Facebook Infer

Dhruv Maroo

Abstract

In this report, I give a brief overview of Facebook's static analysis tool *Infer*, and how it works. I highlight the underlying separation logic and bi-abduction techniques. Then I enumerate many of the various analyses it can perform and also demonstrate it's analysis on some sample programs and see how it can be used to find bugs in the code. In the end, I conclude with some other similar tools.

Overview

Infer is a static analysis tool developed by Facebook, written in OCaml. It can be used to analyze Cm C++, Java, Objective-C, C#, .NET programs, and provides many different analyzers which can be used to spot bugs or potential bugs in the code. It is open-source and the source code can be found on GitHub at `facebook/infer`. It employs modular analysis techniques, which makes it very efficient for small code changes since it only needs to reanalyze the changes functions. This helps in incremental analysis of the code as well.

Under the hood, it employs mathematical techniques like *separation logic* and *bi-abduction* for analyzing the input programs. However, there is a new implementation of Infer, based on *abstract interpretation*, under development as well, and it's known as *Infer.AI* (whereas the old implementation is now called *Infer.SL*).

Techniques

Separation Logic

Separation logic is an extension of Hoare logic and can be used to reason about programs. It allows us to reason about programs which deal with objects on the heap. Without going into the details, the separation logic's modular power can be demonstrated using the *frame rule*. It is as follows:

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}\ \mathrm{mod}(C) \cap \mathrm{fv}(R) = \emptyset$$

Here P and Q are the pre-conditions and post-conditions, respectively, of the stack and the heap. c is the program being analyzed. $*$ is the *separating conjunction operator*. R is the *frame* which is the part of the heap which is not modified by the program c. Here, the side condition ($\mathrm{mod}(C) \cap \mathrm{fv}(R) = \emptyset$) states that the modified variables of the program c and the free variables of the frame R should be disjoint. This is to ensure that the frame R is not modified by the program c.

The rule states that if we can prove that the program c satisfies the pre-conditions P and post-conditions Q, then it also satisfies the pre-conditions $P * R$ and post-conditions $Q * R$. This is very useful in modular analysis of programs, since we can analyze the program in parts, and then combine the individual results to get the final result.

Bi-abduction

Bi-abduction is a technique which can be used to infer pre-conditions and post-conditions of a program. It accomplishes by finding the *frame* and the *anti-frame* in the frame rule (as described above). The equation used in the bi-abduction is as follows:

$$\text{anti-frame} * R \vdash Q * \text{frame}$$

Finding the correct values of the frame and anti-frame would allow us to infer the pre-conditions and post-conditions of the program. This is done by using a constraint solver which tries to find the values of the frame and anti-frame which satisfy the above equation.

Building

Instructions for building Infer are available on *Getting Started*. All the examples which I'll be using in this report can be found at `DMaroo/infer-examples`. The run instructions are provided in that repository itself.

Infer uses a custom modification of LLVM to analyze C, C++ and Objective-C code. This can be a problem while building from source since that would also build the entire LLVM library from source. Using a pre-built Infer binary is much simpler since it would be faster.

Analyzers

Infer provides many different analyzers which can be used to find bugs in the code. Each of the analyzers have a set of issues which they report. These issues represent various types of bugs. There are analyzers to detect memory issues, concurrency issues, compute complexity, program analysis. Some analyzers require (or use) code annotations for performing their analysis. Annotations in languages like C/C++ (which don't have annotation support) can be done using a config file (in JSON format). Also, some analyzers are only available for a subset of the languages, and they may even be deprecated or unmaintained. I outline some useful analyzers below.

Annotation Reachability

This analyzer checks if any *"expensive"* function is called by any *"performance-critical"* function. We can denote a function as *"expensive"* by annotating it with `@Expensive`. Similarly, we can denote a function as *"performance-critical"* by annotating it with `@PerformanceCritical`. This analyzer can be used to find performance issues in the code. Under the hood, this would require *callgraph* construction to get the program flow.

Supported for C/C++/ObjC, Java, C#/.NET

Example

```java
public class ExpensiveInheritance {
    @PerformanceCritical
    public void critical() {
        Cheap c = new Costly();
        Cheap d = new Cheap();
        c.compute();
        d.compute();
    }
}

class Cheap {
    public void compute() {
    }
}
```

```java

class Costly extends Cheap {
    public void compute() {
        costly();
    }

    @Expensive
    public void costly() {
    }
}
```

Output

```
ExpensiveInheritance.java:9: error: Expensive Method Called
  Method `critical()` annotated with `@PerformanceCritical` calls `Costly.costly()` where
  ↪  `Costly.costly()` is annotated with `@Expensive`.
   4.           Cheap c = new Costly();
   5.           Cheap d = new Cheap();
   6. >         c.compute();
   7.           d.compute();
   8.       }
```

InferBO

This analyzer checks for buffer overflows (hence the "BO" in the name) and out-of-bounds accesses. Under the hood it maintains a range of possible array sizes and checks if the index is within the range. It employs *points-to analysis* to make sure the analysis also works for pointer aliases.

