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

Scalable Static Analysis Using Facebook Infer

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

Abstract
Static analysis has nowadays become one of the most popular ways of catching bugs early in the modern software. However, reasonably precise static analyses do still often have problems with scaling to larger codebases. Moreover, efficient static analysers, such as Coverity or Code Sonar, are often proprietary, rather expensive, and difficult to openly evaluate and/or extend. *Facebook Infer* offers a static analysis framework that is open source (despite being heavily used in multiple companies including Facebook itself), extendable, and promoting efficient modular ad incremental analysis. In this work, we propose three new inter-procedural analyzers extending the portfolio of analyzers available with Facebook Infer: *Looper* (a resource bounds analyser), *L2D2* (a low-level deadlock detector) and *Atomer* (an atomicity violation analyser). We evaluated our analyzers on both smaller hand-crafted examples as well as publicly available benchmarks derived from real-life low-level programs and obtained encouraging results. In particular, *L2D2* attained 100 %detection rate and 11 % false positive rate on an extensive benchmark of hundreds functions and millions of lines of code.

Keywords: Facebook Infer — Static Analysis — Abstract Interpretation — Atomicity Violations — Concurrent Programs — Performance — Worst-case Cost — Deadlock

Supplementary Material: Atomer Repository — Looper Repository — L2D2 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 an inherent part of software ever since the inception of the programming discipline. They tend to hide in unexpected places, and when they are triggered, they can cause significant damage. In order to catch bugs early in the development process, extensive automated testing and dynamic analysis tools such as profilers are often used. But while these solutions are sufficient in many cases, they can sometimes still miss too many errors. An alternative solution is static analysis, which has its own shortcomings as well. **[[Here, one should say something about why static analysis is problematic: You can reuse something from the abstract.]]**

Recently, Facebook has proposed its own solution for efficient bug finding and program verification called *Facebook Infer* — a highly scalable *compositional* and *incremental* framework for creating inter-procedural analyses. Facebook Infer is still under development , but it is in everyday use in Facebook (and several other companies, such as Spotify, Uber, Mozilla and others) and it already provides many checkers for various kinds of bugs, e.g., for verification of buffer overflow, thread safety or resource leakage.

However, equally importantly, it provides a suitable framework for creating new analyses quickly.

However, the current version of Infer still misses better support, e.g., for concurrency or performance-based bugs. While it provides a fairly advanced data race and deadlock analyzers, they are limited to Java programs only and fail for C programs, which require more thorough manipulation with locks. Moreover, the only performance-based analyzer focuses on *worst-case execution time* analysis only, which does not provide a reasonable understanding of the programs performance.

In particular, we propose to extend Facebook Infer with three analyzers: the *Looper*, a resource bounds analyser; the *L2D2*, a lightweight deadlock checker; and the *Atomer*, an atomicity violation checker working on the level of sequence of method calls. In experimental evaluation, we show encouraging results, when even our immature implementation could detect both concurrency property violations and infer precise bounds for selected benchmarks, including rather large benchmarks based on real-life code. The development of these checkers has been discussed several times with developers of Facebook Infer, and it is integral part of the H2020 ECSEL project Aquas.

2. Facebook Infer

Facebook Infer is an open-source static analysis framework which is able to either discover various types of bugs or, in some restricted cases, verify correctness of the given program, both in a *scalable* manner. Infer was originally a standalone analyzer focused on sound verification of absence of memory safety violations which has made its breakthrough thanks to an influential paper [1]. Since then, it has evolved into a general *abstract interpretation* [2] framework focused primarily on finding bugs rather than formal verification that can be used to quickly develop new kinds of *compositional* and *incremental* analyses based on the notion of function *summaries*. In theory, a summary is a representation of function's preconditions and postconditions or effects. In practice of Facebook Infer, it is a custom data structure that allows users to store arbitrary information resulting from function's analysis. Infer does (usually) not compute the summaries during a run of the analysis along the control flow graph as done in older analyzers. Instead, it analyzes a program function-by-function along the call tree, starting from its leafs. Hence, a a summary of a function is typically analyzed without knowing its call context. This helps scalability (since summaries computed in different contexts are not distinguished), but it may easily lead to a loss of precision, requiring developers of particular analyzers to rethink the way the analyzers work such that they still can produce useful information. The summary of a function is then used at all of its call sites. Furthermore, thanks to its incrementality, Infer can analyze individual code changes instead of the whole project, which is more suitable for large and quickly changing codebases where the conventional batch analysis is unfeasible. Intuitively, the incrementality is based on re-using summaries of functions for which there is no change in them nor in the functions (transitively) called from them.

