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 THE META INFER FRAMEWORK TO DETECT DATA RACES

STATICKÁ ANALÝZA V NÁSTROJI META INFER ZAMĚŘENÁ NA DETEKCI SOUBĚHU NAD DATY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **LUCIE SVOBODOVÁ**
AUTOR PRÁCE

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

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

148636

Institut:	Department of Intelligent Systems (UITS)
Student:	**Svobodová Lucie**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Statická analýza v nástroji Meta Infer zaměřená na detekci souběhu nad daty**
Category:	Software analysis and testing
Academic year:	2022/23

Assignment:

1. Get acquainted with principles of static analysis. Pay special attention to abstract interpretation and to analyses designed for concurrent programs.
2. Get acquainted with the Meta Infer framework, its support for abstract interpretation, and existing analysers under this framework, especially the L2D2 and Atomer checkers.
3. Design and implement within the Meta Infer framework an analysis for discovering potential data races in multithreaded C programs.
4. Evaluate experimentally the designed analyser on suitably chosen programs, including at least some real-life project.
5. Summarize the achieved results and discuss possibilities of future improvements of the designed analyser.

Literature:

- Rival, X., Yi, K.: Introduction to Static Analysis: An Abstract Interpretation Perspective. MIT Press, 2020.
- Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling Static Analyses at Facebook. Commun. ACM, 62(8):62-70, ACM, 2019.
- Engler, D.R., Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks, In: Proc. of SOSP'03, ACM, 2003.
- Blackshear, S., Gorogiannis, N., O'Hearn, P.W., Sergey, I.: RacerD: Compositional Static Race Detection, In: Proc. of OOPSLA'18, ACM, 2018.
- Liu, B., Liu, P., Li, Y., Tsai, C.-C., Da Silva, D., Huang, J.: When Threads Meet Events: Efficient and Precise Static Race Detection with Origins, In: Proc. of OOPSLA'21, ACM, 2021.
- Harmim, D.: Advanced Static Analysis of Atomicity in Concurrent Programs through Facebook Infer. Master thesis, Brno University of Technology, 2021.
- Marcin, V.: Static Analysis Using Facebook Infer Focused on Deadlock Detection. Bachelor thesis, Brno University of Technology, 2019.

Requirements for the semestral defence:
The first two points and at least some work on the third one.

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

Supervisor:	**Vojnar Tomáš, prof. Ing., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	3.11.2022

Abstract

Modern software systems often use concurrent programs to improve performance and increase efficiency. However, ensuring the reliability and safety of such systems can be challenging due to the increased potential for bugs, including data races, to arise. In this thesis, we introduce a new static data race detector, *DarC*, designed for programs written in C using the Pthreads library. The proposed detector is implemented as an analyser plugin in Meta Infer, a static analysis framework with the emphasis on compositional, incremental, and consequently highly-scalable analysis. Our approach involves recording a set of accesses that occur in the analysed program along with information about the set of locks held during these accesses. The tool then identifies pairs of accesses that may lead to data races and reports them to the user. Our tool was successfully evaluated on a set of benchmarking programs as well as on real life projects, showing its potential for effectively detecting data races in C programs.

Abstrakt

Vícevláknové programy jsou v moderních softwarových systémech využívány ke zlepšení výkonu a zvýšení efektivity. Zajištění spolehlivosti a bezpečnosti takových programů však může být náročné kvůli zvýšenému množství chyb, které se v nich vyskytují, včetně souběhu nad daty. V této práci představujeme nový statický detektor souběhu nad daty, *DarC*, navržený k analýze programů napsaných v jazyce C využívajících knihovnu Pthreads. Navrhovaný nástroj byl implementován jako zásuvný modul prostředí Meta Infer, což je nástroj pro statickou analýzu programů, který klade důraz na kompoziční, inkrementální a díky tomu i vysoce škálující analýzu programů. Nový analyzátor zaznamenává množinu přístupů ke sdíleným proměnným, ke kterým v analyzovaném programu došlo, spolu s informací o množině zámků uzamknutých při jednotlivých přístupech. Množina přístupů je dále použita k identifikaci dvojic přístupů, mezi nimiž by k souběhu nad daty mohlo dojít. Nástroj byl úspěšně ověřen na sadě testovacích vícevláknových programů, stejně tak jako na několika programech běžně využívaných v praxi, čímž byl ukázán jeho potenciál pro efektivní detekci souběhu nad daty v programech napsaných v programovacím jazyce C.

Keywords

static analysis, data race, Infer, program analysis, abstract interpretation, scalability, multithreaded programs, concurrent programs, program verification, incremental analysis

Klíčová slova

statická analýza, souběh nad daty, Infer, analýza programů, abstraktní interpretace, škálovatelnost, vícevláknové programy, paralelní programy, verifikace programů, inkrementální analýza

Reference

SVOBODOVÁ, Lucie. *Static Analysis Using the Meta Infer Framework to Detect Data Races.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

Rozšířený abstrakt

Paralelní programování je důležitým aspektem vývoje moderního softwaru, který se uplatňuje jak u programů provádějících vysoce výkonné výpočty, tak u běžných webových aplikací. Nicméně, paralelní programy jsou často mnohem komplexnější a složitější na porozumění, testování i na následné ladění než klasické sekvenční programy, a to především kvůli jejich nedeterministickému chování a nutnosti synchronizace paralelních vláken tak, aby se vyloučily jejich nežádoucí interakce. Chyba souběhu nad daty (*data race*) je jednou z nejčastěji se vyskytujících chyb v paralelních programech. Souběh nad daty nastává, pokud ve vícevláknovém programu přistupuje ke sdílenému místu v paměti více vláken současně, přičemž tyto přístupy nejsou správně synchronizované a alespoň jedno vlákno na danou paměťovou lokaci zapisuje data. To může vést k neočekávanému a nepředvídatelnému chování, protože není zaručeno pořadí, ve kterém vlákna k dané paměťové lokaci přistoupí. K předejití tomuto typu chyby je potřeba správně synchronizovat všechny přístupy ke sdílené paměti, například s použitím zámků. Nicméně, zvyšování počtu zámků použitých v programu může vést k nesprávnému pořadí jejich zamykání a odemykání, což může následně způsobit jiný druh chyby, uváznutí (*deadlock*). Souběh nad daty, uváznutí, porušení atomicity a další v programu vyskytující se chyby mohou způsobovat nesprávné výsledky, neočekávané chování systému i úplné výpadky systému, což z nich činí kritické problémy při zajištění spolehlivosti a bezpečnosti softwaru.

Tradiční techniky testování a ladění softwaru jsou často nedostačující při hledání chyb ve vícevláknových programech, protože tyto chyby bývají závislé na specifickém pořadí provádění instrukcí jednotlivými vlákny. Ke zvýšení pravděpodobnosti odhalení takových chyb je možné použít jiných přístupů, například *systematického testování* [26] nebo *testování založeného na vkládání šumu* [7]. Dalším efektivním přístupem je použití *dynamických analyzátorů*, například [8, 22, 14], které sledují běh programu a na základě různých metod identifikují potenciální chyby, i když v daném běhu přímo nenastanou. Monitorování běhu velkého programu však může být velmi časově náročné, a navíc ani tyto analyzátory nemusí objevit všechny chyby, ke kterým v programu může dojít. K vyřešení posledního zmíněného problému je možné použít například metodu *model checking*, která je přesná a umožňuje detekci všech chyb, které mohou v programu nastat, přičemž neprodukuje žádná falešná hlášení o chybách (*false alarms*). Vzhledem k tomu, že tato technika funguje na prohledávání velkých částí stavového prostoru analyzovaného programu, není snadné ji použít na rozsáhlé softwarové projekty. Je možno užít i takzvaný *omezený model checking*, díky kterému můžeme omezit například maximální počet kroků či přepnutí kontextu, ale jeho cena je také nemalá a při velkém omezení může snadno nenalézt hlubší chyby.

Statická analýza programů představuje alternativu k výše zmíněným technikám. Na rozdíl od většiny dynamických analyzátorů je možné ji použít k identifikaci chyb v rané fázi vývoje programu, kdy ještě není dostupná spustitelná verze. Díky tomu je možné odhalit chyby, jejichž oprava by byla časově i finančně velmi náročná při pozdějším odhalení. Při provádění statické analýzy není samotný program spouštěn, ale je analyzována pouze syntax zdrojových kódů, případně jejich vhodným způsobem abstrahovaná sémantika, s cílem identifikovat potenciální chyby, zranitelnosti i výkonnostní problémy. Škáluje často lépe než model checking a může nacházet některé chyby, které dynamická analýza nenalezne, ovšem za cenu toho, že některé chyby přehlédne, nebo naopak nahlásí i chyby, které při běhu programu nastat nemohou. Oblast statické analýzy je velmi rozsáhlá a zahrnuje velké množství

přístupů, například analýzu toku dat (*data-flow analysis*) a abstraktní interpretaci (*abstract interpretation*), která je použita i v této práci.

V praxi je používáno mnoho prostředí pro statickou analýzu programů, řada z nich je ale komerční a silně uzavřená, jako například nástroje *Coverity* [28] nebo *KlocWork* [30]. Existují však i nástroje, jejichž zdrojové programy jsou otevřené a rozšiřitelné. Mezi tyto nástroje se řadí *Frama-C* [5] a *Meta Infer* [29]. V rámci této práce se soustředíme právě na nástroj Infer, mezi jehož hlavní výhody patří inkrementální, kompoziční a interprocedurální analýza směrem zdola nahoru dle stromu volání funkcí, díky čemuž je Infer vhodný i pro analýzu rozsáhlých softwarových projektů. Infer poskytuje jednotlivé analyzátory pro detekci různých typů chyb, jako například chyby dereference prázdného ukazatele (*null dereferencing*), přetečení vyrovnávací paměti (*buffer overflow*), nebo únik paměti (*memory leaks*). Výhodou prostředí Infer je také možnost přidání vlastních analyzátorů zaměřujících se na detekci dalších typů chyb.

V rámci této práce představujeme nový statický detektor souběhu nad daty, který je navržen pro analýzu programů napsaných v programovacím jazyce C s použitím nízkoúrovňového zamykání a vláken s použitím knihovny *Pthreads*. Navržený analyzátor byl implementován jako zásuvný modul prostředí *Infer*, ve kterém takový analyzátor dosud scházel. Náš přístup je založen na detekci všech přístupů ke sdíleným proměnným, ke kterým v analyzovaném programu došlo, spolu s informací o množině zámků, které byly zamknuté při daných přístupech. Ze získané množiny přístupů jsou následně vyfiltrovány dvojice přístupů, mezi kterými by mohlo dojít k souběhu nad daty, a tyto dvojice jsou následně nahlášeny uživateli. Potenciál využití nově navrženého analyzátoru byl demonstrován jak na sadě programů vytvořených k testování analyzátorů chyb vyskytujících se v paralelních programech, tak na několika softwarových projektech využívaných v běžné praxi.

Static Analysis Using the Meta Infer Framework to Detect Data Races

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. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Lucie Svobodová
May 9, 2023

Acknowledgements

I would like to thank my supervisor Tomáš Vojnar for his support during the work on this thesis. I would also like to thank Ondřej Pavela and Dominik Harmim for their technical assistance regarding the Infer framework, and to Tomáš Dacík for his help in obtaining the materials for the experimental evaluation. Lastly, i am grateful for the support received from the Czech Science Foundation project AIDE and the Horizon Europe project CHESS.

Contents

Chapter 1

Introduction

Concurrent programming is an essential aspect of modern software development, with applications ranging from high-performance computing to web applications. However, concurrent programs are also more complex and harder to understand, test, and debug than sequential programs due to their non-deterministic behavior and the necessity to suitably synchronize their various concurrently running threads. Improper synchronization may lead to various nasty kinds of bugs that manifest only rarely, are hard to spot and debug, but can be highly destructive. *Data races* are one of the most common and frequent kinds of such bugs in concurrent programs. A data race is a problem that occurs when multiple threads access the same memory location simultaneously without proper synchronization, and at least one of these accesses is a write access. This can lead to unexpected and unpredictable behavior as the order in which the threads access the memory location is not guaranteed. To avoid data races, proper synchronization mechanisms such as locks can be used to ensure that only one thread can access the shared memory at a time. However, increasing the number of locks in the program can lead to other bugs such as deadlocks, making the program even more complex and harder to debug. Data races, deadlocks, atomicity violations and other concurrency bugs can cause unpredictable behavior, incorrect results, and system crashes, making them critical challenges for ensuring the reliability and safety of software.

Traditional testing and debugging techniques are often insufficient for detecting and fixing concurrency bugs due to the already mentioned fact that they depend on specific timing and order of execution, making them hard to reproduce and diagnose. To increase the likelihood of detecting rare behaviors in concurrent programs, different approaches such as *systematic testing* [26] and *noise-based testing* [7] can be used. Another effective approach is to use *dynamic analysers* like [8, 22, 14], which can identify potential errors by observing the program's behavior during its execution, looking for symptoms of potential bugs even if no bugs manifest during the test runs. Unfortunately, monitoring a run of a large program may be quite expensive and time-consuming, and these analysers can still miss errors. On the other hand, approaches based on *model checking* may be precise and capable of detecting all errors in a program without producing false alarms. However, they may search potentially very large parts of the state space of the program, which may be infeasible for larger programs. Model checking can of course be *bounded* (e.g., in the number of context switches or the number of steps that a program may perform), leading to pragmatic error detection that has already reached many successes in practice. However, apart from that,

such an approach may miss some deeper bugs, it may still be quite expensive for larger programs, it requires one to prepare a test harness, and may have problems with certain program constructions (input/output operations).

Static analysis is an alternative to the above approaches. It can help identify and fix problems early on, reducing the risk of costly and time-consuming bugs later on. It can also help to improve the overall quality and maintainability of the software. Static analysis is a technique for evaluating the properties of software without executing the code, which may involve analysing the syntax of the source code or its appropriately abstracted semantics to identify potential bugs, vulnerabilities, and performance issues. It can often scale better than model checking and find bugs not found by dynamic analysis, though for the price of potentially missing some errors and/or producing false alarms. The area of static analysis is very extensive and includes many different approaches, such as *data-flow analysis* and *abstract interpretation*, which will be discussed later in this thesis.

Static analysis frameworks are widely used in software development to automatically identify potential bugs and vulnerabilities in code. While many of these frameworks are proprietary, such as *Coverity* [28] and *KlocWork* [30], there are also open-source alternatives like *Frama-C* [5] and *Meta Infer* [29]. In this work, we focus on the latter framework, which provides several advantages including a bottom-up approach suitable for incrementality and high scalability. Infer is not only a framework that provides multiple checkers for various types of bugs, such as null dereferencing, buffer overflows, and memory leaks, but it also allows for the addition of new analyser plugins focused on other types of bugs.

