BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2019
http://excel.fit.vutbr.cz

Scalable Static Analysis Using Facebook Infer

Dominik Harmim*, Vladimír Marcin**, Ondřej Pavela***

Abstract
What is the problem? What is the topic?, the aim of this paper? Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce ullamcorper suscipit euismod. Mauris sed lectus non massa molestie congue. In hac habitasse platea dictumst. How is the problem solved, the aim achieved (methodology)? Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce ullamcorper suscipit euismod. Mauris sed lectus non massa molestie congue. In hac habitasse platea dictumst. Curabitur massa neque, commodo posuere fringilla ut, cursus at dui. Nulla quis purus a justo pellentesque. What are the specific results? How well is the problem solved? Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce ullamcorper suscipit euismod. Mauris sed lectus non massa molestie congue. In hac habitasse platea dictumst. So what? How useful is this to Science and to the reader? Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce ullamcorper suscipit euismod.

Keywords: Facebook Infer — Static Analysis — Abstract Interpretation — Atomicity Violations — Concurrent Programs

Supplementary Material: Facebook Infer — Facebook Infer Repository — Atomicity Violations Analyzer Repository

* xharmi00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*
** xmarci10@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*
*** xpavel34@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Bugs are inherent part of software since the inception of programming discipline. Current solutions to this problem mostly include extensive automated testing and dynamic analysis tools such as profilers. These solutions are sufficient in many cases but mostly fail in others or they are outright not usable such as in early development cycles without a running prototype. Static analysis can provide complementing alternative which however historically had shortcoming in scalability department. *Facebook Infer* provides a solution to this problem with its highly scalable *compositional* and *incremental* inter-procedural analyses.

However, the current version of Infer still lacks in concurrency and performance areas. It provides fairly advanced data race analysis and relatively new deadlock analysis which unfortunately does not work well on C programs. The only performance oriented analyzer focuses on *worst-case execution time* analysis that has certain drawbacks which we will cover later.

[Motivation] What is the raison d'être of your project? Why should anyone care? No general meaningless claims. Make bulletproof arguments for the importance of your work. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer sit amet neque vel mi sodales interdum nec a mi. Aliquam eget turpis venenatis, tincidunt purus eget, euismod neque. Nulla et porta tortor, id lobortis turpis. Sed scelerisque sem eget ante interdum, vel volutpat arcu volutpat.

[Problem definition] What exactly are you solving? What is the core and what is a bonus? What parameters should a proper solution of the problem have? Define the problem precisely and state how its solution should be evaluated. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque non arcu quis nunc efficitur vestibulum. Integer gravida neque suscipit diam porta aliquet. Maecenas porttitor libero ut turpis porttitor, auctor porta ligula rhoncus. Etiam a turpis blandit, eleifend dolor eget, egestas ligula. Nullam sollicitudin pulvinar mi sit amet interdum. Etiam in ultrices ante. Suspendisse potenti. Duis vel nisi eget tellus volutpat tempor. Etiam laoreet magna elit, et sollicitudin lectus tempor sit. Maecenas porttitor libero ut turpis porttitor, auctor porta ligula rhoncus. Etiam a turpis blandit, eleifend dolor eget, egestas ligula.

[Existing solutions] Discuss existing solutions, be fair in identifying their strengths and weaknesses. Cite important works from the field of your topic. Try to define well what is the *state of the art*. You can include a Section 2 titled "Background" or "Previous Works" and have the details there and make this paragraph short. Or, you can enlarge this paragraph to a whole page. In many scientific papers, *this* is the most valuable part if it is written properly. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent congue enim eu eros dictum sagittis. Aliquam ligula arcu, gravida at augue et, aliquet condimentum nulla. Morbi a lectus arcu. Nam ac commodo nisi, a accumsan nunc. Nam sed ante vel nulla elementum lobortis. Aliquam sed laoreet risus. Etiam ipsum odio, gravida eget sapien dictum, eleifend aliquet ex. Duis dapibus vitae enim vitae bibendum. Phasellus eget pulvinar massa. Mauris ornare urna. Maecenas porttitor libero ut turpis porttitor, auctor porta ligula rhoncus. Etiam a turpis blandit, eleifend dolor eget, egestas ligula. Nullam sollicitudin pulvinar mi sit amet interdum. Etiam in ultrices ante. Suspendisse potenti. Duis vel nisi eget tellus volutpat tempor. Suspendisse potenti. Duis vel nisi eget tellus volutpat tempor.

