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**

*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. And efficient static analysers, such as Coverity or Code Sonar, are often proprietary and difficult to openly evaluate or extend. *Facebook Infer* offers a static analysis framework that is open source, extendable, and promoting efficient modular and incremental analysis. In this work, we propose three inter-procedural analyzers extending the capabilities of 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 a static analysis, which has its own shortcomings as well. Like, for example, a high rate of false positives and, in particular, quite a big problem with scalability.

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 discover various types of bugs of the given program, 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 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.

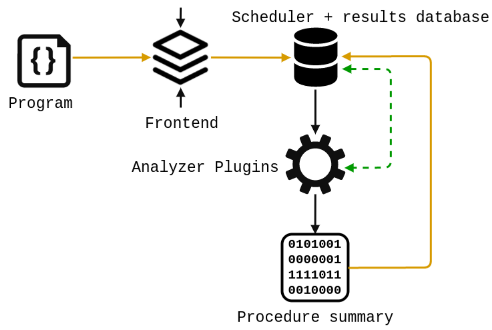



**Figure 1.** Infer's architecture components

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.

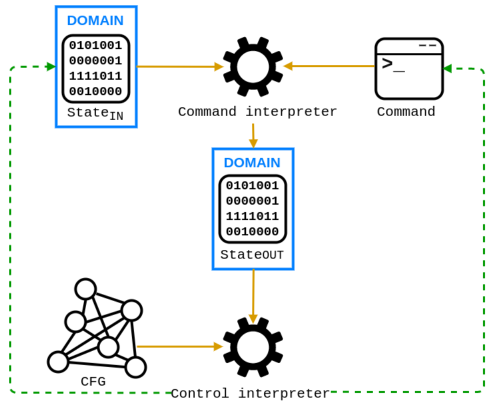



**Figure 2.** The interpretation process in Infer

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 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. The underlying analyzer starts with leaf functions `P5` and `P6` and then proceeds towards the root $P_{\texttt{main}}$ 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 $P_{\texttt{main}}$.

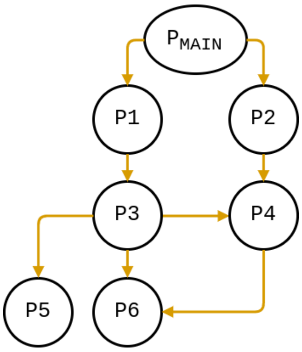



**Figure 3.** A call graph

## 3. Worst-Case Cost Analyzer

Recently, performance issues have become considerably more widespread in code, leading to a poor user experience [3]. This kind of bugs is hard to manifest during testing, and so employing static analysis is particularly useful in this case. Facebook Infer currently provides the *cost* checker [4] only, which implements a *worst-case execution time* complexity analysis (*WCET*). However, this *WCET* analysis provides a numerical bound on the number of executions of the program only — a bound that is hard to interpret, and, above all, it is quite imprecise 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 *amortized complexity analysis* for a broad range of programs. However, Loopus is limited to 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 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 an 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 a *difference constraint program* (DCP), an example of which can be seen in Figure 4b.