Within this work, we present *DarC* – a new *static data race detector designed for low-level C code*. The proposed analyser is implemented as a plugin of the Meta Infer framework, which was so far missing such a detector. It is based on recording the set of accesses to shared variables that occur in the program, along with the information about synchronization between them. These accesses are then filtered in order to exclude those that are properly synchronized, and finally, the pairs of accesses between which a potential data race occurs are reported. The proposed analyser showed promising results in our experiments performed on a set of smaller benchmarks developed for evaluating concurrency testing tools as well as on real-life projects.

Structure of the thesis. The rest of this thesis is structured as follows. Chapter 2 provides an introduction to static analysis and one of its approaches – abstract interpretation. The Meta Infer framework is described in Section 2.2. Section 2.3 then presents some of the existing tools designed for data race detection. Chapter 3 discusses the design of our new data race detector, followed by the implementation details in Chapter 4. Chapter 5 presents the results of our experimental evaluation of the data race detector, including benchmarks developed for testing tools for finding concurrency bugs and real-life projects. Finally, Chapter 6 concludes the thesis and discusses possible future work.

Chapter 2

Preliminaries

This chapter provides the theoretical background for the thesis. It begins with a discussion of static analysis in Section 2.1, followed by an explanation of abstract interpretation on which the Meta Infer framework is based. Meta Infer will be then described in Section 2.2. Finally, Section 2.3 provides an overview of existing solutions for data race detection, covering both static and dynamic analysers.

2.1 Static Analysis

Static analysis is a method of analysing a program without actually executing the code [17]. This is done by examining the source code and looking for potential issues or bugs. The goal of static analysis is to identify potential problems before the software is run, so that they can be fixed before they cause any issues. Static analysis methods based on formal methods can be used to prove the correctness of a program with respect to a specification, but they can have limited scalability and often result in a large number of false alarms. Static tools that are neither sound (may miss some bugs) nor complete (may report false alarms) may be practical for analysing real-life software projects. The goal is not to find all bugs but rather to find bugs that are the most likely to be critical [18].

Static analysis can be done automatically, using specialized tools that examine the code's syntax or abstractly analyse the flow of data and control through the code. These tools then look for potential issues, such as bugs, security vulnerabilities, performance bottlenecks, or various coding issues, such as undefined values or dead code. The area of static analysis is very extensive and includes many different approaches, such as data-flow analysis, control-flow analysis, and abstract interpretation, which this work builds on and which will be discussed in the following subsection [17, 25, 24].

2.1.1 Abstract Interpretation

Abstract interpretation (AI) is a general framework for static analysis introduced by French computer scientist Patrick Cousot and his wife Radhia Cousot at POPL'77 [3]. It is a method for analysing the behavior of computer programs by modeling the program's semantics using abstract domains. The goal of abstract interpretation is to identify prop-

erties of the program that are true for all possible inputs (and, in the case of concurrent programs, all possible interleavings), without necessarily analysing all such inputs (interleavings). When certain properties of the components are met, the analysis is guaranteed to be sound. If a potential error is not signaled, then it should be impossible [13].

The basic idea behind abstract interpretation is to replace the program's concrete semantics (i.e. a set of all possible executions in all possible execution environments) with an abstract semantics (i.e. a simplified model of the program's behavior that is easy to analyse). This way the abstract interpretation approximates the program's semantics with a more tractable abstract domain, while preserving certain properties of the program that are relevant for the analysis [2].

When creating a new analysis using abstract interpretation, the following set of components that establish the semantics of the analysis needs to be defined:

- The *abstract domain*: the set of abstract values that the analysis works with, which abstractly represent the possible states of the program being analysed. The choice of the abstract domain depends on the nature of the program under analysis.
- The *transfer functions*: these define how the abstract state of the program changes as it is symbolically executed. They are used to propagate abstract values through the control flow of the program, approximating the effect of each program statement on the abstract state.
- The *join operator* ($\circ$): This operator is used to combine the abstract values of different program points that may have multiple paths leading to them. The join operator takes the information from each path and merges them into a single abstract value.
- The *widening operator* ($\triangle$): This operator is used to ensure that the fixpoint is reached in a finite number of steps when working with loops. A fixpoint is reached when the so-far computed reachable abstract states of the program no longer change in any possible further computation.
- The *narrowing operator* ($\triangledown$): This operator is used to narrow down the set of possible values in order to make the abstract representation more precise and accurate. The narrowing operator is not required for the analysis but can help improve the precision of the results when used in combination with the widening operator.

The formal definition of abstract interpretation is based on the notion of semilattices. A partially ordered set $(A, \leq_A)$ is a join semilattice if each non-empty, finite subset B of A has a least upper bound in A. A partially ordered set $(A, \leq_A)$ is a complete semilattice if each subset B of A has a least upper bound in A [3].

A Galois connection provides a way to establish a correspondence between two lattices, one representing the concrete domain and the other the abstract domain. According to [13], Galois connection is a quadruple $\pi = (\mathcal{P}, \alpha, \gamma, \mathcal{Q})$ such that:

- $\mathcal{P} = \langle P, \leq \rangle$ and $\mathcal{Q} = \langle Q, \sqsubseteq \rangle$ are *partially ordered* sets (posets),
- $\alpha : P \to Q$ and $\gamma : Q \to P$ are functions such that $\forall p \in P$ and $\forall q \in Q$:

$$p \leq \gamma(q) \iff \alpha(p) \sqsubseteq q.$$

In abstract interpretation, Q is the abstract domain, and P is a (more) concrete domain – elements of both domains, called abstract/concrete contexts, represent sets of states [13].

The abstract interpretation I of a program P with the instruction set `Instr` is a tuple:

$$I = (Q, \sqcup, \sqsubseteq, \top, \bot, \tau)$$

where:

- Q is the abstract domain,
- $\sqcup : Q \times Q \to Q$ is the join operator,
- $\sqsubseteq \subseteq Q \times Q$ is an ordering defined as $x \sqsubseteq y \iff x \sqcup y = y$ where $(Q, \sqsubseteq)$ is a complete (join semi-)lattice,
- $\top \in Q$ is the supremum of $(Q, \sqsubseteq)$,
- $\bot \in Q$ is the infimum of $(Q, \sqsubseteq)$,
- τ : `Instr` $\times Q \to Q$ defines the abstract transformers for particular instructions, required to be monotone on Q for each instruction from `Instr` [13].

The soundness of abstract interpretation is guaranteed when each instruction from `Instr` and the corresponding abstract transformer τ_i respect the Galois connection [4, 18, 13].

2.2 Meta Infer Framework

Infer is an open-source[1] framework for creating highly-scalable, compositional, incremental, and interprocedural static analysers based on the abstract interpretation. It was developed by Meta/Facebook, and it is designed for detecting bugs and security vulnerabilities in Java, C, C++, and Objective-C code. Infer provides several analysers that check for various types of bugs, such as null-dereferencing, memory leaks, or buffer overflows (*Inferbo* [27]). As for concurrency-related bugs, Infer provides a support for detecting data races (*RacerD* [1]), deadlocks, and starvation (*Starvation* [31]), but it is limited to Java programs and some cases of C++ programs only. Infer is used at Meta to analyse the codebase of their mobile and web apps, such as Facebook, Instagram, What's up, Spotify, and Amazon [20].

Infer provides both intraprocedural and interprocedural analyses. Intraprocedural analysis computes a summary for a single function, which can be used later by the abstract interpretation framework (Infer.AI) at specific call sites to create an interprocedural analysis. This is what makes Infer compositional and incremental – rather than analysing the entire codebase after each code change, Infer only computes summaries of the functions that were changed (or that were influenced by changed functions), significantly reducing the analysis time and improving scalability. By leveraging this incremental analysis approach, Infer can be run on large codebases after every code change where running the whole analysis from the beginning would be too time-consuming.

The framework architecture consists of three parts: *frontend*, *scheduler + results database*, and *analyser plugins*. A simplified diagram of the framework architecture is presented in Figure 1. At the beginning of the analysis, the frontend parses the source code and translates it into an intermediate representation (IR) that can be easily analysed. There are two intermediate languages used in Infer, the Smallfoot Intermediate Language (`SIL`),

[1] Infer's open-source repository on GitHub: https://github.com/facebook/infer

and the High-level Intermediate Language (HIL). The input program is then represented as a control flow graph (CFG) where each node of the graph consists of SIL or HIL instructions, depending on the intermediate language used by the particular analysis.

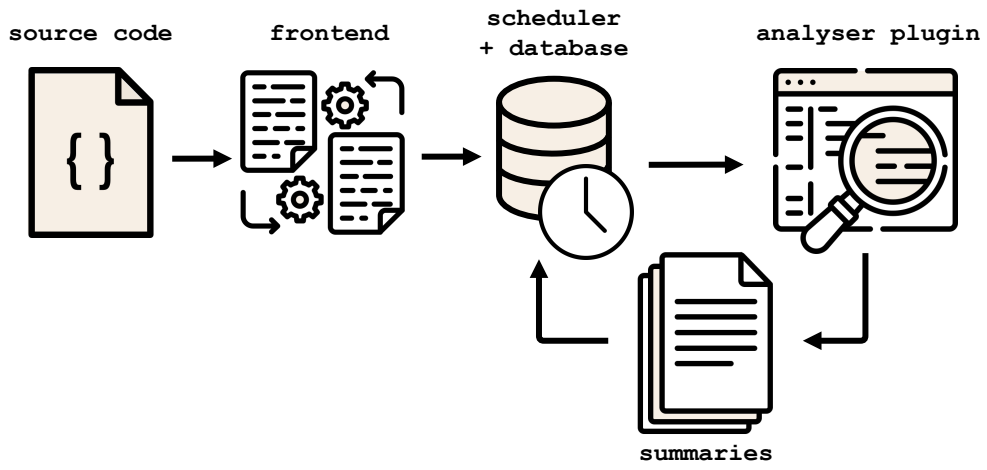

Figure 1: A simplified representation of the Infer.AI framework architecture.

The Smallfoot Intermediate Language is a low-level intermediate representation of the source code, which preserves the structure and control flow of the original program, but abstracts away some of the low-level details, such as memory layout, and machine-specific instructions. It provides four instructions:

- LOAD: loads a value from a memory address denoted by an expression and stores it into a temporary identifier. The address expression can be either a program variable or a more complex expression, e.g., an array-indexing expression or a structure field.
- STORE: stores the value of an expression into a location specified by an address expression. The expression to be stored consists of temporary identifiers created by previous LOAD instructions, constants, exceptions, or more complex operations.
- CALL: represents a function call, provides information about return values, parameters and their types, and call flags.
- PRUNE: splits the control flow of the program into two branches according to the result of a Boolean expression, providing information about the source of the branching, such as if statement, loop, or ternary operator. It is executed twice, once for the true branch and once for the false branch.

The High-level Intermediate Language is a simplified version of SIL that includes only three instructions: CALL, ASSIGN (an abstraction of the STORE instruction in SIL), and ASSUME (an abstraction of the PRUNE instruction). HIL is sufficient for most analysis needs and is easier to work with than SIL. However, HIL does not provide the necessary abstractions for modeling the behavior of pointers and other low-level memory operations, and as a result, it is not suitable for analyses that focus on memory-related bugs.

After creating the control flow graph, Infer.AI is called. It must be instantiated by each checker implemented in Infer. Once Infer.AI is called, the control flow graph is divided into separate functions, which are then analysed separately. The scheduler is responsible for determining the order in which the functions should be analysed by examining the call graph and identifying the dependencies between the functions. The analysis uses the bottom-up approach, whereby the functions located at the leaves of the call graph are analysed first, and then their results are propagated upwards to higher-level functions. An example of such a call graph can be seen in Figure 2.

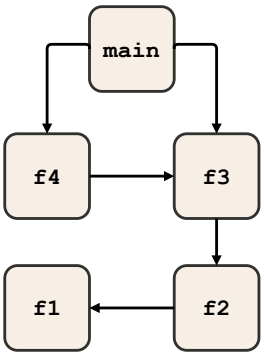

Figure 2: A call graph.

The analysis of a function is performed by traversing its control flow graph and interpreting the `SIL` instructions, which correspond to the nodes of the graph. This interpretation process is performed by an *abstract interpreter* and *transfer functions*, which must be defined for each checker according to the specific analysis requirements. The transfer functions are used to update the abstract state, and this process also involves the use of join and widen operators to ensure that all possible paths are covered in the case a conditional branching or a loop occurs in the program. This process of updating the current abstract state continues until all nodes of the control flow graph have been processed and no further change in the computed abstract state happens.

After a function is analysed, the results are stored in the results database. These results are stored as function summaries, which are data structures containing relevant information about the analysed function. The information stored in the function summaries is defined by each checker independently as the data stored in the summary are relevant to the particular analysis. Function summaries consist of two parts: *preconditions*, which are conditions that are expected to be true before the function starts, and *postconditions*, which are conditions that hold when the function finishes its execution. Some checkers may only use postconditions for the analysis. When the function is called by a higher-level function in the call graph, the summary of the callee function is integrated into the current abstract program state, which makes the analysis interprocedural.

Errors detected by the analyser plugin are reported during the creation of summaries, or after the summary computation is finished, depending on the particular checker [9, 19, 21].

2.3 Related Work on Data Race Detection

There are various tools that can be used for data race detection in concurrent programs. This section lists some of the most popular existing static and dynamic analysers using various approaches to the data race detection.

Dynamic analysis techniques for detecting data races analyse a single execution of a program (or a set of executions) and often rely on either computing *locksets* or *happens-before relations*. Locksets are sets of locks that guard all accesses to a shared variable, and the absence of a data race is ensured when the lockset for a variable is non-empty, indicating that at least one lock is held during each access. Eraser [22] was probably the first algorithm to use locksets for detecting data races. FastTrack [8] is an example of a dynamic analysis

that employs the second approach, a happens-before algorithm, which uses so-called vector clocks [12]. ThreadSanitizer [23] combines both of these approaches, the lockset and the happens-before algorithms, to detect data races. In addition to computing locksets and happens-before relations, some dynamic analysis techniques rely on the use of noise injection. For example, the AtomRace [14] algorithm uses techniques for a careful injection of noise into the scheduling of the monitored programs. Another tool to be mentioned when it comes to dynamic analysers is Helgrind [11], which is a tool developed by the Valgrind[2] project, which uses a combination of instrumentation and thread-aware execution to detect data races in C and C++ programs.

