Overflow Checking in Firefox

Brian Hackett

Goal

- Can we clean a code base of buffer overflows?
  - Keep it clean?
  - Must prove buffer accesses are in bounds
- Verification: prove a code base has a property

Sixgill

- Verifier for buffer accesses in large code bases
  - Note: not quite full verification
- Mostly automatic
  - Can be supplemented with annotations
- Linux: 89% of accesses checked automatically
- Firefox: ditto for 82%
- Firefox javascript engine: 92% checked using annotations

Sixgill (cont)

- Early stages of deployment on Firefox
  - Open source
  - More (not much more) at sixgill.org
- Rest of this lecture
  - Design questions addressed in building Sixgill
  - Sixgill design and architecture
  - Demo!

Verifier Design Questions

- What properties can be checked?
- What level of precision?
- What degree of scalability?
- How are annotations used?
- Can the tool make assumptions?
- Design for clear reports
  - Great majority will be false positives

Sixgill: Properties

- Check properties expressible as assertions
  - Buffer overflows
  - Hand-written 'assert()' failures
  - NULL dereferences
  - Integer overflows
  - ...
- Most properties need customization
  - buf[i] = 0; ⟹ assert(i < ubound(buf));

Sixgill: Precision

- Understand any quantifier-free assertion
  - No loops, no recursion
  - Quantifiers are very hard to reason about
- Understand loop-free pieces of code exactly
  - Use abstractions at function/loop boundaries
- Some technical limitations to these
  - More later

Sixgill: Scalability

- Analyze systems of any size
  - Should parallelize, avoid memory constraints
  - Linux, Firefox: 2-7 MLOC
- Verifiers with comparable power: 5-10 KLOC

Sixgill: Annotations

- Infer information without user input
  - Be robust, deterministic against code changes
- Use annotations when inference breaks down
  - Target: one annotation per 1-3 KLOC
  - Must be clear where to add annotations

Sixgill: Assumptions

- Make some basic assumptions
  - Compiler, hardware behave correctly
  - Program is memory safe, type safe
  - These are made by almost all verifiers
- Make some additional assumptions
  - No integer overflow, heap stability properties, ...
  - More later
- Eventual target is full verification

Why is code correct?

- Buffer accesses are correct for a reason
  - preconditions, postconditions, loop invariants, ...
  - Follow from each other and the code semantics
- Analysis goal: find these reasons
- Reasons follow patterns
  - Use inference for the common patterns
  - Use annotations for the rest

Example

```
void foo(int len)
{
    char *buf = malloc(len);
    bar(buf, len);
}

void bar(char *buf,
{
    for (int i = 0; i < len; i++)
        buf[i] = 0;
}
```

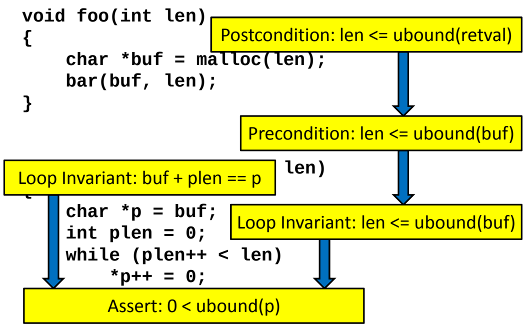
Postcondition: len <= ubound(retval)

Precondition: len <= ubound(buf)

Loop Invariant: len <= ubound(buf)

Assert: i < ubound(buf)

Another Example



Program Facts

- A *fact* is a condition which holds in the program
  - Precondition(foo, *b*)
  - Postcondition(foo, *b*)
  - LoopInvariant(foo, loop, *b*)
  - TypeInvariant(type, *b*)
  - GlobInvariant(*b*)
  - Assert(foo, point, *b*)
- *b* values are quantifier free boolean formulas

Following Facts

- A *goal* fact *f* can follow from zero or more *dependent* facts f_0, f_1, f_2, …
  - If the dependents hold, the goal holds
- Show this using a *memory model*
  - Exact model of a loop free piece of code
    - Note: not quite exact
  - Inject assumes for f_0, f_1, f_2, …
  - Inject asserts for *f*

Memory Example

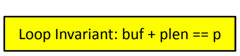
```
void bar(char *buf, int len)
{
    char *p = buf;
    int plen = 0;
    while (plen++ < len)
        *p++ = 0;
}
```

```
bar:
 p = buf;
 plen = 0;
 invoke(loop);

loop:
 if (plen++ < len) {
   *p++ = 0;
   invoke(loop);
 }
```

Memory Example (cont)

```
void bar(char *buf, int len)
{
    char *p = buf;
    int plen = 0;
    while (plen++ < len)
        *p++ = 0;
}
```