Supported for C/C++/ObjC, Java, C#/.NET

Example

Example 1 (in C)

```c
int main(int argc, char *argv[]) {
        int a[16];

        *(a + 16) = 2;
}
```

Example 2 (in Java)

```java
public class Overwrite {
    public static void main(String[] args) {
        int[] x = new int[16];

        for (int i = 0; i < x.length / 2 + 9; i++) {
            x[i] = i;
        }
    }
}
```

Output

For *example 1*:

```
overwrite.c:4: error: Buffer Overrun L1
  Offset: 16 Size: 16.
  2.     int a[16];
  3.
  4.     *(a + 16) = 2;
         ^
  5. }
```

As you can see in example 1, the buffer overflow can be detected even when we use pointer arithmetic to access the array.

For *example 2*:

```
Overwrite.java:6: error: Buffer Overrun L2
  Offset: [0, 16] Size: 16.
  4.
  5.           for (int i = 0; i < x.length / 2 + 9; i++) {
  6. >             x[i] = i;
  7.           }
  8.       }
```

As we can see in example 2, it is able to detect out-of-bounds accesses when the index is not a constant integer value, but instead an expression.

Runtime Complexity Analysis

This analyzer computes the time-complexity of functions. It estimates an upper bound on the worst-case execution cost of the program. This is done by computing the cost of instructions and how many times they will be executed. These values are then passe to a constraint solver which which computes the execution cost based on incoming/outgoing edges of the nodes. This outputs time complexity polynomials.

Supported for C/C++/ObjC, Java, C#/.NET

Example

```c
unsigned int smallest(const int *a, unsigned int len) {
    unsigned int index = 0;

    for (int i = 0; i < len; i++) {
        if (a[index] > a[i]) {
            index = i;
        }
    }

    return index;
}

void *sort(int *a, unsigned int len) {
    for (int i = 0; i < len; i++) {
        unsigned int min = smallest(a + i, len - i);

        /* swap */
        int temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}


int main(int argc, char *argv[]) {
    int arr[argc];

    sort(arr, argc);
}
```

Output

The output is generated in a JSON file (`infer-out/costs-report.json`), which contains a lot of the data about the whole analysis, including the time complexity polynomials. Grepping for the time complexity in that file, we get the following:

```
===
"procedure_name": "main",
---
    "big_o": "O(argc × argc)"
===
"procedure_name": "smallest",
---
    "big_o": "O(len)"
===
"procedure_name": "sort",
---
    "big_o": "O(len × len)"
===
```

Eradicate

It checks the validity of `@Nullable` annotations. It helps in eradicating null pointer exceptions in well-annotated code. The analyzer performs flow-sensitive analysis to propagate the nullability through assignments and calls, and flags errors for unprotected accesses to nullable values or inconsistent/missing annotations. It can also be used to add annotations to a previously un-annotated program.

Supported for Java, C#/.NET

Example

```java
public class FieldAccess {
    public static void main(String[] args) {
        Fields f = new Fields(0);
        int x = get_first(f);

        Fields g = f.rest;
        int y = get_first(g);
    }

    public static int get_first(@Nullable Fields f) {
        return f.first;
    }
}

class Fields {
    public int first;
    @Nullable
    public Fields rest;

    public Fields(int x) {
        first = x;
        rest = null;
    }
}
```

Output

```
FieldAccess.java:11: warning: Nullable Dereference
  `f` is nullable and is not locally checked for null when accessing field `first`.
   9.
  10.       public static int get_first(@Nullable Fields f) {
  11. >         return f.first;
  12.       }
  13.   }
```

Inefficient `keySet` Iterator

This analyzer checks for inefficient uses of iterators that iterate on keys then lookup their values, instead of iterating on key-value pairs directly. As of now, this analyzer does not work.

Supported for Java, C#/.NET

Liveness

This analyzer detects dead stores and unused variables. It is also a common feature in many modern IDEs.

