither surprising nor trustworthy. Therefore, L2D2 distinguishes between two run modes:

(1) In the first mode, the analysis runs in the classical way as described in the previous section, and in the case of a locking error, the analyser will emit a warning so the user can inspect the reported warnings.

(2) In the second mode, the analyser will not report warnings, but will adjust the current lockset as shown in Listing 8. In the case of a lock acquisition, we ask whether it is a double locking (line 3). If this is the case, it is assumed that our analysis considered some non-existent path, and the current lockset is no longer trustworthy. Therefore, it is emptied, and the only lock left in it is the currently acquired one, as this is the only one about which we can safely say that it is locked.

We also check double locking/unlocking when a function call appears in the analysed code. This consists in checking the precondition of a callee, specifically asking whether some lock that should be unlocked is currently held, or whether some lock that should be locked is currently released (line 13 and 14). In case such a lock is found, it is again assumed that L2D2 used a non-existent path to reach the function call, and therefore the current lockset is discarded and as the new one the lockset of the callee will be used.

The last algorithm modification can be found in the `release` function. The modification consists in checking whether the currently released lock is unlocked yet (line 8). If it is, the current lockset is simply set to empty, thereby eliminating any dependencies that would be caused by the locking error.

```
1  let acquire ( A, state )
2      (* double locking *)
3      if ( A ∈ state.lockset) then
4          state.lockset = { A }
5      ...
6  let release ( A, state )
7      (* double unlocking *)
8      if ( A ∈ state.unlockset ) then
9          state.lockset = {}
10     ...
11 let integrate_summary ( callee, state )
12     (* double locking/unlocking *)
13     if ( state.lockset ∩ callee.unlocked ≠ {} ∨
14          state.unlockset ∩ callee.locked ≠ {} ) then
15         state.lockset = callee.lockset
16     ...
```

Listing 8: Extensions of the algorithm from Listing 5 to reduce the number of false positives.

Chapter 4

Implementation and Experiments

The first section of this chapter gives the reader a "recipe" on what it takes to expand Facebook Infer with a new analysis, while concentrating on our analyser L2D2. The next section gives a short guide to L2D2's use, and the last one presents results of an experimental evaluation of L2D2.

L2D2 is implemented as a specialised program analysis using the Infer.AI analysis framework. Its implementation is currently hosted in a GitLab Git repository[1].

The repository contains a clone of the entire Infer tool extended by our analyser. Also, it contains all the information (install instructions, dependencies) necessary to get Infer (containing L2D2) working. In addition, the repository also includes the entire benchmark on which the prototype of our analyser was tested, along with the test results. The source code is provided under the MIT licence[2].

The code of our analyser is written in OCaml[3] (as Infer itself) and its main part can be found in files `Deadlock.ml` and `DeadlockDomain.ml`.

4.1 Analyser Plugin

To be able to extend Infer.AI by a new analyser, one has to provide the framework with:

(1) an **abstract domain** (symbolic values for the analysis to track),

(2) **transfer functions** that specify how program statements transform values from the abstract domain,

(3) and a **way to make a summary** of a function that is independent of its calling context (only when considering an interprocedural analysis).

As a result, one obtains an analysis that works compositionally and can (often) scale to big code. Moreover, without any additional effort, the analysis can run on all of the languages Infer supports (C/C++, Obj-C and Java). However, in practice, it is not so simple. For example, in the case of our analyser, we also need to create models of the appropriate library functions — specifically, locking mechanisms.

[1] https://pajda.fit.vutbr.cz/xmarci10/fbinfer_concurrency

[2] https://opensource.org/licenses/MIT

[3] Ocaml is an industrial strength programming language supporting functional, imperative, and object-oriented styles. More details at http://www.ocaml.org/.

Abstract Domain. In order to create a new abstract domain in INFER, there is a need to define the values determined by the required signature (see Listing 9). Specifically, it is necessary to define the type of an abstract state that the analysis will use (`astate`). Next, it is necessary to define the `join` and `widen` operators discussed in Chapter 2 and the comparison operator `(<=)` used to determine a partial order over the abstract states (which is needed when checking whether the analysis reached a fixpoint). As the last step, it is necessary to define the `pp` function (pretty print), which is used to printout the abstract state of our analyser.

```
1  module type S = sig
2    (* the type of an abstract state *)
3    type astate
4    (* the partial order induced by join *)
5    val ( <= ) : lhs:astate -> rhs:astate -> bool
6    val join : astate -> astate -> astate
7    val widen : prev:astate -> next:astate -> num_iters:int -> astate
8    val pp : F.formatter -> astate -> unit
9  end
```

Listing 9: The required signatures of the operations needed in an abstract domain. If one is not familiar with the syntax used in this listing, its specific meaning can be understood as a C language declaration. For example, the widen's declaration in the C language would look like: **`struct astate widen(struct astate prev, struct astate next, int num_iters);`**. The definition of all these operations for the L2D2 analyser can be found in the `DeadlockDomain.ml` file.