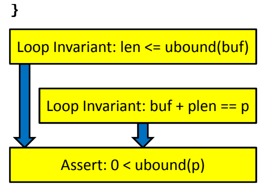



```
bar:
 p = buf;
 plen = 0;
 assert(buf + plen == p)
 invoke(loop);

loop:
 assume(buf + plen == p)
 if (plen++ < len) {
   *p++ = 0;
   assert(buf + plen == p)
   invoke(loop);
 }
```

Memory Example (cont)

```
void bar(char *buf, int len)
{
    char *p = buf;
    int plen = 0;
    while (plen++ < len)
        *p++ = 0;
}
```




```
bar:
 p = buf;
 plen = 0;
 invoke(loop);

loop:
 assume(buf + plen == p)
 assume(len <= ub(buf))
 if (plen++ < len) {
   assert(0 < ub(p))
   *p++ = 0;
   invoke(loop);
 }
```

Memory Model details

- Memory model built on an SMT solver
  - Solves boolean formulas over linear equations
  - We use Yices (from SRI International)
- Solver can't handle nonlinear arithmetic
- Memory model introduces unsoundness

Nonlinear Arithmetic

- Major gap in analysis precision
- Mostly fixable using approximations …
  - (a & b) ⟹ (a & b <= a) && (a & b <= b)
- … but not always

```
int *buf = calloc(width, height * sizeof(int));
int *pos = buf;
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++)
       *pos++ = 0;
}
```

Memory Model unsoundness

- Does not consider integer overflow

```
int *buf = malloc(len * sizeof(int));
for (int i = 0; i < len; i++)
  buf[i] = 0;
```

- Assumes null terminators not overwritten

```
char buf[100];
strcpy(buf, str);
clobber(buf);
int len = strlen(buf);
```

- These can be handled with separate analyses

Analysis

- Start with a goal fact *f*
  - A buffer access or an intermediate fact
- Generate *candidate* sets F_0, F_1, F_2, …
- Test if each candidate *F* is *sufficient* --- *f* follows from the dependent facts in *F*
- Pick a sufficient set and recurse on each dependent

Candidates

```
void bar(char *buf, int len)
{
    for (int i = 0; i < len; i++)
        buf[i] = 0;
}
```

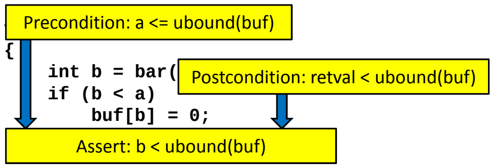
- Target: Assert: i < ubound(buf)
- See compare 'i < len'
- Guess: Loop Invariant: len <= ubound(buf)
- Also guess: Loop Invariant: ubound(buf) <= len

Candidates (cont)

```
void bar(char *buf, int len)
{
    char *p = buf;
    int plen = 0;
    while (plen++ < len)
        *p++ = 0;
}
```

- Target: Assert: 0 < ubound(p)
- See increments of *plen* and *p*
- Initial values of *plen* and *p* are 0 and *buf*
- Guess: Loop Invariant: buf + plen == p
- See compare 'plen < len'
- Add to guess: Loop Invariant: len <= ubound(buf)

Sufficient Choices



- No way to tell which is better
  - Pick one arbitrarily
  - What if we pick wrong?

Annotations

- Annotations are facts which have been specified as holding by a user
  - Assume all annotations when testing candidates
- Untrusted annotations: separately try to prove the annotation holds
  - Same procedure as for buffer accesses

Buffer Write Categories

- Verified
  - proved automatically
- Annotatable
  - provable using untrusted annotations
- Inexpressible
  - Unprovable, but dependent facts can be annotated
  - Limitations of tool
- Unverifiable
  - Dependent facts cannot be annotated
  - Includes all bugs

Results

- Linux 2.6.17.1
  - 55676 buffer writes total
  - All but 6088 verified (89%)
- Firefox 1.9.1
  - 16511 buffer writes total
  - All but 2936 verified (82%)
- More trivially verifiable writes in Linux

```
int buf[10];
buf[9] = 3;
```

Results (cont)

- Detailed results for Firefox javascript engine
- 2801 buffer writes (17% of all of Firefox)
- All but 566 verified (80%)
- 344 annotatable
  - Requiring 64 annotations
- 98 inexpressible
- 124 unverifiable
  - 9 look buggy (not confirmed yet)

Demo

- Tool UI can be used to:
  - Browse and inspect reports
  - Add annotations
  - Reanalyze accesses using added annotations
- Reports are chains of dependents from a buffer access
  - Tool gave up on trying to prove the dependents
- Firefox reports online at sixgill.org