[Our solution] Make a quick outline of your approach – pitch your solution. The solution will be described in detail later, but give the reader a very quick overview now. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi laoreet risus a egestas imperdiet. Ut egestas nibh non fermentum vestibulum. Nullam quis eleifend ex, sed maximus nisl. Mauris maximus non dolor id tristique. Nunc pulvinar congue gravida. Nullam lobortis viverra leo sed commodo. Nulla in elit congue, ullamcorper metus non, eleifend risus. Vivamus porttitor, ex nec porttitor pretium, libero turpis ultrices dui, eu efficitur ante ipsum vel justo. Vivamus nec nulla nisi. Aenean quis mauris vitae metus gravida congue.

[Contributions] Sell your solution. Pinpoint your achievements. Be fair and objective. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer sit amet neque vel mi sodales interdum nec a mi. Aliquam eget turpis venenatis, tincidunt purus eget, euismod neque. Nulla et porta tortor, id lobortis turpis. Sed scelerisque sem eget ante interdum, vel volutpat arcu volutpat. Aliquam cursus, dolor a luctus.

2. Facebook Infer

Facebook Infer is an open-source static analysis tool which is able to discover inter-procedural bugs in a *scalable* manner. Infer was originally a standalone analyzer focused on memory safety violations which has made its breakthrough thanks to the influential paper [1]. Since then it has evolved into a general abstract interpretation framework that can be used to quickly develop new kinds of simple intra-procedural or *compositional* and *incremental* inter-procedural analyses based on *summaries*. Summary is a custom data structure that stores usually parametric information about the analyzed procedure. It allows Infer to analyze each procedure only once and then reuse the acquired info at multiple callsites. The incremental property refers to the ability to analyze individual code changes which makes Infer suitable for large and quickly changing codebases where the conventional batch analysis is unfeasible.

It currently supports analysis of C, C++, Objective-C and Java programs and provides wide range of analyses each focusing on different bug types. List of more matured analyses includes for example *Inferbo* (buffer overruns), *RacerD* (data races) or *Starvation* (concurrency starvation and some types of deadlocks).

The simplified architecture of the abstract interpretation (AI) framework consisting of three main components can be seen in figure 1.

The front-end is responsible for compilation from a source language into the Smallfoot Intermediate Language (SIL) used by AI and generation of Control Flow Graph (CFG) for each analyzed procedure. Thanks to the front-end module, it is possible to write language-independent analyses.

The abstract interpreter is responsible for the analysis of individual procedures. It takes a CFG and a module implementing the effect of each SIL instruc-

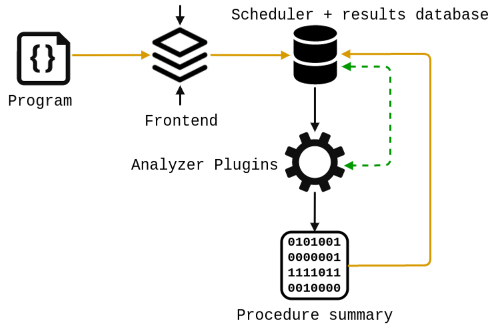


Figure 1. Architecture components

tion and produces a summary. The interpretation process is described in figure 2. The *command interpreter*

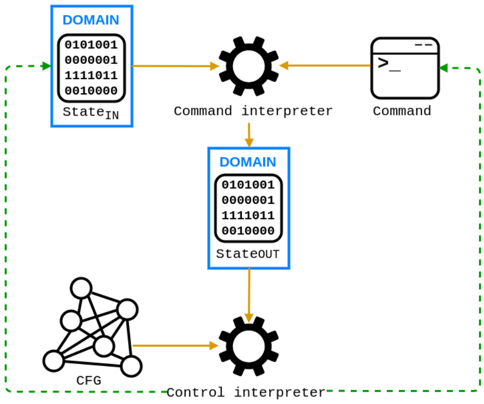


Figure 2. Interpretation process

interprets SIL instruction over the input state, produces new output state and sends it to the *control interpreter* which provides next input state and instruction based on a CFG.

The scheduler determines the order of analysis for each procedure based on a *call graph* and allows Infer to run in a heavily parallelized manner as it checks which procedures can be analyzed concurrently. Scheduler then stores the results in a database for later use in order to ensure the *incremental* property of analysis.

