8 Analisi Statica

La qualità del software è importante, ma spesso imperfetta nella pratica. Possiamo utilizzare molte tecniche per provare a migliorare la qualità, inclusi testing (visto nei capitoli precedenti), inspezione del codice e specifiche formali. Gli strumenti di analisi statica valutano il software in astratto, senza eseguire il software o considerando un input specifico. Piuttosto che provare a dimostrare che il codice soddisfa le sue specifiche, tali strumenti cercano violazioni di pratiche di programmazione ragionevoli o consigliate. Quindi, cercano i punti in cui il codice può dereferenziare un puntatore nullo o fare our of bounds di un array. Gli strumenti di analisi statica potrebbero anche segnalare un problema come un confronto che non può mai essere vero. Sebbene il confronto non causi un errore o un'eccezione, la sua esistenza suggerisce che potrebbe essere il risultato di un errore di codifica che ha portato a un comportamento scorretto del programma. Alcuni strumenti inoltre segnalano o impongono problemi di stile di programmazione, come le convenzioni di denominazione o l'uso di parentesi graffe nei condizionali e nelle strutture di looping. Il programma LINT per i programmi C è generalmente considerato il primo strumento di analisi statica ampiamente utilizzato per il rilevamento dei difetti, sebbene per gli standard odierni sia piuttosto limitato. I ricercatori hanno svolto un lavoro significativo nell'area nell'ultimo decennio, guidati in modo sostanziale dalle preoccupazioni relative a difetti che portano a vulnerabilità della sicurezza, come buffer overlfow, vulnerabilità delle stringhe di formato, SQL injection e cross-site scripting. Un vivace settore commerciale si è sviluppato attorno a avanzati (e costosi) strumenti di analisi statica, e diverse aziende hanno i propri strumenti interni proprietari, come il PREfix di Microsoft. Molti strumenti commerciali sono sofisticati, utilizzando tecniche di analisi approfondite. Alcuni possono utilizzare o dipendere da annotazioni che descrivono invarianti e altre proprietà del software previsto che gli strumenti non possono facilmente dedurre, come la relazione prevista tra i parametri di funzione.

8.1 Vantaggi e svantaggi dell'analisi statica

8.1.1 Limiti dell'analisi statica

Qual è l'analisi del codice sorgente dal punto di vista pratico? Prendiamo i file sorgente e otteniamo alcune informazioni sulla qualità del sistema in breve tempo (molto più breve di un test eseguito). La limitazione principale e matematicamente insormontabile è che in questo modo possiamo rispondere solo a un numero limitato di domande sul sistema analizzato.

L'esempio più famoso di un compito, non risolvibile mediante l'analisi statica, è il problema dell'halt: c'è un teorema, che dimostra che non è possibile elaborare un algo-

ritmo generale, che definisca se un programma con un dato codice sorgente viene ripetuto per sempre o completato per l'ultima volta L'estensione di questo teorema è un teorema di Rice, asserendo che per qualsiasi proprietà non banale delle funzioni computabili, la questione della determinazione se un dato programma calcola una funzione con questa proprietà è un'attività algoritmicamente irrisolvibile. Ad esempio, non è possibile scrivere un analizzatore, che determina in base al codice sorgente se il programma analizzato è un'implementazione di un algoritmo specifico, ad esempio, uno che calcola la quadratura di un numero intero.

Pertanto, la funzionalità degli analizzatori statici ha limiti insormontabili. L'analizzatore statico non sarà mai in grado di rilevare tutti i casi di, ad esempio, il bug "eccezione puntatore nullo" nei linguaggi tipo C. O rilevare tutte le occorrenze di "attributo non trovato" nei linguaggi iterpretati dinamicamente. Tutto ciò che l'analizzatore statico più perfetto può fare è catturare casi particolari. Il numero di essi tra tutti i possibili problemi con il codice sorgente, senza esagerazione, è una goccia nell'oceano.

Limiti dell'analisi statica

L'analisi statica ha diversi limiti. Vediamo alcuni casi tipici

- Mancanza di conoscenza di quello che il programma dovrebbe fare

  ```
  int calculateArea(int length, int width) {
    return (length + width);
  }
  ```

  Uno strumento di analisi statico può rilevare un possibile overflow in questo calcolo. Ma non può determinare fondamentalmente che la funzione non faccia ciò che ci si aspetta!