Transfer Functions. Transfer Functions are represented as a separate module whose main part is the function `exec_instr`. In this function, one has to define an abstract transformer for every `SIL/HIL` instruction. The function takes as its input a `SIL/HIL` instruction and the current abstract state and applies the effect of the instruction to it. The output is then a new abstract state with which the analyser continues to work.

In our case, the `CALL` instruction is the only one that changes the abstract state. After recognising the call in the analysed code, we then identify whether it is a locking event[4]. We find this out by comparing the name of the called function with models[5] of lock/unlock functions and then the corresponding transformer is called for each of these events as described in Subsection 3.2.2. The Definition of the `Transfer Functions` module can be found in `Deadlock.ml` file.

Interprocedural analysis. By defining the abstract domain and the `Transfer Functions` module, it is possible to create an analyser that works intraprocedurally only.

So, in order to create an interprocedural analysis, one has to define the type of summaries, add logic for using summaries in transfer functions, and define a mechanism of conversion between an intraprocedural abstract state and a summary.

[4] A locking event means locking or unlocking one of the locks in the analysed program.

[5] Under the models, we can easily imagine the names of the functions used to acquire or release the lock (e.g.: „`pthread_mutex_lock`" represents the model for the function from the Pthreads library that acquires the lock).

The summary type must be registered in the files `Payloads.ml` and `Payloads.mli`. All further work with summaries, such as saving, loading and updating, is provided by INFER. In the case of our analysis, the type of the abstract state and summary is the same, which means that it is not necessary to convert an abstract state to a summary. Generally, this conversion occurs after the completion of a single function analysis, whose last abstract state is converted and saved as a summary for future use.

The last issue to solve is to extend the transfer functions with the transformer that instantiates the summary of a callee at the current call site. To do that, we also check for calls of user defined functions. We recognise these calls in a way that if it is not a locking event, we will try to read the summary of the callee (`Payload.read_summary pdesc callee_pname`). If the summary cannot be read, it means that it is a call of a library function whose definition we do not have, and it will not affect the abstract state at the current program point. Otherwise, the read summary is instantiated as described in Section 3.2.2.

Putting it all together. Once one defines all of the above mentioned parts, one can finally introduce a new analysis into INFER. This is achieved by instantiating the INFER's abstract interpreter to which we, as one of the parameters, give the `Transfer Functions` module tied with our abstract domain as shown below:

```
module L2D2 =
    LowerHil.MakeAbstractInterpreter (ProcCfg.Normal) (TransferFunctions)
```

So, now, we have the abstract interpreter module (L2D2) that we can use for the analysis. It offers many useful functions where the main one is `compute_post`, which takes a function as its input and computes its postcondition. We invoke it in a callback (`Deadlock.checker`; see Listing 10) which is called by INFER upon every function of the analysed program in the order given by a callgraph.

```
let checker {Callbacks.proc_desc; tenv; summary} =
  ...
  match L2D2.compute_post proc_data ~initial: DeadlockDomain.empty with
  | Some postcondition ->
    (* here one can update the postcondition or report some errors *)
    ...
    (**
       Saving the postcondition as a summary of the analysed function.
       If the types of abstract state and summary are different,
       there is the place to convert it.
    *)
    Payload.update_summary postcondition
  | None -> ()
```

Listing 10: The callback, which is used to run an analysis over each function in the analysed program. The basic structure shown is mandatory for each INFER analyser. A complete definition of this callback for L2D2 can be found in the `Deadlock.ml` file.

The last step is to hook the new analyser up to the INFER CLI. This means that we can run it from CLI along with other analyses in INFER. To do that, we have to register our `Deadlock.checker` callback in the `RegisterCheckers.ml` module. Besides, it requires

the creation of a record of our analyser in the `Config.ml` module, where one has to define a CLI option that will run our analyser, and there is also a message to be displayed in the help after entering `infer --help`.

Reporting. For reporting, we use the `Reporting.ml` module also provided by Infer, and we distinguish two types of reports in our analyser.