Call graph is a directed graph describing call dependencies between procedures. We can demonstrate the analysis order and the incremental property on figure 3.

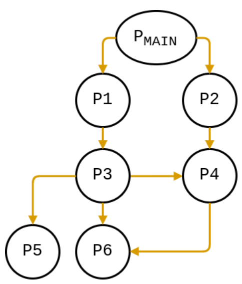


Figure 3. Call graph

The initial run would start with leaf procedures `P5` and `P6` and then would proceed towards the root $\mathrm{P}_{\mathtt{main}}$ while respecting the dependencies represented by edges. This order ensures that we will always have a summary already available when we encounter a procedure call during the analysis.

Each subsequent code change would then trigger re-analysis of directly affected procedures and also all procedures up the call chain. For example, if we modified the procedure `P3`, Infer would have to re-analyze only the `P3`, `P1` and $\mathrm{P}_{\mathtt{main}}$.

3. Worst-case cost analyzer

Recently, performance issues has become considerably more widespread in code leading to a poor user experience [2]. This kind of bugs is hard to manifest during the testing and so employing static analysis is nowadays more common. Facebook Infer currently provides only the *cost* checker [3], which implements a *worst-case execution time* complexity analysis (*WCET*). However, this *WCET* analysis provides only a numerical bound on number of executions of the program — a bound that is hard to interpret and, most of all, is insufficient for more complex algorithms, e.g., requiring amortized reasoning. Loopus [4] is a powerful resource bounds analyzer, which to the best of our knowledge is the only one that can handle the *amortized complexity analysis* for a broad range of programs. However, Loopus is limited to the intraprocedural analysis only and the tool itself does not scale well. Infer, on the other hand, offering the principles of *compositionality*, can handle even large projects. Hence, recasting the powerful analysis of Loopus within the Infer could enable a more efficient resource bounds analysis usable in today's rapid development.

Cost bounds inferred by Loopus refer to the number of possible *back jumps* to loop headers which is a useful metric related to *asymptotic time complexity* as it corresponds to the possible number of executions of instructions inside the loop. The bound algorithm relies on a simple abstract program model called *difference constraint program* (DCP) which can be seen in figure 4b.

Listing 1. Snippet demonstrating the need for amortized complexity analysis. Corresponding abstraction in figure 4b. Cost: $3n$

```
void foo(int n) {
     int i = n, j = 0, z = 0;
l1 : while (i > 0) {
         i--; j++;
l2 :     while (j > 0 && *) {
             j--; z++;
         }
     }
     int x = z;
l3 : while (x > 0)
```

```
        x--;
}
```

Each transition τ of a DCP has a *local bound* τ_v which is a variable v that *locally* limits the number of executions of transition τ as long as some other transitions that might increase the value of v are not executed. For example, the variable j in figure 4b limits the number of consecutive executions of transition τ_2 but not the total number as j might increase on other transitions.

The bound algorithm itself is based on the idea of reasoning about *how often* and *by how much* might the local bound of a transition τ increase which in turn affects the number of executions of τ. There are two main procedures that constitute the algorithm:

1. $\mathcal{VB}$ – computes a *variable bound* expression in terms of program parameters which bounds the value variable v.
2. $\mathcal{TB}$ – computes a bound on the number of times that a transition τ can be executed. Transitions that are not part of any loop have bound of 1.

The $\mathcal{TB}$ procedure is defined in a following way:

$$\mathcal{TB}(\tau) = \texttt{Incr}(\tau_v) + \texttt{Resets}(\tau_v) \quad (1)$$

The $\texttt{Incr}(\tau_v)$ procedure implements the idea of reasoning *how often* and *by how much* might the local bound τ_v increase:

$$\sum_{(\texttt{t},\texttt{c}) \in \mathcal{I}(\tau_v)} \mathcal{TB}(\texttt{t}) \times \texttt{c} \quad (2)$$

The $\mathcal{I}(\tau_v)$ is a set of transitions that increase the value of τ_v by $\texttt{c}$. The $\texttt{Resets}(\tau_v)$ procedure takes into account the possible resets of local bound τ_v to some arbitrary values which also add to the total amount by which it might increase:

$$\sum_{(\texttt{t},\texttt{a},\texttt{c}) \in \mathcal{R}(\tau_v)} \mathcal{TB}(\texttt{t}) \times \max(\mathcal{VB}(\texttt{a}) + \texttt{c}, 0) \quad (3)$$

The $\mathcal{R}(\tau_v)$ is a set of transitions that reset the value of local bound τ_v to $\texttt{a} + \texttt{c}$ where $\texttt{a}$ is a variable.