- Regole che non sono statisticamente applicabili

  Alcune regole di codifica dipendono dalla documentazione esterna. Oppure sono aperti all'interpretazione soggettiva.

  Per esempio nel CERT-C MSC04 si prescrive: Utilizzare i commenti in modo coerente e leggibile. Come si possono giudicare i commenti leggibile

- Possibili difetti portano a falsi positivi e falsi negativi

  In alcune situazioni, uno strumento può solo segnalare che esiste un possibile difetto.

  ```
  int divide() {
    int x;
    if(foo()) { x = 0; } else { x = 5; }
    return (10/x);
  }
  ```

  Se non sappiamo nulla di foo (), non sappiamo quale valore avrà x.

- In generale il risultato è indecidibile. Ciò significa che gli strumenti possono segnalare difetti che in realtà non esistono. Oppure potrebbero non riuscire a segnalare vizi reali.

8.1.2 L'analisi statica non è una ricerca di bug

Ecco una conclusione che segue da quanto sopra: l'analisi statica non è il modo preferito per ridurre il numero di difetti in un programma. Gli esempi di difetti rilevati automaticamente dagli analizzatori sono impressionanti, ma non dobbiamo dimenticare che questi esempi sono stati trovati analizzando un enorme insieme di basi di codice rispetto all'insieme di regole relativamente semplici. Allo stesso modo gli hacker, avendo la possibilità di provare diverse password semplici su un gran numero di account, alla fine trovano gli account con una semplice password.

Questo significa che non è necessario applicare l'analisi statica? Ovviamente no! Dovrebbe essere applicato per lo stesso motivo per cui potresti voler verificare ogni nuova password nell'elenco delle password non sicure.

8.1.3 L'analisi statica è più della ricerca di bug

In effetti, i compiti che possono essere risolti dall'analisi nella pratica sono molto più ampi. Perché in generale, l'analisi statica rappresenta un controllo del codice sorgente, eseguito prima di eseguirlo. Ecco alcune cose che puoi fare:

- Un controllo dello stile di codifica nel senso più ampio di questa parola. Include sia un controllo della formattazione che una ricerca di parentesi vuota / non necessaria, l'impostazione di valori soglia per metriche come un numero di linee / complessità ciclomatica di un metodo e così via - tutte cose che complicano la leggibilità e la manutenibilità del codice. In Java, Checkstyle rappresenta uno strumento con tale funzionalità, in Python - flake. Tali programmi sono solitamente chiamati "linters".

- Non solo il codice eseguibile può essere analizzato. Risorse come i file JSON, YAML, XML e .properties possono (e devono!) essere verificate automaticamente per la validità. La ragione è che è meglio scoprire che, ad esempio, la struttura JSON è rotta a causa delle quotazioni spaiate nella fase iniziale del controllo automatico di una richiesta pull rispetto all'esecuzione di test o in fase di esecuzione, non è vero? Esistono alcuni strumenti pertinenti, ad esempio YAMLlint, JSONLint e xmllint.

- La compilazione (o l'analisi per i linguaggi di programmazione dinamici) è anche una sorta di analisi statica. Di solito, i compilatori possono emettere avvisi che segnalano problemi con la qualità del codice sorgente e non dovrebbero essere ignorati. A volte la compilazione viene applicata non solo al codice eseguibile. Ad esempio, se si dispone di documentazione nel formato AsciiDoctor, quindi nel processo di compilarlo in HTML / PDF, l'AsciiDoctor (plugin Maven) può emettere avvisi, ad esempio, su collegamenti interni non funzionanti. Questo è un motivo significativo per non accettare una richiesta di pull con modifiche alla documentazione.

- Il controllo ortografico è anche una sorta di analisi statica. L'utility aspell è in grado di controllare l'ortografia non solo nella documentazione, ma anche nel codice sorgente dei programmi (commenti e letterali) in vari linguaggi di programmazione, inclusi C / C ++, Java e Python. Anche l'errore di ortografia nell'interfaccia utente o nella documentazione è un difetto!

- I test di configurazione rappresentano in realtà una forma di analisi statica, poiché non eseguono il codice sorgente durante il processo di esecuzione, anche se i test di configurazione vengono eseguiti come test di unità pytest.

Come possiamo vedere, la ricerca di bug ha il ruolo meno significativo in questo elenco e tutto il resto è disponibile quando si utilizzano strumenti open source gratuiti.