The first of them is a deadlock report that is reported as an error. The algorithm used for reporting deadlocks has been described in Section 3.3, and an example of the report is shown in Listing 6. Deadlock reporting is defined in a special callback (`Deadlock.reporting`) that will be called by Infer after the analysis of all functions has been completed. It is also necessary to register this callback in the `RegisterCheckers.ml` module.

The second kind of reports are warnings when there is a situation of double locking or double unlocking. These warnings are generated after the analysis of each function, and the reporting is called in the `Deadlock.checker` callback mentioned above.

To use the `Reporting.ml` module, each of the mentioned types of errors must be additionally defined in the `IssueType.ml` file.

4.2 Usage

L2D2 itself accepts its input in the form of C/C++ files and prints all detected errors/warnings on the standard output. It is also possible to choose the mode in which the analysis will run by using the appropriate option (see Listing 11).

```
(* default mode *)
infer --deadlock-only -- gcc -c source_code.c
(**
    suppressing the warnings and using the heuristics
    described in Section 3.4
*)
infer --locking_error --deadlock-only -- gcc -c source_code.c
```

Listing 11: Possible instances of running L2D2 analyser.

4.3 Experimental Evaluation

In this section, we will experimentally evaluate our analyser using two independent test sets consisting of concurrent C programs using the POSIX threads execution model. We will also compare our analyser with the Cprover tool mentioned in Section 2.3.

4.3.1 Basic Examples

The first set consists of 349 student projects with total 82 949 LOC[6]; all programs in this test set have been verified to have no deadlocks. This fact was successfully proved by our analyser when it concluded the analysis with zero false positives rate in both of its modes. To analyse these programs, L2D2 needed only 3 minutes, which demonstrates its scalability.

[6] Lines of code were measured using `cloc` 1.74.

4.3.2 Cprover Test-Suite

The analyser was also put through a testing with a set of 1 002 concurrent C programs derived from the Debian GNU/Linux distribution. These programs were originally used for an experimental evaluation of Daniel Kroening's static deadlock analyser [16] implemented in the CPROVER framework with which we have compared our results. The entire benchmark, along with the test results, is also available online at `https://bit.ly/2WJBQLQ`.

The benchmark set consists of 11.4 MLOC. Of all the programs, 994 are assumed to be deadlock-free, and 8 of them contain a known deadlock. Our experiments were run on a CORE i7-7700HQ processor at 2.8 GHz running Ubuntu 18.04 with 64-bit binaries. The CPROVER experiments were run on a Xeon X5667 at 3GHz running Fedora 20 with 64-bit binaries. In the case of CPROVER, the memory and the CPU time were restricted to 24 GB and 1 800 seconds per benchmark, respectively.

Results. Like CPROVER, L2D2 was able to detect all eight (potential) deadlocks in both of its modes. A comparison of the results obtained for deadlock-free programs can be seen in Table 4.1. The table gives numbers of programs shown as deadlock-free, numbers of false alarms, number of time-outs and memory-outs, and, finally, numbers of programs for which the analysis failed.

Table 4.1: Results for programs without a deadlock (t/o — timed out, m/o — out of memory). Within the table, we recognise two modes of our analyser. Namely, $mode_1$ which is the default mode of L2D2, and $mode_2$ which uses the above described optimisations. For a more detailed description of each mode, see Section 3.4, mode (1) and mode (2).

	proved	alarms	t/o	m/o	failures
CPROVER	**292**	114	453	135	0
$L2D2_{mode1}$	**833**	83	0	0	78
$L2D2_{mode2}$	**877**	39	0	0	78

As one can see, our analyser has achieved better results than CPROVER in both its modes. The first mode of our analyser reported by 31 fewer false positives than the CPROVER, and the second mode even reported by 75 false alarms fewer. A much larger difference can be seen in the cases where it was proved that there was no deadlock. Here the differences against CPROVER are as follows: 541 examples in favour of the first mode and 585 examples in favour of the second mode.

The only aspect where L2D2 was worse compared to CPROVER is the number of failures of the analysers. For L2D2, the failures were caused by the syntax of analysed programs that INFER does not support. On the other hand, CPROVER did not manage 588 examples that were unsuccessful either because of the lack of time or memory needed to verify them.

Finally, the most significant difference was in the time needed for the benchmarks analysis. While CPROVER needed 300 hours for the analysis, L2D2 needed an average of 1 hour and 45 minutes, which is less than 1 % of the time of CPROVER. These results successfully demonstrate the scalability of our solution.