**Listing 1.** A snippet demonstrating the need for amortized complexity analysis. The corresponding abstraction is shown in Figure 4b. The cost of the outer loop is $3n$

```
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 $\tau$ of a DCP has a *local bound* $\tau_v$ which is a variable $v$ that *locally* limits the number of executions of the transition $\tau$ 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 the transition $\tau_2$ 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 $\tau$ increase, which in turn affects the number of executions of $\tau$. 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 of the variable $v$.
2. $\mathcal{TB}$ – computes a bound on the number of times that a transition $\tau$ can be executed. Transitions that are not part of any loop have bound of 1.

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

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

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

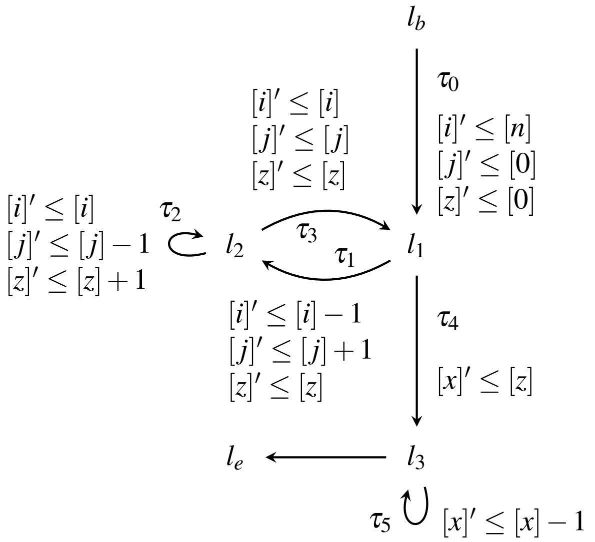
$$\texttt{Incr}(\tau_v) = \sum_{(\texttt{t},\texttt{c}) \in \mathcal{I}(\tau_v)} \mathcal{TB}(\texttt{t}) \times \texttt{c} \tag{2}$$

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

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

| 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)** A simplified computation of the bound for $\tau_5$. $\mathtt{Incr}([x])$ 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)** An abstraction obtained from Listing 1. Each transition is denoted by a set of invariant inequalities.

**Figure 4**

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

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

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

It picks the maximal value of all possible resets of the variable $\mathtt{v}$ as an initial value which is subsequently increased by the value 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 the 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 bounds 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* (Figure 4a), which means that it performs necessary recursive calls only and does not greedily compute all possible invariants but only the ones that are needed for computation of the specified bound. For a full *flow* and *path sensitive* algorithm and its extension please refer to [5].

Table 4a presents a simplified computation of the transition bound of $\tau_5$ from the DCP shown in Figure 4b, which was from the code snippet shown in Listing 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 the 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 the 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 the 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 implementation of procedures $\mathcal{TB}$ and $\mathcal{VB}$ was quite straightforward thanks to the use of a functional language (OCaml). However, the first step was to implement a custom algorithm for conversion of the native CFG used by Infer into a *labeled transition system* (LTS) which is subsequently converted into a DCP by Loopus' abstraction algorithm. We leverage the AI framework in order to obtain the LTS through a symbolic execution of a program. Additionally, it was necessary to implement the abstraction algorithm and an algorithm which determines local bounds for a given DCP. We also managed to implement some extensions which improve the precision of the basic presented bound algorithm such as reasoning based on *reset chains* or an algorithm that converts the standard DCP into a *flow-sensitive* one by means of variable renaming. For details please refer to [5].

Table 1 presents results which we were able to achieve with our current implementation on several (so far) toy 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 $3n+\max(m1,m2)$. The displayed cost of these examples is actually the worst-case asymptotic complexity instead of the cost.

**Table 1.** An experimental evaluation of *Looper* on selected examples used originally for an evaluation of Loopus [5]. Benchmarks are publicly available at bitbucket.

| | Bound | Inferred bound | | Time [s] | |
|---|---|---|---|---|---|
| | | Looper | Cost | Looper | Cost |
| #1 | $n$ | $2n$ | $n^2$ | 0.3 | 0.4 |
| #2 | $2n$ | $2n$ | $5n$ | 0.5 | 0.4 |
| #3 | $4n$ | $5n$ | $\infty$ | 0.8 | 1.4 |
| #4 | $*n^2$ | $n^2$ | $\infty$ | 0.6 | 0.9 |
| #5 | $2n$ | $2n$ | $12n$ | 0.3 | 0.5 |
| #6 | $*n$ | $n$ | $\infty$ | 0.6 | 0.7 |
| #7 | $2n$ | $2n$ | $\infty$ | 0.4 | 1 |
| #8 | $2n$ | $2n$ | $\infty$ | 0.7 | 1.8 |

## 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 deadlocks during testing is very hard due to many possible interleavings between threads. Of course, one can use extrapolating dynamic analysers and/or techniques such as noise injection or systematic testing [7] to increase chances of finding deadlocks, but such techniques decrease the scalability of the testing process and can still have problems to discover some errors. That is the reason why many static 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 the *starvation* analyzer implemented in Facebook Infer. The problem of this analyzer is that it uses a heuristic based on using the class of the root of the access path of a lock, and so it does not handle pure C locks. Also worth mentioning is the RacerX analyzer [8], which is based on counting so-called *locksets*, i.e., sets of locks currently held. RacerX uses interprocedural, flow-sensitive, and context-sensitive analysis. This means that each function needs to be reanalysed in a new context, which reduces the scalability. Hence, we have decided to adapt the *lockset analysis* from RacerX to follow principles of Facebook Infer and, this way, create a new context-insensitive analysis, which will be faster and more scalable. We have implemented this analysis in our *Low-Level Deadlock Detector (L2D2)*, the principle of which will be illustrated by the example in Listing 2 (a full description of the algorithm with all its optimisations is beyond the scope of this paper).

L2D2 works in two phases. In the first phase, it computes a summary for each function by looking for lock and unlock events present in the function. An example of a lock and unlock event is illustrated in Listing 2 at lines 22 and 27. If a call of an user-defined function appears in the analyzed code during the 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 (which effectively leads to analysing the code along the call tree, starting at its leaves as usual in Facebook Infer). The summary is then applied to an abstract state at a call site. So, in our example, the summary of `foo` will be applied to the abstract state of `thread1`. More details on how the summaries look like and how they are computed will be given in Section 4.1.

**Listing 2.** A simple example capturing a deadlock between two global locks in the C language using the 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 }
```

