



# **RacerX: effective, static detection of race conditions and deadlock**

---

basato su lavoro di  
Dawson Engler e Ken Ashcraft  
Stanford University



# Cos'è RacerX?

---

- Difficile trovare race conditions e deadlock perché dipendono da sequenze intricate di eventi che hanno bassa probabilità di verificarsi
- Tool statico che utilizza flow-sensitive, interprocedural analysis su Control Flow Graph (CFG) per trovare race conditions e deadlock



# Altri tool

---

- Dinamici

- Esecuzione del programma e individuazione dei lock posseduti
- Vantaggio: path che vengono realmente eseguiti
- Svantaggio: alti costi computazionali

- Statici

- Non eseguono il codice
- Vantaggio: verificano programmi più grandi
- Svantaggio: meno precisi e analizzano path che possono non verificarsi mai



# Come funziona RacerX?

---

1. elencare le funzioni di locking
2. estrarre un control flow graph (CFG)
3. eseguire RacerX sul CFG
4. classificare gli errori
5. ispezione



# Come funziona RacerX

---

## Fase 1

- L'utente deve specificare quali sono le funzioni utilizzate per acquisire o rilasciare i lock



# Come funziona RacerX

---

## Fase 2

- Fase di estrazione: viene costruito un CFG per ogni file del sistema controllato



# Come funziona RacerX

---

## Fase 3

- Fase di analisi: per ogni radice del CFG viene eseguita una visita in profondità di tipo flow-sensitive e intraprocedural e vengono calcolati gli insiemi di lock (lockset) e vengono memorizzati nella statement cache. Vengono memorizzati nella summary cache i lockset che produce ogni funzione.



# Come funziona RacerX

---

## Fase 4

- Classificazione degli errori: in base a due metriche
  - La probabilità che sia un false positive
  - La difficoltà nel trovare l'errore



# Come funziona RacerX

---

## Fase 5

- Ispezione: viene effettuata dall'utente



# Lockset

---

- È l'insieme dei lock attualmente tenuti
  - per ogni radice del CFG viene calcolato il lockset ad ogni statement  $s$ 
    - $initial \rightarrow \text{lockset} = \{ \}$
    - $lock(I) \rightarrow \text{lockset} = \text{lockset} \cup \{ I \}$
    - $unlock(I) \rightarrow \text{lockset} = \text{lockset} - \{ I \}$
- Statement cashe
- Summary cashe

# Esempio (1)

```
connect() {  
→ lock(a);  
→ open_conn();  
  send();  
}
```

{ a }  
{ a }

{ a }

summary:  
{ a } → ?

```
open_conn() {  
→ if (x)           { a }  
→   lock(b);       { a, b }  
→ else             { a }  
→   lock(c);       { a, c }  
→ }                { a, b } { a, c }
```

# Esempio (2)

```
connect() {  
  lock(a);           { a }  
  open_conn();      { a }  
  send();           { a, b }, { a, c }  
}
```

summary:

{ a } → { a, b }, { a, c }

{ a, b }, { a, c }

```
open_conn() {  
  if (x)           { a }  
    lock(b);       { a, b }  
  else             { a }  
    lock(c);       { a, c }  
}
```

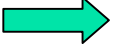
{ a, b }, { a, c }



# Deadlock checking

---

## 1. Estrazione dei vincoli fra i lock

lock(a);  
lock(b);          "a→b"

## 2. Soluzione dei vincoli

chiusura transitiva e segnalazione cicli ("a→b→a")

## 3. Classificazione

Lock globali rispetto a lock locali

Profondità delle call chain

Numero di thread coinvolti



# False positive

---

Creazione di dipendenze non valide

cause ↓

- Problema del rendezvous
- Release-on-block (se il thread si blocca viene rilasciato il lock)
- Lockset non validi



# Il problema del rendezvous

---

Semafori utilizzati per implementare le dipendenze di scheduling



Belief analysis per distinguere quando un semaforo è utilizzato come lock o come signal-wait



## Release-on-block

---

Il lock viene rilasciato nel momento in cui il thread si blocca



Evitare di emettere un vincolo quando un thread possiede un lock di tipo release-on-block e prova ad acquisire un altro lock



# Locksets non validi

---

- Analisi sugli unlockset
- Summay selection



## Analisi sugli unlockset

---

Nel lockset di una statement  $s$  viene eliminato ogni lock / se non esiste in una statement successiva  $s'$ , raggiungibile da  $s$ , un unlock /

**initial** → unlockset = { }

**lock(I)** → unlockset = unlockset - { I }

**unlock(I)** → unlockset = unlockset U { I }

$s.\text{unlockset} = s.\text{unlockset} \cup \text{unlockset}$

**lockset = intersect(s.unlockset, lockset);**



# Summary selection

---

- Majority summary selection: fra tutti i lockset di uscita di una funzione viene preso il lockset che compare più spesso
- Minimum-size summary selection: fra tutti i lockset di uscita di una funzione si prende quello più piccolo



# Race detection

---

1. Simple checking: errore se accesso con lockset vuoto
2. Simple statistical: deduce quali variabili e funzioni devono essere protette
3. Precise statistical: deduce quale lock protegge un accesso e segnala l'errore se si accede senza avere quel lock



# Cosa è importante sapere

---

- Se il lockset è valido
- Se il codice è multithread
- Se una variabile deve essere protetta



Il lockset è valido?

---

Strategie viste nel deadlock checking



# Il codice è multithread?

---

- Multithread inference

RacerX marca una funzione come multithread se vi sono operazioni concorrenti al suo interno

- Annotazioni scritte dal programmatore

L'utente marca le funzioni multithread



# Una variabile $x$ ha bisogno di essere protetta da un lock?

---

Analisi statistica:

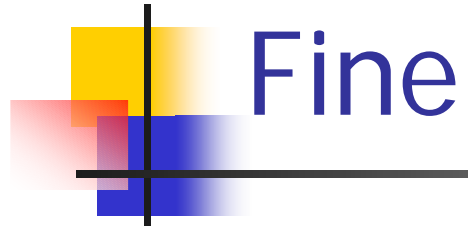
rapporto tra il numero di volte che la variabile è acceduta tramite l'utilizzo di un lock ( $s$ ) e il numero di volte in cui la variabile è acceduta senza alcun lock ( $f$ )



# Limitazioni di RacerX

---

- I puntatori a variabili locali o parametri sono rappresentati dal loro tipo e non dal loro nome  
"struct foo \*f" → `<struct:foo:local>`
- Funzioni chiamate più volte ma con diversi lockset: numerosi calcoli (no > 100 lockset)
- Deadlock: solo se coinvolgono un numero fissato di thread
- Analisi statica: tutti i path in CFG, anche quelli che non verranno mai eseguiti



Fine

---

Grazie per l'attenzione