To sum it up. The L2D2 analyser running in $mode_2$ is the best of this comparison. However, its disadvantage over $mode_1$ is the fact that it may not reveal a bigger number of real errors due to the approximations it uses to reduce false positives. On the other hand, the first mode produces more false errors which represent a higher burden on the programmer to control them.

Chapter 5

Conclusion

In this thesis, we have presented the L2D2 analyser which is based on a new method for deadlock detection that handles low-level locking and that fits the compositional analysis style used in the Facebook INFER static analysis framework. We have implemented our analyser as one of the INFER checkers and applied it to a suite of multi-threaded C programs (using Pthreads library) comprising 11.4 MLOC. While unsound and incomplete, our approach shows as effective, finding all of the known deadlocks in the test suite with reporting false positives in 3.9 % of the tested programs only. We have also successfully demonstrated the scalability of our approach, needing only 1 hour and 45 minutes to analyse 1 002 concurrent programs.

Future work will focus on further increasing the accuracy of our approach. For that, we might use more advanced techniques to reduce the number of false alarms. The plan is to, for example, implement the modified *unlockset analysis* introduced at [8]. Its main idea is: at a program point p, remove any lock L from the current lockset if there exists no successor statement s reachable from p that contains an unlock of L. If L does not reach any unlock operation after the program point p on any subsequent path, then it is almost certain our analysis has made a mistake. Another possibility is to perform various kinds of filtering of the obtained alarms: e.g.: (1) removing cases of deadlocks that appear multiple times, since it is likely that many errors of the same kind would be spotted by the programmer or (2) combining our analyser with a dynamic analysis based on the noise injection, where noise can be inserted in the vicinity of problematic locks and thus maximise the chance of revealing real errors, etc. Also, we would like to extend the reporting module to the ability to report deadlocks between more than two locks and also provide the user with more detailed reasons why deadlocks (may) occur. The last plan is to find a more appropriate way to determine the lock aliasing.

Bibliography

[1] Bessey, A.; Block, K.; Chelf, B.; et al.: A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM*. vol. 53, no. 2. February 2010: pp. 66–75. ISSN 0001-0782. doi:10.1145/1646353.1646374. Retrieved from: http://doi.acm.org/10.1145/1646353.1646374

[2] Blackshear, S.; Gorogiannis, N.; W. O'Hearn, P.; et al.: RacerD: compositional static race detection. *Proceedings of the ACM on Programming Languages*. vol. 2. 10 2018: pp. 1–28. doi:10.1145/3276514.

[3] Calcagno, C.; Distefano, D.; Dubreil, J.; et al.: Moving Fast with Software Verification. In *NASA Formal Methods*, edited by K. Havelund; G. Holzmann; R. Joshi. Cham: Springer International Publishing. 2015. ISBN 978-3-319-17524-9. pp. 3–11.

[4] Cousot, P.: Abstract Interpretation Based Formal Methods and Future Challenges. In *Informatics - 10 Years Back. 10 Years Ahead.*. Berlin, Heidelberg: Springer-Verlag. 2001. ISBN 3-540-41635-8. pp. 138–156. Retrieved from: http://dl.acm.org/citation.cfm?id=647348.724445

[5] Cousot, P.: Basic concepts of abstract interpretation. August 4 to August 16, 2009 2009. Summer School Marktoberdorf 2009: Logics and Languages for Reliability and Security.

[6] Cousot, P.; Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY. 1977. pp. 238–252.

[7] Cousot, P.; Cousot, R.: Abstract Interpretation Frameworks. *Journal of Logic and Computation*. vol. 2, no. 4. 08 1992: pp. 511–547. ISSN 0955-792X. doi:10.1093/logcom/2.4.511. Retrieved from: https://doi.org/10.1093/logcom/2.4.511

[8] Engler, D.; Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. New York, NY, USA: ACM. 2003. ISBN 1-58113-757-5. pp. 237–252. doi:10.1145/945445.945468. Retrieved from: http://doi.acm.org/10.1145/945445.945468

[9] Fiedor, J.; Hrubá, V.; Křena, B.; et al.: Advances in Noise-based Testing of Concurrent Software. *Softw. Test. Verif. Reliab.*. vol. 25, no. 3. May 2015: pp. 272–309. ISSN 0960-0833. doi:10.1002/stvr.1546.
Retrieved from: http://dx.doi.org/10.1002/stvr.1546

[10] Harmim, D.; Marcin, V.; Onřej, P.: Scalable Static Analysis Using Facebook Infer. In *Excel@FIT'19*. Brno, Czech Republic. 2019.
Retrieved from: http://excel.fit.vutbr.cz/submissions/2019/059/59.pdf

[11] Harrow, J. J.: Runtime Checking of Multithreaded Applications with Visual Threads. In *SPIN Model Checking and Software Verification*, edited by K. Havelund; J. Penix; W. Visser. Berlin, Heidelberg: Springer Berlin Heidelberg. 2000. ISBN 978-3-540-45297-3. pp. 331–342.