In the second phase, L2D2 looks through all the computed summaries of the analyzed program, and concentrates on the so-called *dependencies* that are a part of the summaries. The dependencies record that some lock got locked at a moment when another lock was still locked. L2D2 interprets the obtained set of dependencies as a relation, computes its transitive closure, and reports a deadlock if some lock depends on itself in the transitive closure.

**Listing 3.** Summaries of the functions in Listing 2

```
foo()
  PRECONDITION: { unlocked={lock2} }
  POSTCONDITION: { lockset={lock2} }
thread1(...)
```

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

If we run L2D2 on the code from our example, it will report a possible deadlock between two threads due to the cyclic dependency between `lock1` and `lock2` that arises if thread 1 holds `lock1` and waits on `lock2` and thread 2 holds `lock2` and waits on `lock1`. This cycle will be deduced from the dependencies `lock1`→`lock2` and `lock2`→`lock1` that appear in the summaries of `thread1` and `thread2` (see Listing 3).

## 4.1 Computing Procedure Summaries

In this subsection, we describe the structure of the summaries used and the process of computing them. To detect potential deadlocks, we need to record information that will allow us to answer the following questions:

(1) What is the state of the locks used in the analyzed program at a given point in the code?
(2) Could a cyclic dependency on pending lock requests occur?

To answer question (1), we compute sets *lockset* and *unlockset*, which contain the currently locked and unlocked locks, respectively. These sets are a part of the postconditions of functions and record what locks are locked/unlocked upon returning from a function, respectively. Further, we also compute 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. When analyzing a function, the sets are manipulated as follows:

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

Each summary also contains a set of *dependencies* by using which we can answer the $(2)^{nd}$ question. Extraction of the 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 above described basic computation of the dependencies would, however, be very imprecise and lead to many false alarms. The imprecision is caused by invalid locksets. The main reasons for imprecision of the locksets are imprecision in dealing with conditionals (all outcomes are considered as possible), function calls (missing context), and aliasing (any aliasing is considered to be possible).

Next, as we mentioned at the beginning of this section, if a function call appears in the analyzed code, we have to apply a summary of the function to the abstract state at the callsite. Given a callee $g$, its lockset $L_g$, unlockset $U_g$, and a 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 `lock4` will be replaced by `lock2` in the summary of $g$.
(2) Update the precondition of $f$:
$if(\exists l : l \in unlocked_g \land l \notin unlockset_f)$
  add lock $l$ to $unlocked_f$
$if(\exists l : l \in locked_g \land l \notin lockset_f)$
  add lock $l$ to $locked_f$
(3) Update $D_f$ by adding new dependencies for all locks in $L_f$ with locks which were locked in $g$.

However, what happens 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. To cope with problem, we have yet another set in the summaries whose semantics is similar to the semantics of the lockset except that the unlock statement does not remove locks 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 problem, we create `unlock`→`lock` type dependencies in the summaries, that can be used to safely determine the order of operations in the callee. This finally ensures that the only newly created correct dependency in our example will be `lock2`→`lock3`.

(4) Update $L_f$: $L_f = (L_f \setminus U_g) \cup L_g$

(5) Update $U_f$: $U_f = (U_f \setminus L_g) \cup U_g$

### 4.2 Experimental Evaluation

We performed our experiments by using a benchmark of 1002 concurrent C programs derived from the Debian GNU/Linux distribution.The entire benchmark is available online at gitlab. These programs were originally used for an experimental evaluation of Daniel Kroening's static deadlock analyser [9] implemented in the CPROVER framework.

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

Both our analyzer and CPROVER correctly report all 8 potential deadlocks in the benchmarks with known issues. A comparison of results for deadlock-free programs can be seen in Table 2.