On the other hand, static analysers inspect the source code without running the program. They can look for concrete patterns that are likely to cause a data race, or like dynamic analysers, they are based on computing locksets or happens-before relations. RacerX [6] is a static, flow-sensitive, context-sensitive, and non-compositional analysis for data race and deadlock detection, which uses a top-down approach. It is designed for programs written in C and is also based on storing summaries. To use the analyser, developers must first add annotations to the code. RacerX employs various heuristics to reduce the number of alarms emitted, such as a ranking algorithm. RacerD [1] is a static data race detector that is already implemented in Infer, so it has all the features that were presented above – it is highly scalable, incremental, and compositional. It is designed to detect data races mainly in Java programs, C and C++ are not the primary languages that it supports, and it only reports lock consistency violations for programs written in C++. Coderrect/O2 [15] is an analyser for both C/C++ and Java/Android applications, which leverages origins, an abstraction that unifies threads and events by treating them as entry points of code paths attributed with data pointers.

[2] **Valgrind** – an instrumentation framework for building dynamic analysis tools: https://valgrind.org/

Chapter 3

Data Race Detection

The purpose of this chapter is to present an outline of the design principles that underpin our newly developed analyser. In the first section, we discuss the design principles that guided the development of our data race checker, which works in two phases. Section 3.2 describes the first phase of the data race detection process, which involves tracking memory accesses made by each thread in the analysed program. These accesses are then used for computing potential data races in the second phase, which will be described in Section 3.3.

3.1 Design Principles

The design of the new data race detector is based on the following principles:

1. Bottom-up analysis: The analysis starts at the leaf nodes of the call tree (i.e., the lowest level of functions in the program) and works its way up to the root node (i.e., the main function or another entry point of the program).

2. Interprocedural analysis: The analysis allows for more accurate and comprehensive analysis as it considers the interactions between different procedures in the program, including those in different modules or libraries.

3. Compositional analysis: Each function defined in the program is analysed independently from its context. Only functions that were changed during further development or depend on the changed functions are analysed again, which leads to much higher scalability.

4. Points-to analysis: Variables are represented by access expressions and a light-weight points-to analysis for detecting data races between variables referenced by pointers is performed.

These design principles were chosen to address some of the key challenges in data race detection and have influenced the design of our new data race detector. In addition to these principles, we also had other important considerations in mind when designing our checker. One such consideration was the desire to avoid using annotations in the code. While it is common for analysers like RacerX [6] to require annotations that specify which locks protect which data, we believe that it is better to be able to work without these annotations. This

is particularly important for large projects that have already been developed as adding annotations to them can be a time-consuming and tedious process.

Another important consideration was the need to minimize the number of reported errors and avoid overwhelming developers with too many warnings. To achieve this, our tool is designed to report only one data race for each variable if a data race is found. By doing so, developers can focus on one error at a time, fix it, and then move on to the next one, rather than feeling discouraged by a large number of errors and not fixing any of them.

A data race between two accesses occurs when all of the following conditions are satisfied:

1. The accesses are to the *same memory location*.
2. At least one of the accesses is a *write*.
3. The *intersection of the set of locks* that are locked when accessing the memory location is *empty*.
4. The accesses are from *different threads*.
5. Neither access is to *thread-local* memory.

These conditions led us to propose the structure of function summaries that are computed during the analysis, from which the set of accesses that occurred in each analysed function is especially crucial for the data races computation.

The analysis can be split into two phases: first, computing the set of accesses that occur in the program under analysis on global or shared variables, and second, checking whether there are any pairs of accesses in the set for which the conditions written above hold. For such accesses, data races are then reported. Both phases will be discussed in the following sections.

3.2 Phase 1: Recording a Set of Accesses

The first phase of the analysis involves computing the set of memory accesses that occur in the analysed program. As our analysis uses a bottom-up approach, each function is analysed individually, and a summary of this function is created. These summaries are then propagated in a bottom-up manner from lower-level to higher-level functions. When a recursive call occurs, the function is analysed up to the point of the recursive call, and because at this point the summary for the function being analysed is not yet computed, an empty summary is used to complete the computation. The analysis of the function then continues as usual. When the analysis of all functions used in the program is complete, the resulting set of accesses at the top-level function includes all the accesses that occurred in the program and is used in the second phase to compute the data races.

To help better understand which information needs to be tracked during the process of recording the set of accesses, let us consider a sample program shown in Listing 1. In this example, the `main` function creates a new thread (`thread1`) by calling `pthread_create`, which invokes the `foo` function in the new thread. Meanwhile, the main thread continues to execute and sets the value of the global variable `i` to 42. The `foo` function also accesses `i` but under the protection of a mutex `lock`. However, the lock is not held during the write access by the main thread, which creates a potential data race since it is possible for the two threads to access the same memory location concurrently without proper synchronization.

```c
int i;            // shared variable
void *foo() {
  pthread_mutex_lock(lock);
  i = 0;          // data race
  pthread_mutex_unlock(lock);
}
int main() {
  pthread_create(t1, foo);
  i = 42;         // data race
}
```

Listing 1: A sample code illustrating a data race between two accesses to a shared variable in the C language using POSIX[3] threads.

To detect this data race, we keep track of the set of memory accesses that occur during the execution of the program. Specifically, we record the following information for each memory access:

1. The *variable* that is accessed: variables are represented as so-called *access expressions* (e.g. `&p`, `p`, `*p`, `**p`), which helps with pointer handling.
2. The *location in the code* where the variable is accessed: this information is used to identify where a possible data race could occur in the code.
3. The *type of access*: whether it is a read or write access.
4. The set of *locked locks*: locks that must be locked at the current program point.
5. The set of *unlocked locks*: locks that may be unlocked at the current program point.
6. The set of *active threads*: threads that may be running when the access occurs.
7. The set of *joined threads*: threads that must be joined when the access occurs.
8. The *thread on which the access occurs*: because of the bottom-up approach, the information about the thread on which the access happens is not always available. Therefore, `None` is used when the thread is currently unknown.

Listing 2 shows the set of memory accesses from Listing 1, along with the information for determining whether there could be a data race between those accesses.

```
Accesses:
  (i on line 4, Write, locked_locks={lock}, unlocked_locks={},
                acive_threads={main, t1}, joined_threads={}, on thread t1),

  (i on line 9, Write, locked_locks={}, unlocked_locks={lock},
                active_threads={main, t1}, joined_threads={}, on thread main)
```

Listing 2: A set of accesses to the variable `i` from the example shown in Listing 1. For each access, we record information about the accessed variable, the location where it was accessed, the type of access (in this case, both are write accesses), the set of locked locks, the set of unlocked locks, the set of active threads, the set of joined threads, and the thread executing the access.

[3]POSIX standard: https://pubs.opengroup.org/onlinepubs/9699919799/

Memory accesses are one of the key parts of the abstract states computed by our analysis. In particular, our abstract state consists of the following information:

1. *Accesses*: the set of memory accesses that occur in the function.
2. *Active threads*: the set of threads that may be active at the end of the function.
3. *Join threads*: the set of threads that were joined in the analysed function.
4. *Locked locks*: the set of locks that remain locked after the function ends.
5. *Unlocked locks*: the set of locks that remain unlocked after the function ends.
6. *Local variables*: the set of local variables that are read or written during the function.
7. *Points-to relations*: the set of points-to relations used to correctly identify the memory locations being accessed through pointers.

During the analysis, the abstract states are transformed using abstract state transformers as described in Section 2.2. At the end of the function analysis, the resulting abstract state is saved as the *postcondition* in the function summary. The *precondition* part of the summary is not used in our analysis. Function summaries are then propagated from lower-level functions to higher-level functions along the call tree, as previously described in Section 2.2. The individual components of our abstract state and the propagation of function summaries will be described in more detail in the following subsections.

3.2.1 Threads That May Be Running

During the analysis of a program, the information about currently running threads needs to be stored in order to accurately identify potential data races. The POSIX standard provides the `pthread_create` function for creating new threads where the first argument of the function is the thread to be created. To keep track of all active threads at any given point during the analysis, we maintain the set of active threads called `threads_active`. Each thread in this set is identified by its abstract thread identifier, which is the *access expression* representing the variable used to reference that thread in the program, along with the line number on which the thread was created. When a thread is created by calling the `pthread_create` function, it is added to the `threads_active` set to ensure that it is considered during the analysis. Similarly, when the `pthread_join` function is called, we assume that the thread will no longer be active after that program point as the caller function will wait for the thread to finish its execution before continuing its own execution. Therefore, after joining the thread, it is removed from the current set of active threads. Moreover, when analysing the `main` function we know that the main thread is running, therefore the main thread is created and added to the set of active threads at the beginning of the analysis of `main`.

When dealing with conditional statements where threads may be created or joined, worst-case scenarios are considered. This is, as shown in Listing 3, if a thread is created in only one branch of a conditional branching statement, it is added to the set of active threads as though it was sure that it would always be created. Similarly, if a `pthread_join` function call appears in a conditional statement, we must assume that it needs not happen and that the thread *may still be running* even after the execution of the conditional statement. Therefore, the thread remains in the set of active threads, as shown in Listing 4.

```
1 // threads_active: {}
2 if (x) {
3  pthread_create(&th, NULL, f, NULL);
4    // threads_active: {th}
5 } else {
6  ; // threads_active: {}
7 }
8 // thread th may be running
9 // threads_active: {th}
```

Listing 3: A simple code illustrating the creation of a thread in one branch of a conditional statement. After the branching, we assume that the thread was created regardless of the branch taken.

```
1 // threads_active: {th}
2 if (x) {
3  pthread_join(th, NULL);
4    // threads_active: {}
5 } else {
6  ; // threads_active: {th}
7 }
8 // thread th still may be running
9 // threads_active: {th}
```

Listing 4: An example of joining a thread in a conditional statement. The thread may be joined with the currently running thread, but we assume the thread is still active after line 7 during the analysis.

The `threads_active` set is stored in the summary of the function for the purpose of propagating information about newly created threads into higher-level functions. When a function containing a new thread in its summary is called, the resulting new abstract state is then the union of the `threads_active` set of the callee and the `threads_active` set in the current abstract state.

However, using only the set of active threads to determine whether a thread has been joined in a function is not sufficient. This is because the resulting set of active threads in the summary of a function would be the same in the following three cases:

1. there is no thread created or joined in the function,
2. there is a thread both created and joined in the function,
3. there is only a join of the thread, but the thread is not created before in the function.

The first two cases do not affect the current abstract state if they have the same set of active threads. However, the third case is different because if we only use the set of active threads to determine if a thread has been joined, we would consider a thread running even if it has already been joined. This could lead to many false positives because the accesses that occur from this point in the program on would be taken as running concurrently with the joined thread. To avoid this issue, we keep track of *joined threads* in the summary so that we can distinguish between these cases and propagate information about joined threads correctly to higher-level functions.

When integrating the set of active and joined threads from the summary of a called function (callee) into the current abstract state, the new set of active threads is computed from the current set of active threads ta and the current set of joined threads tj as $((ta \cup ta\_callee) \setminus tj\_callee)$ where ta_callee and tj_callee are the sets of active and joined threads from the summary of the callee, respectively. The new set of joined sets is then computed as $((tj \setminus ta\_callee) \cup tj\_callee)$.

In case there is a `pthread_join` function call in a conditional statement, and it occurs only in one branch, the thread is not added to the set of joined threads as it may still be running in the other branch. The same logic as in the `threads_active` case is used where the set of joined threads is updated only if the join occurs in all possible execution paths.

The threads in the set of joined threads are identified in the same way as in `threads_active`, that is, by using the abstract thread identifier and the location where the thread was created. When a function only contains a call to `pthread_join` and the thread being joined was not created within the same function and is not in the `threads_active` set, the location where the thread was created is unknown at that point in the analysis. To handle this, the thread is added to the set of joined threads with a negative line of code as the location, indicating that the thread's creation location is unknown. When a higher-level function calls the function where the thread was joined, it checks whether the thread is in the current abstract state and updates the `threads_joined` set with the location of the thread's creation to maintain consistency between the sets of active and joined threads. The same applies when a `pthread_join` function call precedes the `pthread_create` call for the same thread within the same function.

3.2.2 Locks That Must Be Locked

To identify whether multiple accesses to a variable are properly synchronized, it is essential to keep track of the locks that are locked at a given program point. To achieve this, we store a set of locked locks, called `lockset`, during the analysis. Whenever a lock is acquired using the `pthread_mutex_lock` function, the lock passed to the function as an argument is added to this lockset. When a lock is released through the `pthread_mutex_unlock` function, the lock is removed from the `lockset`. This means that the lock is no longer locked after this program point.

The `lockset` contains a set of locks that *must be locked* for proper synchronization. Hence, opposite to the case of thread creation, if a lock is acquired in only one branch of the statement and not in the other, we do not add the lock to the set, as shown in Listing 5. This approach assumes the worst-case scenario where the branch in which the lock is not locked would be executed. In such a case, the lock would not be locked, and the access that would occur after the conditional statement would not be synchronized by the lock. Similarly, if a lock is released in only one branch of a conditional statement, we assume that the branch without the `pthread_mutex_unlock` function call will be executed. Therefore, the lock is not removed from the set of currently locked locks. This case is shown in Listing 6.

```
1 // lockset: {}
2 if (x) {
3   pthread_mutex_lock(lock);
4     // lockset: {lock}
5 } else {
6   ; // lockset: {}
7 }
8 // lock lock may still be unlocked
9 // lockset: {}
```

Listing 5: Example of a code snippet demonstrating that when a lock is locked in only one branch, it may still be unlocked after the branching, and therefore it is not added to `lockset`.

```
1 // lockset: {lock}
2 if (x) {
3   pthread_mutex_unlock(lock);
4     // lockset: {}
5 } else {
6   ; // lockset: {lock}
7 }
8 // lock lock may be unlocked
9 // lockset: {}
```

Listing 6: Example of unlocking a lock in only one branch of a conditional statement. After the branching, the lock may be unlocked, and thus it is removed from the set of locked locks.

However, in addition to the lockset, it is also important to keep track of the set of unlocked locks, called `unlockset`, during the analysis. The unlockset is crucial in accurately identifying data races. Without the unlockset, it is possible to miss real data races if a lock was unlocked in a function and the information is not recorded in the unlockset. For example, consider the code snippet in Listing 7, where a lock is unlocked in the function `foo` without being recorded in the `unlockset`. In this case, the `lockset` in the summary of the function `foo` would be empty, but the lock `m` would still be present in the `lockset` of the caller function `bar`. This would incorrectly record the access to `i` in `bar` as properly synchronized by the lock `m`, potentially leading to a false negative if there is simultaneous access to `i` on another thread.

To avoid this issue, we record the information about unlocking the lock `m` in `foo` in the *unlockset*. When integrating the summary of `foo` into `bar`, we update the *lockset* of `bar` to not contain `m` anymore because it was unlocked in `foo`, and therefore the access to `i` in `bar` will be correctly identified as unsynchronized.