[12] Havelund, K.: Using Runtime Analysis to Guide Model Checking of Java Programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. London, UK, UK: Springer-Verlag. 2000. ISBN 3-540-41030-9. pp. 245–264.
Retrieved from: http://dl.acm.org/citation.cfm?id=645880.672085

[13] Hoare, C. A. R.: An Axiomatic Basis for Computer Programming. *Commun. ACM*. vol. 12, no. 10. October 1969: pp. 576–580. ISSN 0001-0782. doi:10.1145/363235.363259.
Retrieved from: http://doi.acm.org/10.1145/363235.363259

[14] Jones, N. D.; Muchnick, S. S.: Flow Analysis and Optimization of LISP-like Structures. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '79. New York, NY, USA: ACM. 1979. pp. 244–256. doi:10.1145/567752.567776.
Retrieved from: http://doi.acm.org/10.1145/567752.567776

[15] Joshi, P.; Park, C.-S.; Sen, K.; et al.: A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. New York, NY, USA: ACM. 2009. ISBN 978-1-60558-392-1. pp. 110–120. doi:10.1145/1542476.1542489.
Retrieved from: http://doi.acm.org/10.1145/1542476.1542489

[16] Kroening, D.; Poetzl, D.; Schrammel, P.; et al.: Sound Static Deadlock Analysis for C/Pthreads. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. New York, NY, USA: ACM. 2016. ISBN 978-1-4503-3845-5. pp. 379–390. doi:10.1145/2970276.2970309.
Retrieved from: http://doi.acm.org/10.1145/2970276.2970309

[17] Leino, K. R. M.: Extended Static Checking. In *Proceedings of the IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods*. PROCOMET '98. London, UK, UK: Chapman & Hall, Ltd.. 1998. ISBN 0-412-83760-9. pp. 1–2.
Retrieved from: http://dl.acm.org/citation.cfm?id=647321.721335

[18] Naik, M.; Park, C.-S.; Sen, K.; et al.: Effective Static Deadlock Detection. In *Proceedings of the 31st International Conference on Software Engineering*. ICSE '09.

Washington, DC, USA: IEEE Computer Society. 2009. ISBN 978-1-4244-3453-4. pp. 386–396. doi:10.1109/ICSE.2009.5070538.
Retrieved from: http://dx.doi.org/10.1109/ICSE.2009.5070538

[19] O'Hearn, P.: Separation Logic. *Commun. ACM*. vol. 62, no. 2. January 2019: pp. 86–95. ISSN 0001-0782. doi:10.1145/3211968.
Retrieved from: http://doi.acm.org/10.1145/3211968

[20] O'Hearn, P. W.; Reynolds, J. C.; Yang, H.: Local Reasoning About Programs That Alter Data Structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*. CSL '01. London, UK, UK: Springer-Verlag. 2001. ISBN 3-540-42554-3. pp. 1–19.
Retrieved from: http://dl.acm.org/citation.cfm?id=647851.737404

[21] Russell, J.; Cohn, R.: *Rice's Theorem*. Book on Demand. 2012. ISBN 9785511707440.

[22] Sipser, M.: *Introduction to the Theory of Computation*. International Thomson Publishing. first edition. 1996. ISBN 053494728X. "Section 4.2: The Halting Problem".

[23] Sterling, N.: Warlock: A static data race analysis tool. In *Proceedings of the 1993 USENIX Winter Technical Conference*. 1993. pp. 97–106.

[24] Williams, A.; Thies, W.; Ernst, M. D.: Static Deadlock Detection for Java Libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming*. ECOOP'05. Berlin, Heidelberg: Springer-Verlag. 2005. ISBN 3-540-27992-X, 978-3-540-27992-1. pp. 602–629. doi:10.1007/11531142_26.
Retrieved from: http://dx.doi.org/10.1007/11531142_26

Appendix A

Storage Medium

`/L2D2/*` — source code of Infer (containing L2D2) from date May 15, 2019

`/README.txt` — useful information about the storage medium content

`/text/*` — source code of this thesis

`/xmarci10.pdf` — final version of this thesis