The remaining $\mathcal{VB}(\texttt{v})$ procedure is defined as:

$$\mathcal{VB}(\texttt{v}) = \texttt{Incr}(\texttt{v}) + \max_{(\texttt{t},\texttt{a},\texttt{c}) \in \mathcal{R}(\texttt{v})} (\mathcal{VB}(\texttt{a}) + \texttt{c}) \quad (4)$$

It picks the maximal value of all possible resets of variable `v` as an initial value which is subsequently increased by the amount obtained from $\texttt{Incr}(\texttt{v})$. Note that the procedure returns `v` itself if it is a program parameter or a numeric constant.

The complete bound algorithm is thus obtained through the mutual recursion of procedures $\mathcal{TB}$ and $\mathcal{VB}$. The main reason why this approach scales so well is *local* reasoning. Loopus does not rely on any global program analysis and is able to obtain complex invariants such as $x \leq \max(m1, m2) + 2n$ by means of bound analysis. These invariants are not expressible in common abstract domains such as *octagon* or *polyhedra* which would lead to a less precise result. This approach is also *demand-driven* (4a) which means that it performs only necessary recursive calls and does not greedily compute all possible invariants but only the ones that are needed for computation of specified bound. For full *flow* and *path sensitive* algorithm and its extension please refer to [4]

The table 4a presents simplified computation of transition bound of τ_5 from DCP 4b which was obtained through abstraction algorithm from the code snippet 1. This code snippet demonstrates the need for amortized complexity analysis as the worst-case cost of the l_2 loop can indeed be n. However, the amortized cost is 1 because the total number of iterations (total cost) is also equal to n due to the local bound j which is bounded by n. Loopus is thus able to obtain bound of n instead of n^2 for the inner loop l_2 unlike many other tools that cannot reason about amortized complexity. Another challenging problem is the computation of bound for the loop l_3. It is easy to infer z as the bound but the real challenge lies in expressing the bound in terms of program parameters. Thus, the real task is to obtain an invariant of form $z \leq \texttt{expr}(n)$ where $\texttt{expr}(n)$ denotes an expression over program parameters, n in this case. Loopus is able to obtain the invariant $z \leq n$ simply with the $\mathcal{VB}$ procedure and consequently infer the bound n for the loop l_3.

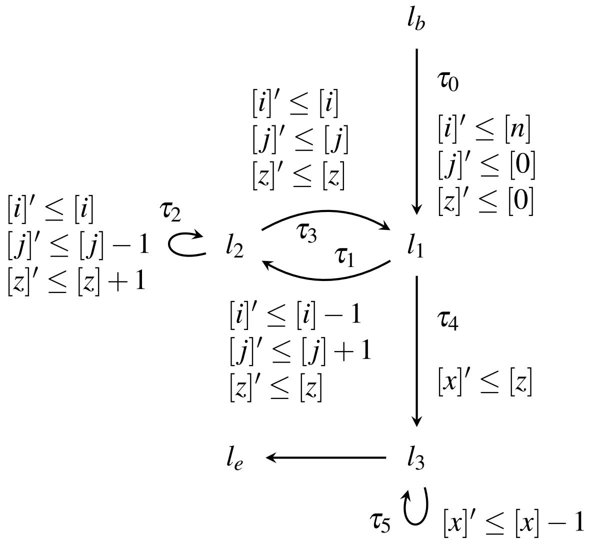
The table 1 presents results which we were able to achieve with our current implementation on few artificial examples. We compared the results of *Looper* (Loopus in Infer) with the *Cost* analyzer mentioned in the introduction of this section. Please note that the real cost of examples #4 and #6 is in fact $n \times max(n - 1, 0) + n$ and $3n + \max(m1, m2)$. Displayed cost of these examples is actually the worst-case asymptotic complexity instead of cost.