When integrating the lockset and unlockset from the summary of a called function (callee) into the current abstract state with the lockset ls and unlockset us, the new lockset is computed as $((ls \cup ls\_callee) \setminus us\_callee)$, and the unlockset is computed as $((us \setminus ls\_callee) \cup us\_callee)$ where ls_callee and us_callee are the lockset and the unlockset from the summary of the called function, respectively.

```
1  int i;
2  pthread_mutex_t *lock;
3  void foo() {
4    pthread_mutex_unlock(lock);
5  }
6  void bar() {
7    pthread_mutex_lock(lock);
8    foo();
9    // lock is unlocked at this point
10   i = 0;
11 }
```

Listing 7: Example of unlocking a lock in a function call, demonstrating the need of storing the set of unlocked locks, `unlockset`. This set helps determine that the access to the variable `i` on line 9 in the function `bar` is no longer guarded by the lock `lock` due to its unlocking in the function `foo`, which could lead to a data race.

The unlockset keeps the set of locks that *may be unlocked* during program execution. In the case of conditional branching, if the `pthread_mutex_unlock` function call occurs only in one branch, it is assumed that the lock may be unlocked regardless of the branch taken. Therefore, the lock is added to the `unlockset` and removed from the `lockset`, as it is possible that the accesses that occur after the branching will not be properly guarded by the lock, leading to a potential data race.

3.2.3 Thread-local Memory

In order to accurately decide whether thread-local memory is accessed, it is necessary to store the set of variables that are local to the function that is currently analysed. However,

when a new thread is created by calling the `pthread_create` function, the function's local variables may no longer be thread-local. Thus, to ensure that the analysis remains accurate, any variable that is passed to a new thread as the fourth argument of the `pthread_create` function must be removed from the set of local variables being tracked. An example of passing a local variable to another thread is listed in Listing 8.

```
1 void *f(void *arg) {
2   ...
3 }
4 int main() {
5   int *p = ...;
6   // p is local
7   pthread_create(&th, NULL, f, p);
8   // p is now shared with the function f
9 }
```

Listing 8: An example of passing a local variable `p` to a newly created thread `th`. Once the thread is created, the variable becomes shared and can be accessed from multiple threads.

3.2.4 Points-to Analysis

Since the proposed checker is designed to detect data races in programs written in C, where the memory is commonly manipulated using pointers, including a points-to analysis was an important part of the design. A points-to analysis attempts to determine the addresses that a pointer holds. For example, in a C program, two pointers may point to the same memory location, and if the program writes to that memory location through one pointer, the value read through the other will be affected. Without tracking the set of variables that may alias, it would be impossible to detect that both accesses are to the same memory location, and therefore data races for these types of accesses would not be detected. Thus, during the analysis of a function, a set of points-to relations is recorded.

Points-to relations computed in our analysis consist of pairs *(p, a)* where the variable *p may point to the memory location a*. The memory location *a* can be represented in two ways. If *a* is an address of a variable v allocated on the stack, the corresponding points-to relation is then *(p, &v)*, indicating that the variable *p* may point to the address of the variable *v*. For memory locations allocated on the heap, we represent the address as the location in the code where the memory was allocated, such as by calling the `malloc` function. For example, if the memory is dynamically allocated on line 2 and assigned to a pointer variable *q*, the corresponding points-to pair is denoted as *(p, line 2)*. Both types of points-to relations can be seen in Listing 9.

```
1 ...
2 int v;                          // points-to relations:
3 int *p = &v;                    // (p, &v)
4 int *q = malloc(sizeof(int));   // (q, line 4)
5 ...
```

Listing 9: An example of points-to relations in C code where the pointer variable `p` points to the address of the variable `v`, represented as `(p, &v)`, and the pointer variable `q` points to a dynamically allocated memory location on line 4, represented as `(q, line 4)`.

Listing 10 demonstrates how pointers can point to other pointers in C code and how the points-to relations are updated during pointer assignments. On line 2, a pointer variable `y` is assigned the address of an integer variable `x`, creating a points-to relation $(y, \&x)$ where y may point to the address of x. Then, a pointer to a pointer variable `m` is assigned the address of `y`, creating a new points-to relation $(m, \&y)$. When the pointer variable `m` is dereferenced on line 7, the analysis looks at the points-to set and retrieves all the relations that have `m` on the left side, which is the relation $(m, \&y)$ in this case. Since `m` is being dereferenced, the analysis knows how to retrieve the value pointed to by `m`, which is the address of `y`. It then returns the points-to relation that has `y` on the left side, which is $(y, \&x)$ in this case. This updates the points-to relation from $(y, \&x)$ to $(y, \&z)$, meaning that `y` now points to the address of `z`.

When a points-to relation is created inside a conditional statement, such as an if-else statement, the analysis must consider all possible paths. Specifically, if a points-to relation is created in one branch of the conditional statement, it is added to the set of points-to relations, even if there is a different points-to relation created in the other branch, as shown in Listing 11. This is because the points-to relation is of the *may-points-to* type, and we must consider all variables that may alias to make sure that any data race is not missed during the analysis.

```
1  int x;
2  int *y = &x;
3  // y -> x
4  int **m = &y;
5  // m -> y -> x
6  int z = 0;
7  *m = &z;
8  // m -> y -> z
```

Listing 10: Example demonstrating the points-to relations that are kept during the analysis. The notation `y -> x` indicates that the variable `y` may point to the address of variable `x`. Similarly, the notation `m -> y -> z` indicates that the variable `m` may point to the address of variable `y`, and the variable `y` may point to the address of variable `z`.

```
1  int i, j, *p;
2  if (x) {
3    p = &i;  // alias: p -> i
4  } else {
5    p = &j;  // alias: p -> j
6  }
7  //      ,-> i
8  //   p -|
9  //      '-> j
```

Listing 11: Example illustrating points-to relations in a conditional statement. The pointer variable `p` is assigned the address of `i` in one branch and the address of `j` in the other branch. The points-to set includes both possibilities after the branching, indicating that `p` may point to either `i` or `j`.

When analysing a function, the formal arguments that are of pointer type are initially added to the points-to set with a default negative location to indicate that their addresses are unknown at this point. During the analysis, the points-to relations are updated as described above. Finally, only the points-to relations that refer to shared variables are kept in the summary, which ensures that the analysis does not have to store all aliases, even those of local variables. When a higher-level function calls a function, the formal parameters of the called function are replaced with the actual parameters. The points-to set for the called function is then propagated to the higher-level function, allowing the analysis to track aliases across function calls.

3.2.5 Accessing a Variable

The key component in data race detection is the set of accesses that occur in the analysed function. Each access is added to this set along with additional information that is used later to detect possible data races. As we have already mentioned at the beginning of Section 3.2, a memory access consists of the variable that is accessed, along with the information about the location in the code where the access occurs, the type of access, sets of locked and unlocked locks, sets of active and joined threads, and the thread on which the access occurs. Each time a variable is accessed in the program, the access is added to the set of accesses. At this point, the access stores information about the current state of the program and holds the information described above.

The thread, sets of locks, and sets of threads are updated later on when more information from the functions higher up in the call tree is collected and when the summaries are integrated into the abstract state. For this, the same approach is used as when integrating the summary of the called function (callee) to the summary of the caller. However, instead of using the sets of the summary of the callee, we use the sets recorded in the access. In each access, the lockset and unlockset are updated as described in Subsection 3.2.2, and the sets of active and joined threads are updated as described in Subsection 3.2.1. The thread on which the access occurs is updated in a specific manner. If there is already information about the thread on which the access occurred, then the thread is not updated any further. When the information is not yet known, and the `main` function is currently analysed, the main thread is added as the thread on which the accesses occur. If a summary of a callback function of `pthread_create` is being integrated, and therefore the thread that was just created is known, then this thread is added as the thread on which the accesses occur. When the thread is not yet known, the information remains unknown.

A sample code shown in Listing 12 will be used to help ground the description of how the accesses are updated and the summaries integrated. The code illustrates a data race between two accesses to the shared variable `x`. The program creates two threads, `t1` in the `main` function, and `t2` in the function `foo`. The thread `t2` accesses the variable `x` while holding a lock `m`, which is then unlocked by the function `foo`. Meanwhile, the main thread writes to `x` without any locks held in the function `main`.

The summaries for the functions from Listing 12 are shown in Listing 13. The summary for the function `bar` includes the lock `m` in the lockset since this lock was acquired by the `pthread_mutex_lock` function, and it was not released. Since no threads are created or joined in the function, the sets of active threads (`active_th`) and joined threads (`joined_th`) are empty. There is one access to the global variable `x` on line 7. At this point, the sets of locks and threads in the access capture the current state when the access occurs, therefore these are the same as in the summary. The thread on which the access occurs is not yet known, but it will be added later when the summary is propagated to a higher-level function. Since there are no pointer variables in the code, the points-to set is not shown in the summary as it would be empty for all functions.

Next, the function `foo` is analysed. A new thread `t2` is created on line 11 by calling the `pthread_create` function and passing the function `bar` as its argument. This involves adding the newly created thread `t2` into the current set of active threads and integrating the summary of `bar` into the current abstract state. The set of accesses from the summary of `bar` is updated with the current set of active threads, which contains the thread `t2`.

```
1  int x;
2  pthread_t t1, t2;
3  pthread_mutex_t m;
4
5  void *bar() {
6    pthread_mutex_lock(&m);
7    x = 10;
8  }
9
10 void* foo() {
11   pthread_create(&t2, bar);
12   pthread_mutex_unlock(&m);
13 }
14
15 int main() {
16   pthread_create(&t1, foo);
17   x = 0;
18 }
```

Listing 12: A sample code illustrating a data race between two write accesses to the variable `x`. The first access occurs on the thread `t2` in the function `bar`. The access is protected by a lock `m`, which is then unlocked in the function `foo`. The second access is performed by the `main` thread in the `main` function, but it is not protected by any lock. These two accesses lead to a data race.

```
1  bar:
2    lockset={m}, unlockset={},
3    active_th={}, joined_th={},
4    accesses={(write to x on line 7,
                lockset={m},unlockset={},
                active_th={}, joined_th={},
                on thread ⊥)}
5  foo:
6    lockset={}, unlockset={m},
7    active_th={t2}, joined_th={},
8    accesses={(write to x on line 7,
                lockset={m},unlockset={},
                active_th={t2}, joined_th={},
                on thread t2)}
9  main:
10   lockset={}, unlockset={m},
11   active_th={main,t1,t2}, joined_th={},
12   accesses={(write to x on line 7,
                lockset={m},unlockset={},
                active_th={main,t1,t2},
                joined_th={},
                on thread t2),
13              (write to x on line 17,
14               lockset={},unlockset={m},
15               active_th={main,t1,t2},
16               joined_th={},
17               on thread main)}
```

Listing 13: The summaries of the functions `bar`, `foo`, and `main` from Listing 12. The information from `bar` and `foo` is propagated to the summary of the `main` function, which holds information about all accesses that occur in the program.

We now know that the function `bar` is executing on the thread `t2`, therefore the information about the thread is added to the accesses in the summary of `bar`. These accesses are then added to the set of accesses in `foo`. Finally, `pthread_mutex_unlock` is called, which removes the lock `m` from the current `lockset` and adds it to the `unlockset` of `foo`.

Moving on to the `main` function, at the beginning of its analysis, the `main` thread is added to the set of active threads. Next, a new thread `t1` is created on line 16 by calling `pthread_create` and passing `foo` as the callback function. This involves adding the thread `t1` into the current set of active threads and integrating the summary of `foo` into the current state. The set of accesses from the summary of `foo` is updated with the current `threads_active` set. Accesses from `foo` are updated with the thread `t1` as the thread on which the access occurs, but because the only access in `foo` already contains the information about the thread, it is not edited. The set of accesses from `foo` is then added to the set of accesses in `main`. Next, on line 17, there is a write access to the variable `x`. The information about the current abstract state is captured in the sets of locks and threads in the new access record. Because it is known that the analysis of the main function is currently being done, the main thread is added as the thread on which the access on line

17 occurs. This access is then added to the set of accesses in the summary of `main`. When the analysis of `main` is finished, the set of accesses from the summary of main is used for the next phase, computing the data races and reporting potential issues.

3.3 Phase 2: Data Races Computation

The second phase of the analysis involves detecting data races between accesses obtained in Phase 1. Once all functions have been analysed, all accesses that occurred in the program are contained in the summary of the top-level function. Only these accesses are considered for data race computation, and the summaries of the other functions are no longer needed.

The algorithm starts by creating pairs of accesses, which are created as a Cartesian product of the set of accesses that occurred in the analysed program. To remove redundant pairs, this set is then filtered to include only those accesses where the location of the first access is less than or equal to the location of the second access. Next, it is evaluated whether each pair satisfies the conditions for a potential data race. A pair is removed from the set if at least one condition from the following is met:

1. *Different memory locations* are accessed.
2. In both accesses, the variable is *read*.
3. There is at least one lock in the *intersection of locks* held during the accesses.
4. Both accesses are on the *same thread*.
5. *Thread-local* memory is accessed.

The pairs of accesses that are left in the set after filtering represent possible data races. In order to improve computation speed and reduce the size of the set of pairs, accesses that refer to the same memory location can be merged before creating the set. The merged accesses are used to create smaller sets, one set for each variable, and then the filtering process described earlier is applied to each of these sets.

When a single access to a shared variable in the program is unsynchronized, but all other accesses are protected by a lock properly, the filtering process may still result in a large set of pairs of accesses that can create a data race. To address this issue, we decided to report only one pair of accesses for each variable, ideally the most probable to contain a data race, to ensure that the reports are manageable.

Chapter 4

Implementation

This chapter presents the implementation of *DarC*, *a static data race checker* proposed in Chapter 3. This chapter begins by outlining the necessary steps to add a new analyser to the Infer framework, with a focus on *DarC*. This includes the implementation of required components and the definition of abstract interpretation aspects in the new analyser. Section 4.2 discusses the structure of function summaries and abstract state used in the analysis, along with the information about the modules implemented and utilized in the function summaries. The following section discusses the abstract transformers of our analyser, which are used to process `SIL` instructions and transform the current abstract state to the new abstract state during the analysis. The implementation of the data races check algorithm, including the reporting of detected data races, is described in Section 4.4.

The source code of Infer is available at GitHub[4], and so is the implementation of the new data race detector, *DarC*[5]. Note that all files mentioned in this chapter refer to the implementation in the DarC checker's repository. The main part of the implementation can be found in files `Darc.ml(i)` and `DarcDomain.ml(i)` in the `infer/src/concurrency` directory. Throughout this chapter, the implementation is demonstrated using listings written in *OCaml* or using pseudocodes that are similar to OCaml, which is an implementation language of Infer. It is an extension of the *Caml* language with object-oriented programming features. It is a functional language but also offers a variety of imperative features.