Quale di questi tipi di analisi statiche dovrebbe essere utilizzato nel tuo progetto? Certo, più è meglio! Ciò che è importante qui è una corretta implementazione, che sarà discussa ulteriormente.

8.2 Classificazione dei risultati dell'analisi statica

Analisi statica è più che la ricerca di bugs

Uno strumento di analisi statico S analizza il codice sorgente di un programma P per determinare se soddisfa una proprietà φ (P è negativo) o la viola (P risulta positivo). S accetta i programmi che soddisfano φ, ma potrebbe sbagliare in due modi:

- Se S è **sound**, (pessimistic o conservativo) se accetta un programma P, sicuramente soddisfa φ
  - Accetta solo programmi che soddisfano la proprietà.
    - Nessun **falso negativo**
  - Può rifiutare i programmi che soddisfano la proprietà.
    - Possibile falso positivo (sembra che ci sia un difetto ma non c'è).

- Se S è completo (ottimistico) se P soddisfa φ allora lo accetta
  - Accetta sempre programmi che soddisfano la proprietà
    - nessun falso positivo
  - Può accettare programmi che non soddisfano la proprietà.
    - Possibile falso negativo (sembra che non abbia il difetto ma ce l'ha - passono inosservati)

- Troppi falsi positivi fanno perdere tempo al programmatore che quindi smetterà di usare questa tecnica

8.3 Tools per l'analisi statica

Ci sono tantissimi tool anche commerciali per l'analisi statica. Eccone alcuni:

8.4 findbugs

What is findBugs?

- Static analysis tool to find defects in Java code
  - not a style checker
- Can find hundreds of defects in each of large apps such as Bea WebLogic, IBM Websphere, Sun's JDK
- real defects, stuff that should be fixed
- Doesn't focus on security
- lower tolerance for false positives

Common Wisdom about Bugs

- Programmers are smart
- Smart people don't make dumb mistakes
- We have good techniques (e.g., unit testing, pair programming, code inspections) for finding bugs early
- So, bugs remaining in production code must be subtle, and require sophisticated techniques to find

Would You Write Code Like This?

- `if (in == null) try { in.close(); ...`
  - Oops
  - This code is from Eclipse (versions 3.0 - 3.2)
- You may be surprised what is lurking in your code

Why Do Bugs Occur?

- Nobody is perfect
- Common types of errors:
  - Misunderstood language features, API methods
  - Typos (using wrong boolean operator, forgetting parentheses or brackets, etc.)
  - Misunderstood class or method invariants
- Everyone makes syntax errors, but the compiler catches them
  - What about bugs one step removed from a syntax error?

Infinite recursive loop

8.4.1 Bug Patterns

```
/** Construct a WebSpider */
public WebSpider() { WebSpider w = new WebSpider(); }
```

- Double check against JDK: Found 4 infinite recursive loops
- Including one written by Joshua Bloch

```
public String foundType() { return this.foundType(); }
```

- Smart people make dumb mistakes
- Embrace and fix your dumb mistakes

Hashcode/Equals

- Equal objects must have equal hash codes
- Programmers sometimes override equals() but not hashCode()
- Or, override hashCode() but not equals()
- Objects violating the contract won't work in hash tables, maps, sets
  - Examples (53 bugs in 1.6.0-b29)
    - javax.management.Attribute
    - java.awt.geom.Area

Fixing hashCode

- What if you want to define equals, but don't think your objects will ever get put into a HashTable?
- Suggestion:

```
public int hashCode() {
  assert false : "hashCode␣method␣not␣designed";
  return 0;
}
```

- Or throw a RuntimeException

Null Pointer Dereference

- Dereferencing a null value results in NullPointerException
- Warn if there is a statement or branch that if executed, guarantees a NPE
- Example:

```
// Eclipse 3.0.0M8
Control c = getControl();
if (c == null && c.isDisposed()) return;

// Eclipse 3.0.0M8
String sig = type.getSignature();
if (sig != null || sig.length() == 1) {
  return sig;
}
// JDK 1.5 build 42
if (name != null || name.length > 0) { ..
```

More Null Pointer Dereferences

```
javax.security.auth.kerberos.KerberosTicket
// flags is a parameter
// this.flags is a field
if (flags != null) {
  if (flags.length >= NUM_FLAGS)
    this.flags = ...
  else
    this.flags = ...
} else
  this.flags = ...
  if (flags[RENEWABLE_TICKET_FLAG]) {
```