4. Deadlock analyzer

According to [5] deadlock is perhaps the most common concurrency error that might occur in almost all parallel programming paradigms including both shared-memory and distributed memory. To detect deadlock during testing is very hard due to many possible interleavings between threads. That's the reason why many of detectors were created, but most of them are quite heavyweight and do not scale well. However,

Call	Evaluation and Simplification
$\mathcal{TB}(\tau_5)$	$\rightarrow \texttt{Incr}([x]) +$ $\mathcal{TB}(\tau_4) \times \max(\mathcal{VB}([z]) + 0, 0)$ $\rightarrow 0 + 1 \times \max([n] + 0, 0) = [n]$
$\mathcal{VB}([z])$	$\rightarrow \texttt{Incr}([z]) + \max(\mathcal{VB}(0) + 0) = [n]$
$\texttt{Incr}([z])$	$\rightarrow \mathcal{TB}(\tau_2) \times 1 = [n]$
$\mathcal{TB}(\tau_2)$	$\rightarrow \texttt{Incr}([j]) + \mathcal{TB}(\tau_0) \times 0$ $\rightarrow [n] + 1 \times 0 = [n]$
$\texttt{Incr}([j])$	$\rightarrow \mathcal{TB}(\tau_1) \times 1 = [n]$
$\mathcal{TB}(\tau_1)$	$\rightarrow \texttt{Incr}([i]) + \mathcal{TB}(\tau_0) \times \max([n] + 0, 0)$ $\rightarrow 0 + 1 \times [n] = [n]$

(a) Simplified computation of bound for τ_5. $\texttt{Incr}([x])$ and $\texttt{Incr}([i])$ are 0 as there are no transitions that increase the value of $[x]$ or $[i]$. $\mathcal{TB}(\tau_0)$ and $\mathcal{TB}(\tau_4)$ are 1 as they are not part of any loop.



(b) Abstraction obtained from 1. Each transition is denoted by a set of invariant inequalities.

	Bound	Inferred bound		Time [s]	
		Looper	Cost	Looper	Cost
#1	n	$2n$	–	0.3	–
#2	$2n$	$2n$	–	0.5	–
#3	$4n$	$5n$	–	0.8	–
#4	n^2*	n^2	–	0.6	–
#5	$2n$	$2n$	–	0.3	–
#6	$n*$	n	–	0.6	–
#7	$2n$	$2n$	–	0.4	–
#8	$2n$	$2n$	–	0.7	–

Table 1. Experimental evaluation on selected examples used for evaluation of Loopus [4]. Benchmarks are publicly available at bitbucket.

there are a few that meet the scalability condition, like *starvation* analyzer implemented in Facebook Infer. The problem of this analyzer is that it uses heuristic on the class root of the access path of the lock so it doesn't handle a pure C lock. Also worth mentioning is the RacerX analyzer [6], which is based on counting so called *locksets* i.e. sets of locks currently held. RacerX uses interprocedural, flow-sensitive and context-sensitive analysis. What means that each function needs to be reanalysed in a new context. Hence, we decide to adapt lockset analysis from RacerX to follow principles of Facebook Infer and by that create context-insensitive analysis which will be faster and more scalable. So we present Low Level Deadlock Detector (L2D2), the principle of which will be illustrate with the example in Listing 2.

L2D2 works by first computing a summary for each function by looking for lock and unlock events. Example of lock and unlock is illustrated in Listing 2 at lines 22 and 27. If user function call appears in the analyzed code during analysis, like at line 26 of our example, the analyzer is provided with a summary of the function if available or the function is analyzed on demand. The summary is than applied to an abstract state at a call site. So in our example summary of `foo` will be applied to the abstract state of `thread1`.

Listing 2. Simple example capturing a deadlock between two global locks in C language using POSIX threads execution model

```
16 void foo() {
17     pthread_mutex_lock(&lock2);
18 }
21 void *thread1(...) {
22     pthread_mutex_lock(&lock1);
       ⋮
26     foo();
27     pthread_mutex_unlock(&lock1);
28 }
29 void *thread2(...) {
30     pthread_mutex_lock(&lock2);
       ⋮
36     pthread_mutex_lock(&lock1);
37 }
```

Next L2D2 looks through all the summaries of analyzed program and checks whether a potential deadlock can occur by computing transitive closure of relation consisting of all `dependencies` (see Listing 3) and checking if any lock depends on itself. The summaries for functions from the above example record information about the state of locks `lock1` and `lock2` as follows:

Listing 3. Summaries of functions in Listing 2

```
foo()
  PRECONDITION: { unlocked={lock2} }
```

```
  POSTCONDITION: { lockset={lock2} }
thread1(...)
  PRECONDITION: { unlocked={lock1, lock2} }
  POSTCONDITION: {
    lockset={lock1, lock2},
    dependencies={lock1->lock2}
  }
thread2(...)
  PRECONDITION: { unlocked={lock1, lock2} }
  POSTCONDITION: {
    lockset={lock1, lock2},
    dependencies={lock2->lock1}
  }
```

If we run L2D2 on code from our example it will report a possible deadlock between two threads due to cyclic dependency between `lock1`→`lock2` and `lock2`→`lock1` that arises if thread 1 holds `lock1` and waits on `lock2` and thread 2 hold `lock2` and waits on `lock1`.