4.1 Integration of the DarC Plugin with Infer

When extending Infer with a new analyser, the analyser must be first registered in a file `infer/src/backend/registerCheckers.ml`. Three main components need to be implemented in the framework. First, the abstract domain used by the checker must be specified, which includes the definition of the type of abstract states (`astate`) that is used in the analysis. Next, a comparator for comparing two abstract states, `<=`, must be provided, along with an implementation for joining two abstract states. Therefore, the `join` and `widen` procedures need to be implemented. The abstract domain should also have the pretty print (`pp`) function defined, which prints the abstract state of the domain.

[4]Infer repository on GitHub: https://github.com/facebook/infer
[5]Data race detector on GitHub: https://github.com/svobodovaLucie/infer/tree/darc

The next component, that needs to be defined is the type of the function `summary`, which has to be added to the `Payloads.t` type in the file `infer/src/backend/Payloads.ml`. The summary is then used for the interprocedural analysis to integrate summaries of lower-level functions into higher-level functions. The type of function summaries in DarC is described in the following section. Both the abstract domain and function summary are implemented in the files `DarcDomain.ml(i)`.

The last component required by the framework is the collection of abstract state transformers. The signature for these transformers is provided in the `TransferFunctions` module implemented in `infer/src/absint/TransferFunctions.ml`, but the definition itself must be provided in the analyser. This module primarily contains the `exec_instr` procedure that takes as input the current abstract state and the `SIL` instruction and computes the new abstract state using the provided transformers. The transfer functions implemented for our checker are described in Section 4.3, and the implementation can be found in the source files `Darc.ml(i)`.

4.2 The Abstract Domain and Function Summaries

The abstract state (`astate`) used in the DarC's abstract domain is implemented as a record type, containing several fields as shown in Listing 14. In order to perform interprocedural analysis, a function summary is computed for every analysed function. For our analysis, the type for the function summary is defined as the same record type as for the abstract state. The type definitions for the abstract state and function summary can be found in the `DarcDomain.ml` file.

```
1  type t =
2  {
3    threads_active: ThreadSet.t;
4    threads_joined: ThreadSet.t;
5    accesses: AccessSet.t;
6    lockset: Lockset.t;
7    unlockset: Lockset.t;
8    points_to: PointsToSet.t;
9    heap_points_to: HeapPointsToSet.t;
10   load_aliases: LoadAliasesSet.t;
11   locals: LocalsSet.t;
12 }
13
14 type summary = t
```

Listing 14: Definition of the abstract state and summary types in the DarC checker.

The fields of the DarC's abstract state record have the types shown in Listing 14. The types of the sets of active and joined threads, accesses, points-to and heap-points-to relations, the set of load aliases, which will be described below, and the set of local variables are defined in modules `ThreadSet`, `AccessSet`, `PointsToSet`, `HeapPointsToSet`, `LoadAliasesSet` and `LocalsSet`, which are a part of the implementation of DarC. These modules will be described in the following subsections. Subsection 4.2.8 discusses the implementation of abstract interpretation operators: `join`, `widen`, and `<=` (`leq`).

4.2.1 The `ThreadSet` Module

The `ThreadSet` module represents a finite set of threads. The type of its elements is `ThreadEvent.t` shown in Listing 15. In our implementation, each thread is represented as a tuple containing an access expression, which is the abstract thread identifier of the newly created thread, a location in the code where the thread was created, and a Boolean flag indicating whether the thread was created in a loop. The `HilExp.AccessExpression.t` type is provided by Infer, and DarC uses it for storing and handling access expressions that represent variables in the analysed source code.

```
1 ThreadEvent.t = (HilExp.AccessExpression.t * Location.t * Bool.t)
2 module ThreadSet = AbstractDomain.FiniteSet(ThreadEvent)
```

Listing 15: Definition of the type `t` of `ThreadEvent` module and creation of the `ThreadSet` module using `AbstractDomain.FiniteSet` for representing a finite set of `ThreadEvent`.

When a thread is first created, the `create_in_loop` flag is set to false. However, if a thread should be added to the current `threads_active` set during the handling of the `pthread_create` function, and it already exists in the set, the flag is changed to true and the thread is added once more. This approach resolves the issue of detecting multiple thread creations within the same loop when the corresponding source code instruction is on the same line, as shown in Listing 16, which could result in the inability to detect that more than one thread was created.

```
1 for (...) {
2    pthread_create(&t, NULL, foo, NULL);
3 }
4 // threads_active: {
5 //   (t, line 2, created_in_loop=false),
6 //   (t, line 2, created_in_loop=true)
7 // }
```

Listing 16: Example code showing the use of the `created_in_loop` flag when creating threads in a loop. Without using this flag only one thread would be considered to have been created.

4.2.2 The `DeadlockDomain.Lockset` Module

The `DeadlockDomain.Lockset` module deals with locks and was developed as part of the *L2D2: Modular Low-Level Deadlock Detector*, in the bachelor's thesis of Vladimír Marcin [16]. The original implementation of this detector is available on Gitlab[6]. The implementation of the L2D2 plugin was also included in the DarC's repository as part of our previous work on this plugin [10]. The `Lockset` module represents each lock as a `LockEvent`, which is a tuple of the lock's access path and the location where the lock was created. The `Lockset` module is implemented as a finite set of these lock events. The functions for acquiring and releasing locks in our checker were inspired by the thesis and will be discussed further in Section 4.3.

[6] L2D2: https://pajda.fit.vutbr.cz/xmarci10/fbinfer_concurrency

4.2.3 The `ReadWriteModels` Module

The module ReadWriteModels contains the type definition `t` shown in Listing 17, which represents two possible access types: read or write. The module also provides a comparison function to compare two access types, with `Write` taking precedence over `Read`.

```
1 ReadWriteModels.t =
2   | Read
3   | Write
```

Listing 17: Definition of the `ReadWriteModels.t` type representing read or write accesses.

4.2.4 The `AccessSet` Module

The `AccessEvent` module defines the type `t` to represent an access to a variable, which includes the variable `var` being accessed, the location `loc` where the access occurs, the type of the access (`access_type`: `Read` or `Write`), the set of locks held during the access (`locked`), the set of locks that were unlocked before the access (`unlocked`), the set of active threads (`threads_active`) at the time of access, the set of threads joined before the access (`threads_joined`), and the identifier of the thread to indicate the thread on which the access occurs (`thread`). The module also provides a function to compare two access events and predicate functions corresponding to the conditions underlying a data race (described in Section 3.3) that are used to filter out pairs of accesses when computing data races at the end of the analysis.

The `AccessSet` module defines the type `t` to represent a set of access events on shared variables and provides functions to add or remove access events from the set, check if an access event is in the set, and iterate over the set of access events.

The type of `AccessEvent` and the definition of the `AccessSet` module are shown in Listing 18. The process of adding new accesses to the abstract state and making subsequent modifications to them will be discussed in Section 4.3.

```
1  AccessEvent.t =
2  {
3      var: HilExp.AccessExpression.t;
4      loc: Location.t;
5      access_type: ReadWriteModels.t;
6      locked: Lockset.t;
7      unlocked: Lockset.t;
8      threads_active: ThreadSet.t;
9      threads_joined: ThreadSet.t;
10     thread: ThreadEvent.t option;
11 }
12
13 module AccessSet = AbstractDomain.FiniteSet(AccessEvent)
```

Listing 18: Definition of the `AccessEvent` type which represents one access to a variable, containing all the necessary information for data race detection. On the last line, the `AccessSet` module is created as a set of `AccessEvent.t` elements.

4.2.5 `PointsToSet` and `HeapPointsToSet` Modules

The types of the `PointsToSet` and `HeapPointsToSet` modules represent the points-to relations used in our points-to analysis. These relations are stored in separate sets due to differences in their implementation types. The points-to set that represents the relation between variables allocated on the stack is represented by a pair of access expressions that indicate the variable names. On the other hand, the memory allocated on the heap is represented by a pair of an access expression for the pointer variable and a `Location.t` type that indicates the location where the memory was allocated in the source code. Despite these differences, both sets are used in a similar way. The creation of both modules and the types of their elements are shown in Listing 19.

```
1 PointsTo.t = (HilExp.AccessExpression.t * HilExp.AccessExpression.t)
2 module PointsToSet = AbstractDomain.FiniteSet(PointsTo)
3
4 HeapPointsTo.t = (HilExp.AccessExpression.t * Location.t)
5 module HeapPointsToSet = AbstractDomain.FiniteSet(HeapPointsTo)
```

Listing 19: Definition of the `PointsToSet` and `HeapPointsToSet` modules, finite sets of points-to relations created using the `AbstractDomain.FiniteSet` module, which uses the `PointsTo.t` and `HeapPointsTo.t` types as their elements, respectively.

4.2.6 The `LoadAliasesSet` Module

The `LoadAliasesSet` module is used during the analysis to handle the `LOAD` instruction, which is one of the four instructions of the Smallfoot Intermediate Language (`SIL`) used by Infer. The `LOAD` instruction loads a value from a memory address denoted by an expression, which can be a program variable or a more complex expression like array indexing or a structure field. The loaded value is then stored into a temporary identifier, which is represented as the first member of the tuple of the `LoadAlias.t` type shown in Listing 20. The second element represents the program variable that is stored in the temporary identifier. More details on how `LOAD` instructions are processed and how the load aliases are handled in the analysis will be provided in the following section.

```
1 LoadAlias.t = (HilExp.AccessExpression.t * HilExp.AccessExpression.t)
2 module LoadAliasesSet = AbstractDomain.FiniteSet(LoadAlias)
```

Listing 20: Definition of the `LoadAliasesSet` set, which is used to store information about which temporary identifier created by the `LOAD` instruction corresponds to which variable in the program.

4.2.7 The Locals Module

The `Locals` module is a set of variables that are local to the function currently analysed. Each member of this set is of the type `HilExp.AccessExpression.t`, which represents the program variables. At the beginning of the function analysis, a list of local variables that is provided by Infer is added to the empty summary. This list may be updated later when the `pthread_create` function is called with a local variable passed to the callback function

as an argument, which will be further explained in the following section. We assume that from this point the variable is shared and therefore removed from the set of local variables. During the analysis of a function, accesses to variables that are local are not added to the set of accesses, which helps to reduce the size of the set of accesses and improve scalability.

4.2.8 Abstract Interpretation Operators

The Infer.AI framework requires the analysis to implement the abstract interpretation operators join, widen, and less-that-or-equal. These operators are used to manipulate the abstract state during program analysis, especially after branching or when handling loops.

The `<=` operator compares two abstract states, `astate1` and `astate2`, and returns true if `astate1` is less than or equal to `astate2`. The comparison is performed by comparing each component of the abstract state using the following rules:

1. Known threads are compared based on the identity of the tuples that represent them.
2. An unknown thread is considered smaller than any known thread.
3. Access types are compared such that $(\mathtt{r},\mathtt{r}) <= (\mathtt{r},\mathtt{w})/(\mathtt{w},\mathtt{r}) <= (\mathtt{w},\mathtt{w})$ where `w` denotes a write and `r` denotes a read.
4. Variables, locations, and locks are compared based on their identity.
5. Accesses, which are tuples of the above entities, are compared on a per-component basis, applying the rules described above.
6. Sets of threads, locks, points-to relations, and aliases are compared on inclusion. The set of threads always consists of threads that are known, as threads are added to the set only when they are created or joined and are therefore identifiable.
7. Abstract states, i.e., tuples of the above objects, are compared on a per-component basis.

The `join` operator is shown in Listing 21. Joining two abstract states by the `join` operator consists of joining each member of the abstract states independently and then putting them all together to form the new abstract state. The join of locked locks is implemented as their intersection because, after a branching, we need to have only those locks that must be locked for sure in the lockset as described in Section 3.2. The same principle applies to the set of joined threads. For the other fields of the abstract state, `join` is implemented as their union. The `widen` operator is simply implemented as the join of two abstract states.

```
let join astate1 astate2 =
  threads_active := astate1.threads_active ∪ astate2.threads_active;
  threads_joined := astate1.threads_joined ∩ astate2.threads_joined;
  accesses := astate1.accesses ∪ astate2.accesses;
  lockset := astate1.lockset ∩ astate2.lockset;
  unlockset := astate1.unlockset ∪ astate2.unlockset;
  points_to := astate1.points_to ∪ astate2.points_to;
  heap_points_to := astate1.heap_points_to ∪ astate2.heap_points_to;
  load_aliases := astate1.load_aliases ∪ astate2.load_aliases;
  locals := astate1.locals ∪ astate2.locals;
  in (threads_active, threads_joined, accesses, lockset, ..., locals)
```

Listing 21: The implementation of the `join` operator used in the DarC checker. The operator combines two abstract states (`astate1` and `astate2`) at the end of a branching.

4.3 Transfer Functions

During the analysis, the abstract state is transformed using the `TransferFunctions` module, which defines how individual `SIL` instructions, including `LOAD`, `STORE`, `CALL`, and `PRUNE`, transform the abstract state. To implement this transformation, we define the `exec_instr` function in the `TransferFunctions` module, which takes the current abstract state and a `SIL` instruction as input, and produces a new abstract state as output.

The transformers for the `LOAD`, `STORE`, and `CALL` instructions will be described in the following subsections. When a `PRUNE` instruction is encountered, which splits the control flow of the program into two branches, the abstract state in both branches is transformed in the same way as if there was no branching, and the resulting abstract states are joined using the join operator.

4.3.1 The `LOAD` Instruction Transformer

The `LOAD` instruction loads a value from an address denoted by an expression `e` into a temporary identifier `id`. The expression `e` can be either a program variable or a more complex expression, which may include array indexes or other temporary identifiers created by previous `LOAD` instructions. The latter case happens, for example, when dereferencing a pointer or accessing a structure field. The temporary identifier `id` is then used in other instructions, such as `STORE` and `CALL`. However, the original program variables are not available at this point, which makes it difficult to determine which temporary identifier corresponds to which program variable.

To address this issue, we store a set of load aliases during the analysis, where a load alias is a pair (id, e), where id is the temporary identifier generated by Infer and e is the address of the corresponding program variable. When a `STORE` instruction is processed, the right-hand side expression is first loaded by the `LOAD` instruction into a temporary identifier, and then the load alias for that identifier is used to determine which variable is being written to. Similarly, in the case of a `CALL` instruction, the parameters passed to the function are first loaded into temporary identifiers, which are then used to determine the program variables passed to the function. In the case of a `LOAD` instruction, if a program variable is read, an access to the corresponding program variable is added to the set of accesses in the current abstract state.