Supported for C/C++/ObjC

Example

```
int main(int argc, char *argv[]) {
    int g;
    g = 3;
    int x = g;
    int y;
    return g - 3;
}
```

Output

```
dead.cpp:4: error: Dead Store
  The value written to &x (type int) is never used.
  2.     int g;
  3.     g = 3;
  4.     int x = g;
         ^

  5.     int y;
  6.     return g - 3;
```

Loop Hoisting

This analyzer checks for *loop invariant* code which can be moved outside the loop, to make the loop more efficient. It uses purity analysis to determine function purity and find loop invariant code. The analysis does not work as of now (most likely because purity analysis is still experimental).

Supported for C/C++/ObjC, Java, C#/.NET

Pulse

This analyzer performs inter-procedure memory safety and lifetime analysis. The analysis is conservative, i.e. errors only get reported when all conditions on the erroneous path are true regardless of input. This is developed as an replacement for the original bi-abduction analyzer in Infer.

This analyzer can detect constant address dereferences, memory leaks, null pointer dereferencing, use-after-free and other memory and lifetime issues.

Supported for C/C++/ObjC, Java, C#/.NET

Example

```c
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int *a = malloc(16);

    int *b = malloc(16);
    b[10] = 0;
    free(b);

    return 0;
}
```

Output

```
leaky.c:4: error: Memory Leak
  Pulse found a potential memory leak. Memory dynamically allocated at line 4 by call to `malloc`,
  ↪  is not freed after the last access at line 4, column 5.
  2.
  3. int main(int argc, char *argv[]) {
  4.     int *a = malloc(16);
         ^
  5.
  6.     int *b = malloc(16);

leaky.c:7: error: Nullptr Dereference
  Pulse found a potential null pointer dereference  on line 6.
  5.
  6.     int *b = malloc(16);
  7.     b[10] = 0;
         ^
  8.     free(b);
  9.
```

As we can see, two errors are found. One of them is a possible null pointer dereference of `b`, since `b` can be `NULL`, as `malloc` can return `NULL`. And the other is memory leak, since the memory pointed to by `a` is not freed.

Purity

This analyzer detects pure functions. Pure functions are functions which have no side effects, i.e. they do not modify any of their arguments or any global variables. This analyzer performs interprocedural analysis, and keeps track of purity of all the functions. This is used by other analyzers like *Loop Hoisting* and *Runtime Complexity Analysis*.

Purity analysis requires using points-to analysis, but Infer does not have a points-to analysis implementation. So it uses InferBO's points-to analysis implementation, which is not very precise. This makes the purity analysis not entirely precise.

Experimental support for C/C++/ObjC, Java, C#/.NET

RacerD

This analyzer performs thread safety analysis. This analysis does not attempt to prove the absence of concurrency issues, rather, it searches for a high-confidence class of data races. The analysis is inter-procedural and is triggered by either the presence of `synchronized` statements or by `@ThreadSafe` annotation. Since alias-analysis in Infer is not very precise, this analyzer misses race conditions (for aliasing variables) in some cases.

There are many other annotations which can be added to improve and guide the analysis. This analysis can be used to find interfaces which aren't thread-safe but are marked thread-safe. It also detects lock consistency violations and thread safety violations.

Supported for C/C++/ObjC, Java, C#/.NET

Starvation

This analysis detects deadlocking situations, i.e. situations where no progress is begin made because of concurrency bugs. It can detect deadlocks, violation of `@Lockless` violations and other Android specific concurrency issues.

Supported for C/C++/ObjC, Java, C#/.NET

Unintialized Value

This analysis warns when values are used before their initialization. This is already a common feature in modern IDEs as well.

Supported for C/C++/ObjC

Example

```c
int main(int argc, char *argv[]) {
    int a;

    int b = 1;
    b += a;

    return b;
}
```

Output

```
uninit.c:5: error: Uninitialized Value
  The value read from a was never initialized.
  3.
  4.     int b = 1;
  5.     b += a;
         ^
  6.
  7.     return b;
```

Deprecated (but Notable) Analyzers

- **Immutable Cast**: Detects object cast from immutable types to mutable types
- **AST Language (AL)**: Declarative linting framework over the Clang AST
- `printf()` **Argument Types**: Detect mismatches between the Java `printf` format strings and the argument types

Similar Frameworks and Tools

- SonarQube
- Coverity
- ReSharper
- CodeQL
- PVS-Studio

Many of these tools are proprietary. These are commonly used as IDE extensions (like PVS, ReSharper), or used in CI/CD for finding code defects (like Coverity, CodeQL).

Important Links

- Infer website: `fbinfer.com`
- Infer source code: `facebook/infer`
- Separation logic and bi-abduction
- Examples' code: `DMaroo/infer-examples`
- Presentation