4.1 Computing procedure summaries

In this subsection, we describe structure of the summary and process of computing it. To detect potential deadlock we need to record information that will allow us to answer these questions:

(1) What is the state of locks used in the analyzed program?
(2) Could cyclic dependency on pending threads occur?

To answer question (1), we have defined sets *lockset* and *unlockset*, which contains currently locked and unlocked locks respectively. We have also added sets *locked* and *unlocked* that serve as a precondition for a given function and contain locks that should be locked/unlocked before calling this function. Semantic of these sets is as follows:

```
semantics of lockset:
  lock(l)  → lockset = lockset ∪ {l}
  unlock(l) → lockset = lockset - {l}
semantics of unlockset:
  lock(l)  → unlockset = unlockset - {l}
  unlock(l) → unlockset = unlockset ∪ {l}
semantics of locked:
  if(lock(l) is first operation in f)
    unlocked_f = unlocked_f ∪ {l}
semantics of unlocked:
  if(unlock(l) is first operation in f)
    locked_f = locked_f ∪ {l}
```

The summary also contains a set of one-level *dependencies* by using which we can answer (2)nd question. Extraction of these *dependencies* is called on every lock acquisition and iterates over every lock in the current *lockset*, emitting the ordering constraint produced by the current acquisition. For example, if `lock2` is in the current *lockset* and `lock1` has just been acquired, the dependency `lock2`→`lock1` will be emitted, as we can see in Listing 2 in function `thread2`.

The most difficult part of dependencies extraction is elimination of false ones caused by invalid *locksets*. The main reasons for errors in *locksets* include the number of conditionals, function calls and degree of aliasing involved

Applying procedure summaries. As we mentioned at the beginning of this section, if a function call appears in an analyzed code, we have to apply a summary of the function to an abstract state at a callsite. Given callee g, its lockset L_g, unlockset U_g and caller f, its lockset L_f and unlockset U_f, we:

(1) Update the summary of g by replacing formal parameters with actual ones in case that locks were passed to the g as parameters. In the example below, you can notice that in the summary of g will be `lock4` replaced with `lock2`.
(2) Update the precondition of f:
$if(\exists l : l \in unlocked_g \wedge l \notin unlockset_f)$
add lock l to $unlocked_f$
$if(\exists l : l \in locked_g \wedge l \notin lockset_f)$
add lock l to $locked_f$
(3) Update the L_f: $L_f = (L_f \setminus U_g) \cup L_g$
(4) Update the U_f: $U_f = (U_f \setminus L_g) \cup U_g$
(5) Update the dependencies of f by adding new dependencies for all locks in the caller's lockset with locks which were locked in the callee. But what if all the locks that were acquired in the callee were also released there, as we can see in the example below.

```
void f() {
    pthread_mutex_lock(&lock2);
    g(&lock2);
}
void g(pthread_mutex_t *lock4) {
    pthread_mutex_lock(&lock3);
    pthread_mutex_unlock(lock4);
    pthread_mutex_lock(&lock1);
        ⋮
    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock3);
}
```

In that case, the callee's lockset will be empty and we have no information about these locks. So we had to add a new set to the summary which semantics is similar to the semantics of lockset except that unlock statement does not remove a lock from it. In our example, this set would contain `lock3` and `lock1` but there is still one problem left. What if the lock from the current lockset was unlocked in the callee before we locked another lock there? Then we will emit the wrong dependency `lock2→lock1`. In order to avoid this, we create `unlock→lock` type dependencies in summary, that can be used to safely determine the order of an operation in the callee. So this ensures that the only newly created correct dependency in our example will be `lock2→lock3`.