To illustrate how the `LOAD` instruction works and how the corresponding load aliases are created, consider the following examples:

- `int i = j`: The `LOAD` instruction loads the address of `j` into a temporary identifier `n$1`, and the load alias `(n$1,j)` is created and added to the `load_aliases` set. Since this expression is accessing the variable `j`, a new access of the `ReadWriteModels.Read` type is added to the current abstract state. The assignment to the variable `i` is then processed by the `STORE` instruction, which will be described in Subsection 4.3.2.
- `int i = j + k`: In this example, three `LOAD` instructions are generated. The first `LOAD` instruction loads the address of `j` into the temporary identifier `n$1`, and the load alias `(n$1,j)` is created. The second `LOAD` instruction loads the address of `k` into the temporary identifier `n$2`, and the load alias `(n$2,k)` is created. The third `LOAD` instruction loads the expression `n$1+n$2` into the temporary identifier `n$3`, and

the alias `(n$3,n$1+n$2)` is created. Since this is a complex expression and does not correspond to only one program variable, new read accesses are created and added to the set of accesses only when the first and the second `LOAD` instructions are processed. The assignment of `n$3` to the variable `i` is handled later when processing `STORE`.

- `int i = *p`: This example illustrates a case where a temporary identifier is loaded from another temporary identifier. The first `LOAD` instruction loads the address of `p` into a temporary identifier `n$1`, creating the load alias `(n$1,p)`. The second `LOAD` instruction then dereferences the address in `n$1` and loads the value into a new temporary identifier `n$2`. Initially, a load alias `(n$2,n$1)` would be created, but since the temporary identifier `n$1` has a corresponding program variable, we obtain its load alias, which indicates that `p` is loaded to `n$1`. Since `n$2` now holds the address pointed to by `p`, we add a dereference to the load alias of `n$1` (i.e., the alias of `p`), resulting in a new load alias `(n$2,*p)`. This alias is then added to the set of load aliases. When processing the `LOAD` instructions in this example, two accesses of the read type are added – one for accessing the pointer variable `p`, and another for accessing the address pointed to by `p`, that is, `*p`.
- `int i = s.v`: When a structure field is accessed, such as in this case, there are two `LOAD` instructions generated. The first instruction loads the address of the structure `s` into a temporary identifier `n$1`, and a load alias `(n$1,s)` is created. The second instruction loads the address of the field `n$1.v` into a temporary identifier `n$2`, which would create an alias `(n$2,n$1.v)`. Since the value of `n$1` is already present in the set of load aliases, it is replaced in this alias, and the final load alias created is `(n$2,s.v)`. Two read accesses are created when processing these instructions: one for the structure `s`, and the second one for the structure field `s.v`.
- `int i = arr[x]`: When accessing an array element, there are two `LOAD` instructions generated. The first instruction loads the address of the index `x` into a temporary identifier `n$1`, and a load alias `(n$1,x)` is created. An access to the variable `x` of the read type is added to the set of accesses. The second instruction loads the address of the offset of the array element into a temporary identifier `n$2`, which would create the alias `(n$2,arr[n$1])`. Since we do not differentiate between the particular index values in the array, the final read access will be added to the `arr[_]` access expression.

All accesses created when processing the `LOAD` instructions in the **load** instruction transformer are of the `ReadWriteModels.Read` type and represent the current abstract state of the program when the access occurs. As we have already said, these accesses store information about the current set of locks that must be locked, the set of locks that may be unlocked, the set of threads that may be currently running, and the set of threads that must be joined. If the access occurs within the `main` function, the `access.thread` field is assigned to `main_thread`; otherwise, the thread field is set to `None`, indicating that the thread is not yet known.

4.3.2 The `STORE` Instruction Transformer

The **store** transformer is responsible for handling `STORE` instructions in the analysed control flow graph. The transformer takes as an input the expression being stored (`e2`) and an address expression `e1` into which `e2` is stored, along with the information about its type. The right-hand side expression is usually created by the previous `LOAD` instructions and is

represented as a temporary identifier. The left-hand side expression may be a temporary identifier, e.g., in the case of pointer dereferencing, or a program variable.

After resolving the temporary identifier on the left-hand side of an assignment expression, the write access is added to the set of accesses for the corresponding variable as well as to all variables that may alias with it. Similarly to adding a new access in the **load** transformer, the write access contains information about the current abstract state, such as the sets of locked and unlocked locks, and the sets of active and joined threads. Additionally, information about the thread is added based on whether the access occurs in `main` or if the thread is unknown, in which case `None` is used.

If the type of the expression on the left-hand side is a pointer type, the write access may also lead to the editing of the sets of points-to and heap-points-to relations. Specifically, the points-to relation for the left-hand side variable is updated to point to the address that is being written to. If there are any existing relations with the left-hand side variable, they are removed from the points-to set to indicate that the variable now points to the new address. In a similar way to how the points-to relation is updated, if the left-hand side of an expression involves a heap-allocated variable, the heap-points-to relation may also be updated to reflect any changes.

Consider the following examples to illustrate how the **store** transformer works. We start with the case where the points-to information is precise, i.e., we do not have more possible targets of a single pointer.

- `int i = j`: In this assignment, the `STORE` instruction in the form of `STORE n$1 to &i` is generated where `n$1` is a temporary identifier generated by the previous `LOAD` instruction. The set of load aliases contains the pair `(n$1,j)`, therefore we know that `j` is assigned to `i`. However, since the type of the expression is not a pointer type, no points-to relations are created. Only a write access to the variable `i` is added to the set of accesses when this instruction is processed as the read access of `j` has already been added when processing the `LOAD` instruction.
- `int *p = &i`: This example represents the case when the right-hand side expression is not represented by a temporary identifier but rather directly as `&i`. That is because the `LOAD` instruction does not process the address operator. Therefore, the `STORE` instruction generated in this case stores the address of `i` into `p`. First, the write access to `p` is added to the set of accesses. Since this expression has a pointer type, the set of points-to relations is updated. The points-to relation `(p,&i)` is created and added to the set of points-to relations. If there is already an existing points-to relation with the variable `p` on the left side, it is removed from the set of points-to relations. This indicates that `p` now points to the address of the variable `i`.
- `*p = 0`: The `STORE` instruction generated for this expression is in the form of `STORE Const 0 into n$1` where `n$1` is a temporary identifier generated by the previous `LOAD` instruction, and thus, there is an alias `(n$1,p)` in the set of load aliases. Since there is a dereference on the left-hand side, the location where the variable `p` points to needs to be determined. This information is stored in the points-to set, which contains the relation `(p,&i)` assuming that the above expression has been extended previously. Consequently, the variable `i` is retrieved, and the write access is added with the information that the variable accessed is `i`. Since this expression is not of the pointer type, no points-to relations are updated.

- `int *q = p`: This assignment statement is first processed by the **load** transformer, which generates the load alias `(n$1,p)` and stores it in the set of load aliases. The `STORE` instruction then stores the temporary identifier `n$1` to the variable `q`. The write access to the variable `q` is added to the set of accesses. Since `q` is a pointer-type variable, the points-to relations set is also updated. To determine where `p` points to, the points-to relation for `p` is obtained, which is `(p,&i)` assuming that the statement `p=&i` described above has been executed. This information is then used to create a new points-to relation `(q,&i)` in the points-to set. This new relation indicates that `q` now points to the same address as `p`, which is the address of the variable `i`.

These examples illustrate how the `STORE` instruction is processed and how the points-to relations set is updated. The same principle applies when adding new heap-points-to relations when dynamically allocating memory using functions such as `malloc`. The only difference is that the new relation is added to the set of heap-points-to relations.

To illustrate a situation where one variable may point to multiple addresses in our analysis, let us examine a more complex example in Listing 22. In this example, each branch of the conditional statement is processed separately. In the if branch, we follow the same logic as described above and add the points-to relation `(p,&i)` to the points-to relations set, indicating that the variable `p` points to the address of the variable `i`. In the else branch, a dynamic memory allocation is performed using the `malloc` function, and the heap-points-to relation `(p,line 5)` is added to the heap-points-to relations set, indicating that the variable `p` points to a newly allocated memory block on line 5.

```
1 int i, *p;
2 if (x) {
3   p = &i;                    // points-to: {(p, &i)}
4 } else {
5   p = malloc(sizeof(int));   // points-to: {(p, line 5)}
6 }
7 // points-to after join: {(p, &i), (p, line 5)}
8 *p = 0;
```

Listing 22: Example code demonstrating a scenario where a variable can have multiple points-to relations during the analysis. The points-to set contains two relations, `(p, &i)` and `(p, line 5)`, after the conditional branching. Upon accessing the address pointed to by `p` on line 7, a write access is conservatively added to both variable `i` and the access expression `*p` for the dynamically allocated memory block allocated on line 5.

After the branching, the abstract states from both branches are joined together into a single abstract state. As a result, the variable `p` now has two points-to relations, `(p,&i)` and `(p,line 5)`. When dereferencing `p`, it is unclear whether the write access is to the variable `i` or to the dynamically allocated memory block on line 5. Therefore, both options must be considered in our analysis, and a write access must be added to both locations. The sets of points-to and heap-points-to relations are used to identify these locations. Consequently, the new accesses that will be added to the current set of accesses will include `(i, write)` for the variable `i`, and `(*p, write)` for the dynamically allocated memory block referenced by the pointer `p`.

The points-to analysis used in our analysis performs a weak update when updating points-to relations for a pointer that may point to multiple addresses. This means that if we have

a pointer to pointer, where the pointer on top has more points-to relations in the set, and we perform a double dereference of this pointer and assign it a new address, the original points-to relations will not be removed from the set. Instead, only new relations will be added, resulting in a set of points-to relations that includes both the original relations and the new ones. However, if a pointer has only one points-to relation in the set, updating that relation will result in a strong update, where the original relation is replaced by the new one.

The functions responsible for updating aliases and resolving them to get a list of all aliases are implemented in the `infer/src/concurrency/DarcDomain.ml` file. Specifically, the functions `update_aliases` and `resolve_entire_aliasing_of_var` are used to update the points-to and heap-points-to relations sets and to obtain a list of all the aliases of a given variable, respectively.

4.3.3 `CALL` Instruction Transformers

When a function is called in the analysed source code, a `CALL` instruction is generated in the control flow graph. This instruction provides information about the called function, including the name of the function, the list of actual arguments passed to the function, and the temporary identifier representing the return expression of the function. It is important to note that indirect function calls are handled through a combination of `LOAD` and `CALL` instructions.

Certain function calls have a significant role in computing data races and need to be handled individually. These include functions for creating and joining threads, locking and unlocking locks, and handling dynamic memory allocation. The transformers used for handling these function calls are implemented in the `Darc.ml` and `DarcDomain.ml` files, and they are listed below.

The acquire_lock transformer. This transformer is responsible for handling the function `pthread_mutex_lock` that locks a particular lock. The transformer first resolves the name of the lock being acquired as the lock may be represented by a temporary identifier in the program, which must be changed to its corresponding variable. Next, the transformer adds the lock to the current `lockset` as described in Section 3.2.2. If the lock is present in `unlockset`, it is removed.

The release_lock transformer. This transformer is called upon every lock release in the analysed source code. It works similarly to the acquire_lock transformer as it first resolves the name of the lock being released by `pthread_mutex_unlock` and then this lock is removed from the current `lockset` and added to the `unlockset`.

The handle_malloc transformer. This transformer is responsible for handling the dynamic memory allocation, which is done by the `malloc`, `calloc`, and C++ `__new` function calls. The process involves two steps: the first step is executed when the `CALL` instruction is processed, and the second step is executed when the `STORE` instruction is processed. To help illustrate this process, consider as an example the following dynamic allocation assuming that it appears on line 2 of some code: `int *x = malloc(sizeof(int))`.

The first part of the handling is done when processing the `malloc`, `calloc`, or `__new` function call itself by the `CALL` instruction. However, the transformer does not have access to information about the variable to which the return value of the allocation function is stored, which is the variable `x` in our example, as this information is provided by the `STORE` instruction later. For this reason, the `analysis_data.extras.heap_tmp` list is kept as part of the analysis state to store the temporary identifier of the return expression along with the information about the line of code where the memory was allocated.

In the above example, the result of the `malloc` function call, which is the return expression in the `CALL` instruction, is stored into a temporary identifier `n$1` generated by Infer. Therefore we know that the address of the memory dynamically allocated on line 2 is stored in the expression `n$1`, and the relation `(n$1, line 2)` is added to the `heap_tmp` list.

In the second part of handling dynamic memory allocations, the `STORE` instruction is processed. If the `analysis_data.extras.heap_tmp` list is not empty, it indicates that there are dynamically allocated variables to be added to the `heap_points_to` set. To achieve this, the `handle_store_after_malloc` procedure is called, which resolves the temporary identifier from the `heap_tmp` list, along with the information about the location where the variable was allocated.

Continuing with the example, when the `STORE` instruction is processed, the relation `(n$1, line 2)` from the `heap_tmp` list is retrieved since it associates the temporary identifier `n$1` with the actual program variable `x`. Finally, the identifier `n$1` is replaced by the actual program variable `x` in the relation, and the newly created relation `(x, line 2)` is added to the set of heap-points-to relations in the `heap_points_to` set.

The integrate_summary transformer. This transformer ensures that the analysis is interprocedural by applying a summary of a called function (callee) to the current abstract state when the callee is called, which involves multiple steps.

In the first step, the accesses from the callee summary are modified using the current abstract state, namely the currently locked and unlocked locks, and the active and joined threads. The process of modifying the accesses was discussed in Subsection 3.2.5 and is shown in Listing 23.

Second, the thread to be updated in accesses in the callee summary is determined based on whether the currently analysed function is the `main` function or not. If the function is `main`, the `main_thread` is added to the accesses, otherwise, the information about the thread is not updated. It is worth noting that if an access already has information about the thread it is running on, it will not be updated to avoid deleting information about accesses on other threads.

Third, *formal* parameters of the callee function are replaced with the *actual* parameters. For each actual parameter, its aliases are found using the set of points-to relations, and all accesses from the callee summary that have the corresponding formal parameter as a variable are updated with the information about the actual parameter and added to the current set of accesses. If an actual parameter has multiple aliases, accesses to all of those aliases will be added as well. Accesses from the callee summary that are to global variables are also added to the current abstract state.

```
let update_access_with_locks_and_threads access astate current_thread =
  access.locked := (astate.lockset ∪ access.locked) \ access.unlocked;
  access.unlocked := (astate.unlockset \ access.locked) ∪ access.unlocked;
  access.threads_active := (astate.threads_active ∪ access.threads_active)
                           \ access.threads_joined;
  acces.threads_joined := astate.threads_joined ∪ access.threads_joined;
  access.thread :=
    match access.thread with
    | None -> current_thread (* access.thread is set to current_thread *)
    | Some th -> Some th     (* access.thread remains unchanged *)
```

Listing 23: Simplified implementation of the `update_access_with_locks_and_threads` function for updating `access` in the summary of the called function with the current abstract state (`astate`), and with the currently running thread (`current_thread`). The `current_thread` is set to `main_thread` if the function that is currently analysed is `main`. The `update_access_with_locks_and_threads` function is applied to each access in the summary of the called function.