**Table 2.** Results for programs without a deadlock (t/o — timed out, m/o — out of memory)

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

As one can see, L2D2 reported false alarms for 104 deadlock-free benchmarks which is by 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 518 examples in favor of our analyzer. In case of L2D2, we have 80 compilation errors that were caused by syntax that Infer does not support. The biggest difference between our analyzer and CPROVER is the 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 reducing the number of alarms. The main reason for such alarms is false dependencies. Reasons for their existence were mentioned in Subsection 4.1 ($4^{th}$ paragraph). So, to eliminate false positives, we need some techniques to eliminate false dependencies. In our implementation of L2D2, we use a number of heuristics that try to reduce the imprecision. An example is that if a locking error occurs (double lock acquisition), then L2D2 sets the current lockset to empty, and adds currently acquired lock to the lockset (we can safely tell that this lock is locked), thereby eliminating any dependencies that could result from the locking error. More precise description of these heuristics is beyond the scope of the paper.

## 5. Atomicity Violations Analyzer

In *concurrent programs* there are often *atomicity requirements* for the execution of specific sequences of instructions. Violating these requirements may cause many kinds of problems, such as an 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-dependent programs*, especially in large projects, and even harder and time-consuming is then finding and fixing these errors.

In this section of the paper there is described a proposal and an implementation of a *static analyzer for finding atomicity violations*. In particular, we concentrate on an atomic execution of sequences of function calls, which is often required, e.g., when using certain library calls.

### 5.1 Contracts for Concurrency

The proposal of a solution is based on the concept of *contracts for concurrency* described in [10]. These contracts allow one to define *sequences of functions* that are required to be *executed atomically*. The proposed analyzer itself (**Atomer**) is able to automatically derive candidates for such contracts, and then verify whether the contracts are fulfilled.

In [10], 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`, illustrated in Listing 4. The contract for this specific scenario contains clause $\varrho_1$, which is defined as follows:

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

Clause $\varrho_1$ specifies that every execution of `index_of` followed by an 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 may then 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 [10] there is described a proposal and an implementation for a static validation which is based on *grammars* and *parsing trees*. The authors of [10] implemented a stand-alone prototype tool[1] for analysing programs written in Java, which led to some promising experimental results but the scalability of the tool was still limited. Moreover, the tool from [10] is no more developed. That is why we decided to get inspired by [10] and reimplement the analysis in Facebook Infer redesigning it in accordance with the principles of Infer, which should make it more scalable. Due to adapting the analysis for the context of Infer, its implementation is significantly different in the end as presented in Sections 5.2 and 5.3. Further, unlike [10], the implementation aims at programs written in C/C++ languages using *POSIX Threads (Pthreads)* locks for a *synchronization of concurrent threads*.

In Facebook Infer there is already implemented analysis called *Lock Consistency Violation*[2], which is a part of the *RacerD* [11]. That analysis finds atomicity violations for writes/reads on single variables that are required to be executed atomically. Atomer is different in that 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** Detection of *atomic sequences*, which is described in Section 5.2.

**Phase 2** Detection of *atomicity violations*, which is described in Section 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. **Phase 1** of Atomer is able to produce such contracts, i.e., it detects *sequences of functions* that should be *executed atomically*. Intuitively, the detection is based on looking for sequences of functions that are executed atomically on some path through the program. The assumption is that if it is once needed to execute the sequence atomically, it should probably be always executed atomically.

The detection of the sequences of calls to be executed atomically is based on analysing all paths through the control flow graph of a function and generating all pairs (**A**, **B**) of sets of function calls such that: **A** is a reduced sequence of function calls that appear between the beginning of the function being analysed and the first lock or between an unlock and a subsequent lock, and **B** is a reduced sequence of function calls that follow the calls from **A** and that appear between a lock and an unlock (or the end of the function being analysed). Here, by a reduced sequence we mean a sequence in which the first appearance of each function is recorded only. The reason is to ensure finiteness of the sequences and of the analysis. The summary then consists of (1) the set of all the **B** sequence and (2) the set of all the concatenations **A.B** of the corresponding **A** and **B** sequences. The latter is recorded for the purpose of analysing functions higher in the call hierarchy since the locks/unlocks can appear in such a higher-level function.