Infer currently supports analysis of programs written in multiple languages including C, C++, Objective-C, and Java and provides a wide range of analyses, each focusing on different types of bugs, such as *Inferbo* (buffer overruns), *RacerD* (data races), or *Starvation* (concurrency starvation and selected types of deadlocks).

The architecture of the Infer's abstract interpretation framework (Infer.AI) can be divided into three main components as depicted in Figure 1: a frontend, an analysis scheduler, and a collective set of analysis plugins.

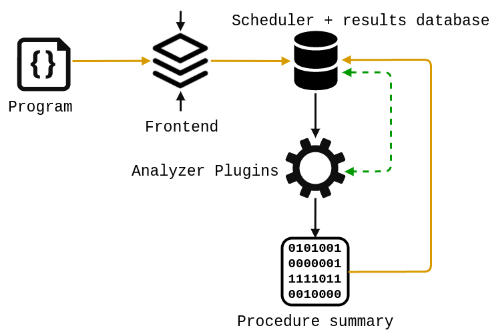


Figure 1. Infer's architecture components

The first component, the front-end, compiles input programs into the Smallfoot Intermediate Language (SIL) in a form of a Control Flow Graph (CFG). Each analyzed procedure has its own CFG representation. The frontend supports multiple languages, so one can write (to some degree) language-independent analyses.

The second component, the abstract interpreter or *command interpreter*, subsequently interprets SIL instructions over input abstract states and produces new output states which are further scheduled for interpretation based on the CFG. Its simplified workflow is described in Figure 2.

The last component, 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. The scheduler then stores the results of

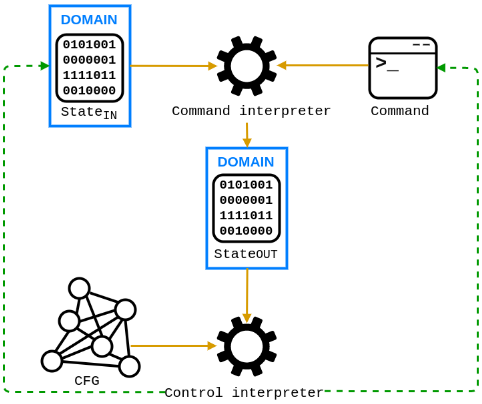


Figure 2. The interpretation process in Infer

analyses in a database for later use in order to ensure the *incremental* property of Infer.

In more detail, a call graph is a directed graph describing call dependencies between procedures. An example of a call graph is shown in Figure 3. Using this figure, we can illustrate the order of analysis in Infer and its incrementality as follows: The underlying

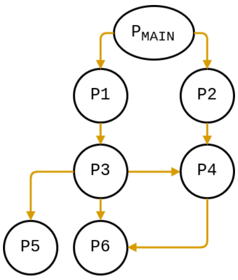


Figure 3. A call graph

analyzer starts with leaf functions `P5` and `P6` and then proceeds towards the root $\mathtt{P_{main}}$ while respecting the dependencies represented by the edges. This order ensures that we will always have a summary already available when we have to abstractly interpret a nested function call during our analysis. Each subsequent code change then triggers a re-analysis of the directly affected functions only as well as all functions up the call chain. For example, if we modify the function `P3`, Infer will re-analyze only `P3`, `P1`, and $\mathtt{P_{main}}$.