4.2 Reporting deadlocks

For deadlock detection, we use algorithm that iterates through all the summaries and computes the transitive closure of all dependencies. It records the cyclic lock dependency and displays the results to the user for inspection. Each deadlock is normally reported twice, at each trace starting point. So in our example in Listing 2, will be the deadlock reported for the first time in function `thread1` and for the second time in function `thread2`. TsF: I'm thinking whether this could be defined as some kind of lemma. E.g "Lemma 1. Given function f and its summary f_s and its transitive closure f_s^*, f contains deadlock iff $\exists a \in something : (a,a) \in f_s^*$. No proof. Fuck proofs.

4.3 Experimental evaluation

We performed our experiments by using 1002 concurrent C programs, that contain locks from the Debian GNU/Linux distribution. All benchmarks are available online at gitlab. These programs were used for experimental evaluation of Daniel Kroening's static deadlock analyser [citace] implemented in the CPROVER framework.

This benchmark set consists of 11.4 MLOC. Of all the programs, 994 are assumed to be deadlock-free and 8 of them have proved deadlock. Our experiments were run on a CORE i7-7700HQ at 2.80 GHz running Ubuntu 18.04 with 64-bit binaries with comparison to the CPROVER experiments which were run on a Xeon X5667 at 3 GHz running Fedora 20 with 64-bit binaries. In case of CPROVER were memory and CPU time restricted to 24GB and 1800 seconds per benchmark.

Results. Our analyzer as same as CPROVER correctly report all 8 potential deadlocks in benchmarks with known issues. Comparison of results for deadlock-free programs you can see in Table 2.

	proved	alarms	t/o	m/o	errors
CPROVER	**292**	114	453	135	0
L2D2	**810**	104	0	0	80

Table 2. Results for the programs without deadlock (t/o – timed out, m/o – out of memory)

As you can see L2D2 reported false alarms for 104 deadlock-free benchmarks what is 10 less than CPROVER. A much larger difference can be seen in cases where it was proved that there was no deadlock. The difference here is up to 518 examples in favor of our analyzer. In case of L2D2 you can see 80 compilation errors that were caused by syntax that Infer does not support. The biggest difference between our analyzer and CPROVER is runtime. While our analyzer needed approximately 2 hours to perform the experiments, CPROVER needed about 300 hours.

There is still space for improving our analysis by reduction of false alarms. The main reason for such alarms is false dependencies. Reasons for their existence we mentioned in subsection 4.1 (4^{th} paragraph). So eliminating false positives consists of techniques to eliminate false dependencies.

5. Atomicity Violations Analyzer

In *concurrent programs*, there are often *atomicity requirements* for an execution of specific sequences of instructions. Violating these requirements may cause many kinds of problems, such as a unexpected behaviour, exceptions, segmentation faults or other failures. *Atomicity violations* are usually not verified by compilers, unlike syntactic or some sorts of semantic rules. Atomicity requirements, in most cases, are not event documented. It means that typically only programmers must take care of following these requirements. In general, it is very difficult to avoid errors in *atomicity-related programs*, especially in large projects, and even harder and time-consuming is then finding and fixing these errors.

In this section of this paper, there is described a proposal and an implementation of an *static analyzer for finding atomicity violations*.

5.1 Contracts for Concurrency

The proposal of a solution is based on the concept of *contracts for concurrency* described in [7]. These contracts allow to define *sequences of functions* that are required to be *executed atomically*. The proposed analyzer itself is able to produce mentioned contracts, and then verify whether the contracts are fulfilled.

In [7], a *basic contract* is formally defined as follows. Let $\Sigma_{\mathbb{M}}$ be a set of all function names of a software module. A *contract* is a set $\mathbb{R}$ of *clauses* where each clause $\varrho \in \mathbb{R}$ is a regular expression over $\Sigma_{\mathbb{M}}$. A *contract violation* occurs if any of the sequences represented by the contract clauses is interleaved with an execution of functions from $\Sigma_{\mathbb{M}}$.

Consider an implementation of a function that replaces item `a` in an array by item `b`, as illustrates Listing 4. The contract for this specific scenario contains clause ϱ_1, which is defined and follows:

$$(\varrho_1)\ \texttt{index\_of set}$$

Clause ϱ_1 specifies that the execution of `index_of` followed by execution of `set` should be atomic. The index of an item in an array is acquired, and then the index is used to modify the array. Without atomicity, a concurrent modification of the array may change a position of the item. The acquired index then may be invalid when `set` is executed.