For instance, the analysis of the function `g` from Listing 5 (assuming *Pthreads* locks and existence of the initialized global variable `lock` of the type `pthread_mutex_t`) produces the following sequences:

`f1` ~~`f1`~~(`f1` ~~`f1`~~ `f2`)|`f1` ~~`f1`~~(`f1 f3`)|~~`f1`~~(~~`f1 f3 f3`~~)

The strikethrough of the functions `f1` and `f3` denotes a removal of already recorded function calls in the **A** and **B** sequences. The strikethrough of the entire sequence `f1 (f1 f3 f3)` means a discardence of sequences already seen before. The derivated sets for the function `g` are then as follows:

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

**Listing 5.** An example of a code for an illustration of the derivation of sequences of functions called atomically

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

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

```c
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);
}
```

Further, we show how the function h from Listing 6 would be analysed using the result of the analysis of the function g. The result of the analysis of the nested function is used as follows: As we can see, when calling an already analysed function, one plugs all the sequences from the second component of its summary into the current **A** or **B** sequence.

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

The derivated sets for the function h are as follows:

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

**Listing 6.** An example of a code for an illustration of the derivation of sequences of functions called atomically with nested function call

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

The above detection of atomic sequences has been implemented and successfully verified on a set of sample programs created for this purpose. The derived sequences of calls assumed to execute atomically, i.e., the **B** sequences, from the summaries of all analysed functions are stored into a file, which is used during **Phase 2**, described below. There are some possibilities for further extending and improving **Phase 1**, e.g., work with nested locks, distinguish the different locks used (currently, we do not distinguish between the locks at all) or extend detection for other types of locks for a synchronization of concurrent threads/processes. On the other hand, to further enhance the scalability, it seems promising to replace working the **A** and **B** sequence by sets of calls: sacrificing some precision but gaining speed.

### 5.3 Detection of Atomicity Violations

In the second phase of the analysis, i.e., when detecting violations of the detected sequences of calls assumed to execute atomically from **Phase 1**, the set of atomic sequences from **Phase 1** is taken, and the analysis looks for pairs of functions that should be called atomically while this is not the case on some path through the control flow graph.

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, an atomicity violation is detected because functions f2 and f3 are not called atomically (under a lock).

**Listing 7.** Atomicity violation

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

An implementation of this phase and its experimental evaluation is currently in progress. Based on its results, we will then think of fine-tuning **Phase 1** as well.

## 6. Conclusions

In this paper, we presented three analyzers which we implemented in the *Facebook Infer* framework. The *Looper*, resource bounds analyzer, was able to infer the precise bound in 6 out of 8 of the selected examples used for evaluation of the original *Loopus* tool, which inspired our analyser. The *L2D2* analyzer, a deadlock detector in C programs, was evaluated on a benchmark derived from real-life programs from the Debian distribution, which was used for an evaluation of the CPROVER-based deadlock detector presented in [9]. L2D2 handled the benchmark with 100 % success rate in detection of potential deadlocks and roughly 11 % false positives rate. It demonstrated its scalability as it managed to finish the benchmark in less than 1 % of the time needed by the CPROVER tool. The first phase of the*Atomer*, an atomicity violations analyzer, was successfully verified on a set of sample programs created for this purpose. The second phase is a work in the progress.

Our analyzers show potential for further improvements of 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 achieve a merge of our implementations to the `master` branch of the *Facebook Infer repository*[3].

## Acknowledgements

We would like to thank our supervisors and colleagues from VeriFIT: Tomáš Fiedor, Adam Rogalewicz and Tomáš Vojnar. Further, we would like to thank Nikos Gorogiannis and Sam Blackshear from Infer team at Facebook for helpful discussions about the development of our checkers. Lastly, we thank for the support received from the H2020 ECSEL project Aquas.

## References

[1] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. *Compositional Shape Analysis by means of Bi-Abduction. In Proc. of POPL'09*.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*.

[3] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. *Understanding and Detecting Real-World Performance Bugs. In Proc. of PLDI'18*.

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

[5] M.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*. 2011.

[7] J. Lourenço, J. Fiedor, B. Křena, and T. Vojnar. *Discovering Concurrency Errors*.

[8] D. R. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proc. of SOSP'03*.

[9] D. Kroening, D. Poetzl, P. Schrammel, and B. Wachter. Sound static deadlock analysis for c/pthreads. In *Proc. of ASE'16*.

[10] R. Dias, C. Ferreira, J. Fiedor, J. Lourenço, A. Smrčka, D. Sousa, and T. Vojnar. Verifying concurrent programs using contracts. In *Proc. of ICST'17*.

[11] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey. Racerd: Compositional static race detection. *Proc. of OOPSLA'18*.

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