3. Worst-case Cost Analyzer

Recently, performance issues has become considerably more widespread in code leading to a poor user experience [3]. 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 [4], 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 [5] 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--;
}
```

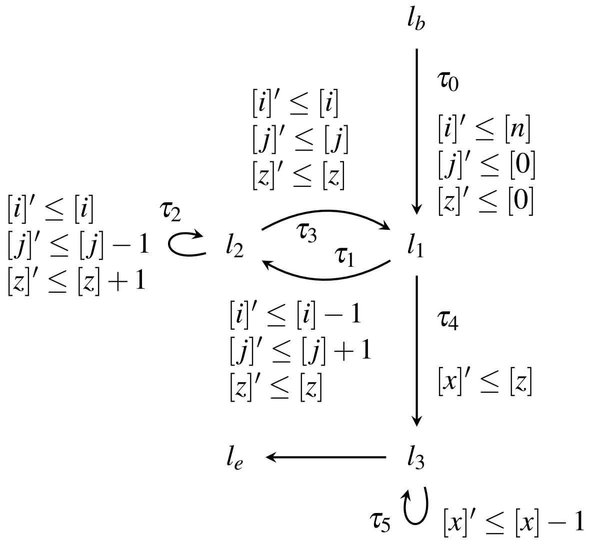
Each transition τ of a DCP has a *local bound* τ_v which is a variable v that *locally* limits the number of executions of transition τ as long as some other transitions that might increase the value of v are not executed. For example, the variable j in figure 4b limits the number of consecutive executions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

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

(a) Simplified computation of bound for τ_5. $\mathtt{Incr}([x])$ and $\mathtt{Incr}([i])$ 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.

that are not part of any loop have bound of 1.

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

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

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

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

The $\mathcal{I}(\tau_v)$ is a set of transitions that increase the value of τ_v by $\mathtt{c}$. The $\mathtt{Resets}(\tau_v)$ 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:

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

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

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

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

It picks the maximal value of all possible resets of variable $\mathtt{v}$ as an initial value which is subsequently increased by the amount obtained from $\mathtt{Incr}(\mathtt{v})$. Note that the procedure returns $\mathtt{v}$ 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 [5]

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 $z \leq \mathtt{expr}(n)$ where $\mathtt{expr}(n)$ 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.

	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 [5]. Benchmarks are publicly available at bitbucket.

4. Deadlock Analyzer

According to [6] 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, 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 [7], 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 the 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, unlockset U_f and dependencies D_f, we:

(1) Update the summary of g by replacing formal parameters with actual ones in case that locks were passed to 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 L_f: $L_f = (L_f \setminus U_g) \cup L_g$
(4) Update U_f: $U_f = (U_f \setminus L_g) \cup U_g$
(5) Update D_f by adding new dependencies for all locks in the L_f with locks which were locked in g. But what if all the locks which were acquired in g 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, L_g 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 operations 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`.

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 [8] 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. Some techniques have already been implemented but we are still working on others.

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 [9]. These contracts allow to define *sequences of functions* that are required to be *executed atomically*. The proposed analyzer itself (**Atomer**) is able to produce mentioned contracts, and then verify whether the contracts are fulfilled.

In [9], 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:

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

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. An 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 [9], there is described a proposal and an implementation for a static validation which is based on *grammars* and *parsing trees*. Within paper [9] was implemented a stand-alone prototype tool[1] for analysing programs written in Java language, which obtained promising experimental results. However, we

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

decided to propose and implement the analysis quite different way, see 5.2 and 5.3. 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], which is part of the *RacerD* [10]. That analysis finds atomicity violations for writes/reads single variables that are required to be executed atomically. Atomer 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.

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

Phase 1 The detection of *atomic sequences* 5.2

Phase 2 The detection of *atomicity violations* 5.3.

5.2 Detection of Atomic Sequences

Before the detection of *atomicity violations* may begin, it is required to have *contracts* introduced in section 5.1. The **Phase 1** of the Atomer is able to produce such contracts, i.e., it detects *sequences of functions* that should be *executed atomically*.

During the analysis, *first occurrences* of functions, which are called *non-atomically* (without a `lock`), are detected. When the beginning of an atomic sequence appears (a `lock` call), a nested detection of first occurrences commences. An `unlock` call closes the atomic sequence, and induces a check of a stored redundant sequences. At the end of the analyzed function, the following two sets are derivated into the *summary* of the function. (i) The set of atomic sequences. (ii) The set of sequences of all function calls (this set is used in a higher level of the function calls tree).

Within an analysis of the function `g` from Listing 5 (assume *Pthreads* locks and existence of the initialized global variable `lock` of the type `pthread_mutex_t`), a process of the detection is as follows (a strikethrough indicates removal of duplicates):

$$\texttt{f1}\ \cancel{\texttt{f1}}(\texttt{f1}\ \cancel{\texttt{f1}}\ \texttt{f2})|\texttt{f1}\ \cancel{\texttt{f1}}(\texttt{f1}\ \texttt{f3})|\cancel{\texttt{f1}}(\cancel{\texttt{f1}\ \texttt{f3}\ \texttt{f3}})$$

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

The derivated sets for the function `g`:

(i) {`(f1 f2) (f1 f3)`}
(ii) {`f1 f2 f3`}

Listing 5. Sequences of functions executed atomically