Redundant Null Comparison

- Comparing a reference to null when it is definitely null or definitely non-null
- Not harmful per se, but often indicates an inconsistency that might be a bug
- Example (JBoss 4.0.0DR3):

```
protected Node findNode(Fqn fqn, ...) {
  int treeNodeSize = fqn.size(); ...
  if (fqn == null) return null;
```

How should we fix this bug?

```
if (name != null || name.length > 0)
```

Should we just change it to

```
if (name != null && name.length > 0)
```

- Will that fix it?
- We have no idea.
- Obviously, we've never tested the situation when name is null.
- Try to write a test case first, then apply the obvious fix

Bad Binary operations

- Binary operators can be used as bitwise operators

```
if ((f.getStyle () & Font.BOLD) == 1) {
  sbuf.append ("<b>"); isBold = true;
}
if ((f.getStyle () & Font.ITALIC) == 1) {
  sbuf.append ("<i>"); isItalic = true;
}
```

What is the meaning? True? Or really 1?

Doomed Equals

- Equals that will never be true

```
public static final ASDDVersion getASDDVersion(BigDecimal version){
  if(SUN_APPSERVER_7_0.toString().equals(version))
    return SUN_APPSERVER_7_0;
...
```

Unintended regular expression

- The use of regular expression?

```
String [] valueSegments = value.split ("."); // NOI18N
```

field Self Assignment

```
public TagHelpItem(String name, String file , String startText ,
  int startOffset , String endText, int endOffset,
  String textBefore , String textAfter ){

  this.name = name;
  this. file = file ;
  this. startText = startText;
  this. startTextOffset = startTextOffset ;
  this.endText = endText;
  this.endTextOffset = endTextOffset;
  this. textBefore = textBefore;
  this. textAfter = textAfter;
  this. identical = null;
}
```

Bad Naming

- Wrong capitalization

```
package org.eclipse.jface.dialogs;
public abstract
class Dialog extends Window {
  protected Button getOKButton() {
    return getButton(IDialogConstants.OK_ID);
  };
}
public class InputDialog extends Dialog {
  protected Button getOkButton() {
    return okButton;
  };
}
```

Confusing/bad naming

- Methods with identical names and signatures – but different capitalization of names – could mean you don't override method in superclass – confusing in general
- Method name same as class name – gets confused with constructor

Ignored return values

- Lots of methods for which return value always should be checked – E.g., operations on immutable objects

```
// Eclipse 3.0.0M8
String name= workingCopy.getName();
name.replace(/, .);
```

Ignored Exception Creation

```
/**
* javax.management.ObjectInstance
* reference impl., version 1.2.1 **/
public ObjectInstance(ObjectName objectName, String className) {
  if (objectName.isPattern()) {
    new RuntimeOperationsException(
    new IllegalArgumentException( " Invalid name->"+ objectName.toString()));
  }
  this.name = objectName;
  this.className = className;
}
```

- Inconsistent Synchronization
- Common idiom for thread safe classes is to synchronize on the receiver object ("this")
- We look for field accesses – find classes where lock on "this" is sometimes, but not always, held – Unsynchronized accesses, if reachable from multiple threads, constitute a race condition

Inconsistent Synchronization Example

- GNU Classpath 0.08,

```
java.util.Vector
public int lastIndexOf(Object elem) {
  return lastIndexOf(elem, elementCount 1);
}
public synchronized int lastIndexOf( Object e, int index) { ... }
```

Unconditional Wait

- Before waiting on a monitor, the condition should be almost always be checked
  - Waiting unconditionally almost always a bug
  - If condition checked without lock held, could miss the notification
  - Example (JBoss 4.0.0DR3):

```
if (!enabled) {
  try { log.debug(...);
    synchronized (lock) { lock.wait(); }
```

  - condition can become true after it is checked but before the wait occurs

Bug Categories

- Correctness
- Bad Practice
  - equals without hashCode
  - comparing Strings with ==,
  - equals should handle null argument
- Performance
- Multithreaded correctness
- Malicious code vulnerability

8.5 Altri tools

- PMD

PMD - Java Rules

- Best Practices
- Code Style
- Design
- Documentation
- Error Prone
- Multithreading

- Performance
- Security
- SolarLint https://www.sonarlint.org/eclipse/