Listing 4. Example of a contract violation

```
void replace(int *array, int a, int b) {
    int i = index_of(array, a);
    if (i >= 0) {
        set(array, i, b);
    }
}
```

In paper [7], there is described a proposal and an implementation for a static validation which is based on *grammars* and *parsing trees*. Within paper [7] was implemented a stand-alone prototype tool[1] for analysing applications, written in Java language, which obtained promising experimental results. However, we decided to propose and implement the analysis quite different way, see 5.2. Moreover, we decided to implement this solution in the *Facebook Infer*, i.e., widely used, active and a open source tool. Therefore the analysis should be faster and more scalable thanks to the way how the Facebook Infer works, as it was described in section 2. The implementation is aimed for programs written in C/C++ languages using *POSIX Threads (Pthreads)* locks for *a synchronization of concurrent threads*. We are also focusing to reduce false positive errors.

In the Facebook Infer, there is already implemented an analysis called *Lock Consistency Violation*, see [2]. That analysis finds atomicity violations for writes/reads single variables that are required to be executed atomically. Ours analysis is more general because it finds *atomicity violations for sequences of functions* that are required to be executed atomically, i.e., it checks whether contracts for concurrency are fulfilled.

5.2 Two-Phase Analysis

The proposed solution is divided into two parts (*phases of analysis*):

Phase 1 Detection of *sequences of functions* that should be *executed atomically*.

Phase 2 Detection of *atomicity violations*.

[[TODO]]

5.3 Future Work

[[TODO]]

6. Conclusions

[Paper Summary] What was the paper about, then? What the reader needs to remember about it? Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin vitae aliquet metus. Sed pharetra vehicula sem ut varius. Aliquam molestie nulla et mauris suscipit, ut commodo nunc mollis.

[Highlights of Results] Exact numbers. Remind the reader that the paper matters. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed tempus fermentum ipsum at venenatis. Curabitur ultricies, mauris eu ullamcorper mattis, ligula purus dapibus mi, vel dapibus odio nulla et ex. Sed viverra cursus mattis. Suspendisse ornare semper condimentum. Interdum et malesuada fames ac ante ipsum.

[Paper Contributions] What is the original contribution of this work? Two or three thoughts that one should definitely take home. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent posuere mattis ante at imperdiet. Cras id tincidunt purus. Aliquam erat volutpat. Morbi non gravida nisi, non iaculis tortor. Quisque at fringilla neque.

[Future Work] How can other researchers / developers make use of the results of this work? Do you have further plans with this work? Or anybody else? Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse sollicitudin posuere massa, non convallis purus ultricies sit amet. Duis at nisl tincidunt, maximus risus a, aliquet massa. Vestibulum libero odio, condimentum ut ex non, eleifend.

Acknowledgements

We would like to thank our supervisors and colleagues from VeriFIT team – Tomáš Fiedor, Adam Rogalewicz and Tomáš Vojnar. We would also like to thank for the support received from the H2020 ECSEL project Aquas.

References

[1] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. *Compositional Shape Analysis by means of Bi-Abduction. In Proc. of POPL'09*, pages 289–300. 2009.

[1] https://github.com/trxsys/gluon

[2] https://fbinfer.com/docs/checkers-bug-types.html#LOCK_CONSISTENCY_VIOLATION

[2] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. *Understanding and Detecting Real-World Performance Bugs. In Proc. of PLDI'18*, pages 77–88. 2012.

[3] Stefan Bygde. *Static WCET analysis based on abstract interpretation and counting of elements*. PhD thesis, Mälardalen University, 2010.

[4] Moritz Sinn. *Automated Complexity Analysis for Imperative Programs*. PhD thesis, Vienna University of Technology, 2016.

[5] A.H. Dogru and V. Bicar. *Modern Software Engineering Concepts and Practices: Advanced Approaches*. Premier reference source. Information Science Reference, 2011.

[6] Dawson R. Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Operating Systems Review – SIGOPS*, volume 37, pages 237–252, January 2003. DOI: 10.1145/1165389.945468.

[7] Ricardo J. Dias, Carla Ferreira, Jan Fiedor, João M. Lourenço, Aleš Smrčka, Diogo G. Sousa, and Tomáš Vojnar. Verifying concurrent programs using contracts. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 196–206. IEEE, March 2017. DOI: 10.1109/ICST.2017.25. ISBN: 9781509060313.