Finally, the current abstract state is updated with the information about the lockset, unlockset, active threads, and joined threads from the callee summary, as shown in Listing 24. After all of these transformations are performed, the updated abstract state is returned.

```
let integrate_summary_without_accesses a c =
 a.threads_active := (a.threads_active ∪ c.threads_active) \ c.threads_joined;
 a.threads_joined := (a.threads_joined \ c.threads_active) ∪ c.threads_joined;
 a.lockset := (a.lockset ∪ c.lockset) \ c.unlockset;
 a.unlockset := (a.unlockset \ c.lockset) ∪ c.unlockset;
 a.points_to := a.points_to ∪ (remove_locals c.points_to);
 a.heap_points_to := a.heap_points_to ∪ (remove_locals c.heap_points_to);
```

Listing 24: The implementation of the function that joins the `lockset`, `unlockset`, `threads_active`, and `threads_joined` sets of the current abstract state (`a`) and the summary of the called function (`c`).

The integrate_pthread_summary transformer. This transformer works similarly to the `integrate_summary` transformer, but it is called specifically when a `pthread_create` function is called in the analysed program. In this case, the callee summary is obtained by calling the `analyse_dependency` function on the callback function provided as the third argument to `pthread_create`. If the callback function has not yet been analysed, it is analysed on demand, and its summary is returned as `callee_summary`. Then, the newly created thread specified in the first parameter of `pthread_create` is added to the `threads_active` set. All accesses from the callee summary are updated with the currently created thread as well as with the current `lockset`, `unlockset`, `threads_active`, and `threads_joined` sets. The fourth argument of the `pthread_create` function is used to pass a variable to the callback function, and this variable replaces the formal parameter in the accesses of the callee summary. The process of replacing the formal parameter is the same as in the `integrate_summary` transformer. Similarly, information about the current sets of locked locks, unlocked locks, active threads, and joined threads is also updated in the same way as in the `integrate_summary` transformer.

The handle_pthread_join transformer. When a function call to `pthread_join` is detected, this transformer is invoked. It removes the thread that is passed as the first argument to this function from the set of currently running threads, `threads_active`. The thread is then added to the set of joined threads as described in Subsection 3.2.1.

4.4 Data Race Detection and Reporting

After the summaries of all functions used in the program are computed, the function `compute_data_races` is called, and the detection of data races begins. Since the `main` function is the top-level function and the functions in the bottom-up analysis are processed from the leaves to the top, by the time the `main` function is analysed, all the functions called by the top-level function have also been processed. As a result, the summary of the `main` function contains *all the accesses* that occurred in the analysed program on shared variables. The algorithm uses only the set from the summary of `main` to compute the data races.

The `compute_data_races` function starts the computation by creating pairs of accesses for each variable that is present in the set of derived accesses. Next, the pairs are filtered using predicate functions implemented in the module `AccessEvent` (one for each of the below mentioned conditions) such that only the pairs $(\mathtt{a}, \mathtt{a}')$ that meet the conditions discussed already in Section 3.3 are retained. To recall, the conditions being checked are as follows:

1. $\mathtt{a.var} = \mathtt{a}'.\mathtt{var}$,
2. $\mathtt{a.access\_type} = \mathtt{Write} \vee \mathtt{a}'.\mathtt{access\_type} = \mathtt{Write}$,
3. $\mathtt{a.thread} \neq \mathtt{a}'.\mathtt{thread}$,
4. $\mathtt{a.lockset} \cap \mathtt{a}'.\mathtt{lockset} = \varnothing$,
5. $\mathtt{a.thread} \in (\mathtt{a.threads\_active} \cap \mathtt{a}'.\mathtt{threads\_active})$
   $\vee \mathtt{a}'.\mathtt{thread} \in (\mathtt{a.threads\_active} \cap \mathtt{a}'.\mathtt{threads\_active})$.

The result of the filtering is a list of all pairs of accesses on which a data race may occur, from which the first pair for each variable is returned. The final step of the analysis is to report the results. Only one data race for each variable is reported to avoid overwhelming developers during report examination. For reporting, the `Reporting` module provided by Infer is used. Its implementation can be found in the files `Reporting.ml(i)`. The issue type reported by each checker must be registered in the `IssueType.ml(i)` module along with the information about the kind of the issue reported, such as *Like*, *Info*, *Advice*, *Warning*, or *Error*. The *Error* type has been chosen for issues reported by *DarC*. The report generated for a program with one discovered data race could look as shown in Listing 25.

```
race-example.c:15:  error:  DarC Checker
  Data race between:  write to i on line 15 on thread t created on line 39,
                      read from i on line 17 on the main thread.
Found 1 issue
  Issue Type(ISSUED_TYPE_ID): #
  DarC Checker(DARC_CHECKER): 1
```

Listing 25: Report generated by DarC checker when analysing a simple program that contains a data race on a shared variable `i`.

Chapter 5

Experimental Evaluation

This chapter presents an experimental evaluation of our new data race detector, *DarC*. The main goal of this chapter is to assess the effectiveness of DarC in detecting data races in various types of codebases. The first section focuses on small and simple programs that mostly contain data races. We report on the number of data races detected by DarC and compare our results to other data race analysers. We also report on the number of data races and both true and false positives reported by DarC in the benchmark used in the division of concurrent programs of the International Competition on Software Verification, SV-COMP [32]. The second section presents the results of our experiments on several real-world projects. These projects include several commonly used utilities, such as *sort*, *grep*, and *memcached*, as well as the *eProsima/Fast-DDS* communication platform. We discuss the challenges we faced in analysing these codebases and the reasons why some races may have been missed. All of the experiments were run on a machine with the AMD Ryzen 5 5500U CPU, 15 GiB of RAM, 64-bit Ubuntu 20.04.4 LTS, using Infer version v1.1.0-0e7270157.

5.1 Simple Codebases

This section describes our experimental evaluation of DarC using three different benchmarks consisting of shorter programs. Since we do not have information about real data races in the first two benchmarks, we decided to compare the results of analysing these programs using DarC with the results of one existing static analyser, *Coderrect/O2* [15], and two dynamic analysers, *ThreadSanitizer* [23] and *Helgrind* [11]. It is important to note that all of these analysis tools may report false positives, meaning that they may report data races that are not really present in the program. However, when multiple tools report the same data race, it becomes more likely that the reported data race is real.

5.1.1 DataRaceBenchmark

DataRaceBenchmark[7] is a set of C/Pthreads concurrent programs consisting mostly of programs with data races obtained from *SCTBench*[8], a C/C++ benchmark for evaluating

[7] *DataRaceBenchmark*: https://github.com/marchartung/DataRaceBenchmark
[8] *SCTBench*: https://github.com/mc-imperial/sctbench

concurrency testing techniques. We have evaluated the results for the *simple_build* benchmark, which includes 67 programs ranging in size from 38 to 257 LOC[9]. The results of our evaluation are presented in Table 5.1.

Table 5.1: Results of the DarC, Coderrect/O2, ThreadSanitizer, and Helgrind analysers on the *DataRaceBenchmark* containing 67 programs. The column *races* shows the number of programs in which data races were detected, *no races* represents the number of programs in which data races were not detected by the tool. The *timeout* column represents the number of programs for which the timeout expired, and the *time* column shows the analysis time. The timeout was set for 6 seconds, and the summary time was measured without the programs that timed out.

analyser	races	no races	timeout	time
DarC	**40**	**28**	**0**	**24.4s**
Coderrect/O2	29	39	0	1m26.2s
ThreadSanitizer	40	25	3	14.1s
Helgrind	40	25	3	25.6s

Upon analysis, DarC reported data races in 40 programs. Among these programs, 39 were the same programs that were identified by both dynamic analysers as containing data races. However, for one program, `thread-pool.example.c`, DarC did not identify any data races while ThreadSanitizer and Helgrind did. As of yet, we have not been able to determine why DarC failed to detect this data race, assuming it is indeed a real data race since it was detected by both dynamic analysers.

On the other hand, DarC identified one program, `twostage_bad.c`, as containing a data race, while the dynamic analysers did not. This program is dependent on the command-line arguments that are specified when running it. When no command-line arguments are provided, no parallel threads are created during program execution, and as a result, the dynamic analysers do not detect any data race. However, if a command-line argument is specified, indicating that more than one thread will be created, then both dynamic analysers detect data races. In this case, DarC reported a data race as expected.

The fact that all programs detected by DarC as containing data races were also reported by both dynamic analysers indicates that there is a high probability that these are genuine data races. Moreover, the analysis time for DarC was similar to the times for dynamic analysers. In contrast, the number of data races reported by O2/Coderrect was significantly lower than that reported by DarC, and the analysis time was more than three times longer.

5.1.2 ConcurrencyBenchmark

ConcurrencyBenchmark[10] is a test suite consisting of 91 small programs that we developed ourselves for testing Darc. These programs are written in C/Pthreads, with a size range of 38–57 LOC, and test various features that we wanted DarC to be able to handle, such as creating threads in a loop and various aliasing examples. Moreover, these programs also include cases in which we know that DarC will report false positives. We also evaluated these programs using Coderrect/O2, ThreadSanitizer, and Helgrind for comparison.

[9] Lines of Code (LOC) - a metric used to evaluate a software program according to its size.
[10] *ConcurrencyBenchmark*: https://github.com/svobodovaLucie/ConcurrencyBenchmark

The experimental results are summarized in Table 5.2. Out of the 91 programs tested, DarC reported data races in 60 of them. In 47 of these programs, data races were also reported by at least one of the dynamic analysers. Out of those, 41 programs were reported by both dynamic analysers, and 17 were reported by all four tools.

Table 5.2: Summary of the analysis results of DarC, Coderrect/O2, ThreadSanitizer, and Helgrind analysers on the *ConcurrencyBenchmark*. This benchmark consists of 91 programs. The *races* column represents the number of programs where data races were detected, *no races* column shows the number of programs where no data races were detected by the tool. The timeout was set for 6 seconds, and the *timeout* column represents the number of programs for which the timeout limit was reached. The *time* column displays the analysis time, which was measured without including the timed-out programs.

analyser	races	no races	timeout	time
DarC	**60**	**31**	**0**	**31.5s**
Coderrect/O2	28	63	0	1m46.0s
ThreadSanitizer	49	38	4	18.6s
Helgrind	44	40	7	34.3s

Out of all programs in the benchmark, only one program (`recursion.c`) that contained a real data race was reported by one dynamic analyser (ThreadSanitizer) and not by any other tool. To better illustrate the reason why DarC failed to detect the data race, consider a simplified version of the program shown in Listing 26.

```
1  int x;
2  pthread_t t1;
3  g() {
4    f();    // summary f: accesses={}
5  }
6  f() {
7    pthread_create(t1, g);  // summary g: accesses={}
8    x = 0;
9  }
10 main() {
11   f();    // summary f: accesses={(x on line 9)}
12 }
```

Listing 26: A sample program with a recursive call that contains a data race on the global variable `x`. The data race is not detected by our analyser due to the recursive call to the function `f` on line 4 in the function `g`.

In the above example, the analysis starts with the analysis of the function `f`. There is a call to the `pthread_create` function on line 7, which creates a new thread `t1` that starts executing the function `g`. Therefore, `g` is analysed on demand. Because the summary of `f` is not yet computed, an empty summary is used for the summary integration on line 4. This leads to `g` also having an empty summary. This summary is then integrated into the current abstract state of function `f` on line 7 and the set of accesses is still empty at this point. Next, on line 8, there is a write access to variable `x`, which is added to the current set of accesses. The analysis of `f` then ends. Finally, the analysis of the `main` function begins,

which is executed on the main thread. On line 13, there is a function call to `f`, and the summary of `f` is integrated, which involves adding the access to `x` on line 8 to the current set of accesses. But since our analysis only recorded this one access and no accesses on other threads were recorded, our detector did not detect any data races, even though they actually exist in the program.

In terms of false positives, there were several reasons why DarC reported them. For instance, when there are two write accesses to an array and each is on a different index. DarC does not distinguish on which index the array was accessed and only records an access to that array. As a result, DarC reports a data race on the array, which is not really present. Another example involves a program that contains a lock in a conditional branch where the condition is always true, and therefore the lock is locked in every case. However, because DarC stores locks that *must be locked*, when there is locking in only one branch of a conditional branching, the lock is not added to the *lockset*. Thus, the accesses that occur after this are not recorded as protected by a lock, resulting in DarC reporting a data race.

5.1.3 SV-COMP Benchmark

To evaluate our checker on a broader range of programs, we used a benchmark from the International Competition on Software Verification (SV-COMP) [32], which contains a large number of programs written in various programming languages and designed to check various kinds of bugs[11]. For our evaluation, we ran DarC only on C/C++ programs that were labelled to contain or not contain data races in the SV-COMP benchmark itself.

The results for the programs that we analysed are shown in Table 5.3. As the benchmark contains numerous programs, we were unable to manually go through every single program and check why DarC did not report data races or why it reported false alarms. While the results on some benchmarks are quite promising, DarC's performance on some benchmarks, such as *pthread-wmm*, was not satisfactory.

For the *pthread-wmm* benchmark, the reason why DarC reported all programs from this benchmark as false positives could be due to the fact that, instead of the Pthreads library functions, these programs extensively use the special synchronization primitives, such as `__VERIFIER_atomic_begin` and `__VERIFIER_atomic_end` macros. These primitives are not recognized by DarC, and therefore it may not be able to correctly reason about the synchronization between threads. These primitives are not commonly used in regular programs and are specific to the SV-COMP benchmark. In addition, it should be noted that many programs in the SV-COMP benchmark are highly preprocessed, making it difficult to manually inspect them and determine the presence of data races. While DarC's performance on this benchmark may not have been optimal, our main priority was not to thoroughly inspect all false positives or false negatives reported by DarC on this benchmark. It can rather be used as a source of knowledge for the further development of DarC.