```
void g(void) {
    f1(); f1();
    pthread_mutex_lock(&lock);
    f1(); f1(); f2();
    pthread_mutex_unlock(&lock);
    f1(); f1();
    pthread_mutex_lock(&lock);
    f1(); f3();
    pthread_mutex_unlock(&lock);
    f1();
    pthread_mutex_lock(&lock);
    f1(); f3(); f3();
    pthread_mutex_unlock(&lock);
}
```

Presume an analysis of the function `h` from Listing 6, where is nested the call of the function `g`. A process of the detection is as follows (the set of sequences of all function calls from the nested function, set (ii), is used):

$$\texttt{f1 g}\ \cancel{\texttt{f1}}\ \texttt{f2 f3}(\texttt{g f1 f2 f3})$$

The derivated sets for the function `h`:

(i) {`(g f1 f2 f3)`}
(ii) {`f1 g f2 f3`}

Listing 6. Sequences of functions executed atomically with nested function call

```
void h(void) {
    f1(); g();
    pthread_mutex_lock(&lock);
    g();
    pthread_mutex_unlock(&lock);
}
```

This detection of atomic sequences has been implemented and successfully verified on a set of sample programs created for this purpose. Sets of atomic sequences, set (i), are stored into a file and it is used during the **Phase 2**, 5.3. There are some possibilities for further extending and improving of the **Phase 1**, e.g., work with nested locks, support for multiple locks or extend detection for other types of locks for a synchronization of concurrent threads/processes.

5.3 Detection of Atomicity Violations

In a *detection of the atomicity violations* phase, the set of atomic sequences from **Phase 1**, 5.2, is taken, and it is detected a violation for any pair of function calls which has occurred consecutively in one of the atomic sequence. For instance, assume functions `g` and `h` from Listing 7. The set of atomic sequences of the function `g` is {`(f2 f3)`}. In the function `h`, atomicity violations is detected because of functions `f2` and `f3` are not called atomically (under a lock).

Listing 7. Atomicity violation

```
void g(void) {
    f1();
    pthread_mutex_lock(&lock);
    f2(); f3();
    pthread_mutex_unlock(&lock);
    f4();
}
void h(void) {
    f1(); f2(); f3(); f4();
}
```

Implementation of this phase is in the process and it will be finalized and verified within a bachelor's thesis.

6. Conclusions

In this paper, we presented three new analyzers which we implemented in the *Facebook Infer* tool alongside the existing ones. The *Looper* resource bounds analyzer was able to infer the precise bound in 6 out of 8 of selected examples used for evaluation of the original *Loopus* tool. The remaining two bounds differed only in the constant factor. The *L2D2* analyzer focusing on deadlock detection in C programs was evaluated on Daniel Kroening's benchmark with 100 % success rate in detection of potential deadlocks and roughly 11 % false positives rate. It also proved the scalability of the approach as it managed to finish the benchmark in less than 1 % of the time needed by the Kroening's `CPROVER` tool. The first phase of the *Atomer* – the atomicity violations analyzer, a detection of sequences of functions that should be executed atomically, was successfully verified on a set of sample programs created for this purpose. The second phase, a detection of atomicity violations, will be finalized and tested within a bachelor's thesis.

Our analyzers have potential for further extending and improving the accuracy of theirs results. So our further work will focus mainly on increasing the accuracy of our methods, and testing them on real-world programs. Furthermore we would like to merge our implementations to a `master` branch of the *Facebook Infer repository*[3].

[3] `https://github.com/facebook/infer`

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.

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

[3] 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.

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

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

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

[7] 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.

[8] D. Kroening, D. Poetzl, P. Schrammel, and B. Wachter. Sound static deadlock analysis for c/pthreads. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 379–390, September 2016.

[9] 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.

[10] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. Racerd: Compositional static race detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):144:1–144:28, October 2018.