[11] *SV-COMP* benchmark: https://github.com/sosy-lab/benchexec

Table 5.3: Results of DarC on the *SV-COMP* benchmarks. The column *races* shows the number of programs containing data races that were detected correctly, *false negatives* gives the number of programs in which data races were not detected. The column *no races* shows the number of programs without data race that were reported correctly by DarC, and the *false positives* column gives the number of programs in which data races were detected incorrectly. The *time* column shows the analysis time, and *count* displays the number of programs in the benchmark.

benchmark	races	false negatives	no races	false positives	time	count
pthread	7	1	30	11	15.55s	49
pthread-atomic	9	1	0	7	5.62s	17
pthread-C-DAC	1	0	3	1	1.77s	5
pthread-complex	1	0	1	3	2.08s	5
pthread-deagle	15	5	4	0	7.51s	24
pthread-divine	11	2	2	1	5.55s	16
pthread-ext	31	0	1	30	20.49s	62
pthread-lit	11	0	0	3	4.87s	14
pthread-nondet	4	2	0	0	1.99s	6
pthread-wmm	0	0	0	283	1m51.4s	283
goblint-reg	34	33	142	7	1m10.8s	216
ldv-races	3	5	7	4	6.92s	19

5.2 Real-World Projects

In this section, we report on various experiments we have performed on real-world projects. We have analysed five applications: *grep* 3.7, *tgrep* (a multi-threaded version of *find* combined with *grep* by Ron Winacott[12]), GNU Coreutils *sort* 8.32, *memcached* version 1.6.10 (a distributed memory object caching system), and *eProsima/Fast-DDS* version 2.10.1, a C++ implementation of the Data Distribution Service of the Object Management Group. The source code of all of these programs apart from *eProsima/Fast-DDS* was pre-processed by the Frama-C static analysis framework [5], and amounted to 49.3 kLOC in total. The codebase for *eProsima/Fast-DDS* is 110kLOC. The runtime for the analysis of each program was as follows: 2.1 seconds for *grep*, 0.8 seconds for *tgrep*, 1.9 seconds for *sort*, 7.7 seconds for *memcached*, and nearly 28 minutes for *eProsima/Fast-DDS*. We now discuss the results of the analysis.

For *memcached*, DarC reported 60 data race issues. Due to the complexity of the code and the large number of threads created in various conditional branches, it was not feasible to investigate all of these issues. However, upon investigating a few of them, we found that they had a similar nature. Due to the use of various conditional branching statements for creating new threads, DarC may add multiple threads to the set of active threads even though they cannot be created simultaneously during actual program execution. This results in the integration of summaries of multiple callback functions that cannot be executed simultaneously. Furthermore, DarC assumes that a thread is active even after it has been joined in a branch selected by the same condition as when the thread was created. Similarly, DarC may assume that a lock is not held if it is only locked in one branch, even though that branch might actually be executed in the real program. These issues arise because DarC

[12] *tgrep*: https://docs.oracle.com/cd/E19455-01/806-5257/6je9h034c/index.html

does not take into account the various conditions and possible program paths. As a result, many false alarms are reported for accesses that cannot lead to data races in reality.

Regarding the other programs, namely *sort*, *grep*, and *tgrep*, DarC did not report any data races. However, we further investigated these programs and found possible reasons why no alarms were reported. First, we examined the analysis report of *tgrep* and discovered that `pthread_create` wrappers, which DarC does not currently support, are used in the code. This may be one of the reasons why no errors were reported. To investigate further, we replaced the wrappers in *tgrep* with calls to the `pthread_create` function itself, which led to the detection of two data races. However, we do not know whether these are false alarms. As a sanity check, we removed one `pthread_mutex_lock` call in *tgrep*, which was used when locking a global variable, and DarC successfully reported the data race created this way.

Regarding the *sort* program, one possible reason why DarC did not report any data races is the recursive nature of the `sortlines` function, which is the only function in the program where new threads are created. The callback function to which the new threads are passed simply calls `sortlines` recursively and returns. Since the analysis of the `sortlines` function is not yet finished, its summary is not available, resulting in an empty summary being used instead. This leads to the callback function also having an empty summary, and no accesses on other threads are added to the current abstract state when analysing `sortlines`. Therefore, there can be no data races detected because all accesses that are computed in the analysis occur on the main thread.

Additionally, during our inspection of the *grep* analysis results, we discovered that memory for threads is dynamically allocated on the heap by calling the `xmalloc` function in this program, and the threads are created in a loop with a thread id computed from an expression `threadId+i`. This expression is complex since it involves a binary operation, which is not currently correctly handled by DarC. Therefore, the `integrate_pthread_summary` transformer is not called, and the summary computed for the callback function is not integrated into the current abstract state. As an experiment, we replaced the expression `threadId+i` with an array of threads and then created threads for the expression `threads[i]` instead. This modification led to DarC detecting one data race in the program, but we are not certain whether it is a false alarm or not. As a further check, we removed one `pthread_mutex_lock` call before writing to a variable `worqueue.producer_done`, where `worqueue` is a global structure, and DarC reported this introduced data race successfully.

Finally, regarding *eProsima/Fast-DDS*, DarC did not report any data races in this program, which could be due to several reasons. First, it uses C++ guard locks, which are not yet supported by DarC. Additionally, the same problems as described above in the description of *sort*, *tgrep* and *grep* could have also been present. However, due to the size of the project, we were unable to go through the code and find all the root causes, and we plan to analyse *eProsima/Fast-DDS* again after further improvement of our checker based on the issues described above.

Chapter 6

Conclusions

In this thesis, we proposed a new static data race detector, *DarC*, designed for analysing concurrent programs written in the C language using the Pthreads library. It was implemented as a plugin of the Infer framework, an open-source framework for highly scalable static program analysis. The proposed solution is based on recording a set of accesses to shared variables that occur in the analysed program followed by checking pairs of these accesses to detect possible data races. Our tool was experimentally evaluated on a set of benchmarks developed for testing concurrency bugs, as well as on real-life projects. For the benchmarks, DarC detected the vast majority of bugs detected by dynamic checkers used for comparison. On the real-life projects, we observed some shortcomings that will be addressed in future work, and additionally, we intentionally introduced data races into those programs, and DarC successfully detected all of them. These promising results demonstrate the potential of our approach to detect data races in concurrent C programs.

Future work will focus on implementing several improvements to the DarC checker. Based on the observations from our experiments, we plan to add support for `pthread_create` wrappers and correctly handle more complex expressions used as abstract thread identifiers during the creation of threads. Another crucial area of focus will be the implementation of heuristics for filtering detected data races, as we are aware of the fact that reporting only the first data race found for a variable can lead to overlooking real bugs. These heuristics will be based on various factors, such as using preconditions to detect which locks are expected to be locked and unlocked, or the number of aliases that a variable might have. Furthermore, we aim to incorporate support for recursive locks and additional locking and unlocking functions used in the program beyond those provided by the Pthread library. These functions could be specified by developers and then used in the analysis to be handled correctly as locking/unlocking functions. Another area of improvement will be to add support for analysing programs that do not have a defined main function, such as library functions, by providing a file or command line argument with entry points. Finally, after incorporating these improvements, we plan to conduct further experiments and analyse more real-life projects, including Firefox, Chromium, and the Linux kernel. These experiments will help us recognise more areas of possible improvements.

Bibliography

[1] 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. oct 2018, vol. 2, OOPSLA. DOI: 10.1145/3276514. Available at: https://doi.org/10.1145/3276514.

[2] Cousot, P. *Abstract Interpretation in a Nutshell* [online]. January 2010 [cit. 2023-01-10]. Available at: https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html.

[3] Cousot, P. and Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In:. January 1977, p. 238–252. DOI: 10.1145/512950.512973.

[4] Cousot, P. and Cousot, R. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M. and Wirsing, M., ed. *Programming Language Implementation and Logic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, p. 269–295. ISBN 978-3-540-47297-1.

[5] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J. et al. *Frama-C: A Software Analysis Perspective*. Berlin, Heidelberg: Springer-Verlag, 2012. DOI: 10.1007/978-3-642-33826-7_16.

[6] Engler, D. and Ashcraft, K. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. New York, NY, USA: Association for Computing Machinery, 2003, p. 237–252. SOSP '03. DOI: 10.1145/945445.945468. ISBN 1581137575. Available at: https://doi.org/10.1145/945445.945468.

[7] Fiedor, J., Hrubá, V., Křena, B., Letko, Z., Ur, S. et al. Advances in Noise-based Testing. *Software Testing, Verification and Reliability*. Willey. 2014, vol. 24, no. 7, p. 1–38.

[8] Flanagan, C. and Freund, S. N. FastTrack: Efficient and Precise Dynamic Race Detection. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2009, p. 121–133. PLDI '09. DOI: 10.1145/1542476.1542490. ISBN 9781605583921. Available at: https://doi.org/10.1145/1542476.1542490.

[9] Harmim, D. *Statická analýza v nástroji Facebook Infer zaměřená na detekci porušení atomičnosti*. Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně,

Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/21689/.

[10] Harmim, D., Marcin, V., Svobodová, L. and Vojnar, T. Static Deadlock Detection In Low-Level C Code. In: *Computer Aided Systems Theory – EUROCAST 2022: 18th International Conference, Las Palmas de Gran Canaria, Spain, February 20–25, 2022, Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2023, p. 267–276. DOI: 10.1007/978-3-031-25312-6_31. ISBN 978-3-031-25311-9. Available at: https://doi.org/10.1007/978-3-031-25312-6_31.

[11] Jannesari, A., Bao, K., Pankratius, V. and Tichy, W. F. Helgrind+: An efficient dynamic race detector. In: *IPDPS*. IEEE, 2009, p. 1–13. Available at: http://dblp.uni-trier.de/db/conf/ipps/ipdps2009.html#JannesariBPT09.

[12] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. jul 1978, vol. 21, no. 7, p. 558–565. DOI: 10.1145/359545.359563. ISSN 0001-0782. Available at: https://doi.org/10.1145/359545.359563.

[13] Lengál, O. and Vojnar, T. *Abstract Interpretation*. Presentation. Vysoké učení technické v Brně, Fakulta informačních technologií, 2022 [cit. 2022-01-15]. Available at: https://www.fit.vutbr.cz/study/courses/SAV/public/Lectures/sav-lecture-06.pdf.

[14] Letko, Z., Vojnar, T. and Křena, B. AtomRace: Data Race and Atomicity Violation Detector and Healer. In: *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. New York, NY, USA: Association for Computing Machinery, 2008. PADTAD '08. DOI: 10.1145/1390841.1390848. ISBN 9781605580524. Available at: https://doi.org/10.1145/1390841.1390848.

[15] Liu, B., Liu, P., Li, Y., Tsai, C.-C., Da Silva, D. et al. When Threads Meet Events: Efficient and Precise Static Race Detection with Origins. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2021, p. 725–739. PLDI 2021. DOI: 10.1145/3453483.3454073. ISBN 9781450383912. Available at: https://doi.org/10.1145/3453483.3454073.

[16] Marcin, V. *Statická analýza v nástroji Facebook Infer zaměřená na detekci uváznutí*. Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/21920/.

[17] Møller, A. and Schwartzbach, I. M. *Static Program Analysis*. Department of Computer Science, Aarhus University, October 2018.

[18] Nielson, F., Nielson, H. and Hankin, C. *Principles of Program Analysis*. Springer Berlin Heidelberg, 2015.

[19] Pavela, O. *Statická analýza v nástroji Facebook Infer zaměřená na analýzu výkonnosti*. Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/21919/.

[20] Peter O'Hearn, C. C. *Open-sourcing Facebook Infer: Identify bugs before you ship - Engineering at Meta* [online]. [cit. 2023-01-10]. Available at: https://engineering.fb.com/2015/06/11/developer-tools/open-sourcing-facebook-infer-identify-bugs-before-you-ship/.

[21] Sam Blackshear, J. V. *Building your own compositional static analyzer with Infer.AI* [online]. [cit. 2023-01-25]. Available at: https://fbinfer.com/downloads/pldi17-infer-ai-tutorial.pdf.

[22] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P. and Anderson, T. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* New York, NY, USA: Association for Computing Machinery. nov 1997, vol. 15, no. 4, p. 391–411. DOI: 10.1145/265924.265927. ISSN 0734-2071.

[23] Serebryany, K. and Iskhodzhanov, T. ThreadSanitizer: Data Race Detection in Practice. In: *Proceedings of the Workshop on Binary Instrumentation and Applications*. New York, NY, USA: Association for Computing Machinery, 2009, p. 62–71. WBIA '09. DOI: 10.1145/1791194.1791203. ISBN 9781605587936.

[24] Thomson, P. Static Analysis: An Introduction: The Fundamental Challenge of Software Engineering is One of Complexity. *Queue*. New York, NY, USA: Association for Computing Machinery. aug 2021, vol. 19, no. 4, p. 29–41. DOI: 10.1145/3487019.3487021. ISSN 1542-7730. Available at: https://doi.org/10.1145/3487019.3487021.

[25] Vojnar, T. *Static Analysis and Verification* [online]. Presentation. Vysoké učení technické v Brně, Fakulta informačních technologií, 2022 [cit. 2022-01-15]. Available at: https://www.fit.vutbr.cz/study/courses/SAV/public/Lectures/sav-lecture-01.pdf.

[26] Wu, J., Tang, Y., Hu, H., Cui, H. and Yang, J. Sound and Precise Analysis of Parallel Programs through Schedule Specialization. In: *Proc. of PLDI'12*. ACM, 2012.

[27] Yi, K. *Inferbo: Infer-based buffer overrun analyzer - Meta Research* [online]. 6. February 2017 [cit. 2022-01-09]. Available at: https://research.facebook.com/blog/2017/2/inferbo-infer-based-buffer-overrun-analyzer/.

[28] *Coverity Scan - Static Analysis* [online]. [cit. 2023-04-23]. Available at: https://scan.coverity.com/.

[29] *Infer Static Analyzer* [online]. February 2015 [cit. 2023-05-05]. Available at: https://fbinfer.com/.

[30] *Klocwork for C, C++, C#, Java, JavaScript, Python, and Kotlin | Perforce* [online]. [cit. 2023-05-23]. Available at: https://www.perforce.com/products/klocwork.

[31] *Starvation*. [cit. 2022-01-09]. Available at: https://fbinfer.com/docs/checker-starvation/.

[32] *SV-COMP - International Competition on Software Verification*. [cit. 2023-04-23]. Available at: https://sv-comp.sosy-lab.org/.

Appendices

Appendix A

Contents of the Attached Memory Media

The attached memory media contains the following:

- `/xsvobo1x_ibt.pdf`
  - This thesis in PDF format.
- `/tex/`
  - The source code of this thesis.
- `/infer/`
  - The source code of the Infer framework with the DarC plugin.
- `/experiments/`
  - Programs used for the experimental evaluation of the DarC plugin.
  - Additional information is provided in the `README.md` file located in this folder.
- `/README.md`
  - File containing an installation and user manual.