Scaling CQUAL to millions of lines of code and millions of users

Jeff Foster, Rob Johnson, John Kodumal and David Wagner

{jfoster,rtjohnso,jkodumal,daw}@cs.berkeley.edu.

UC Berkeley

Overview

- Applications in the Linux kernel
- CQUAL in the real world
- Getting “buy-in” from developers

User pointers in the Linux kernel

- User programs pass pointers to the kernel as syscall arguments
- Malicious programs may pass invalid pointers
  - Pointers to unmapped memory
  - Pointers to kernel memory
- Kernel must always check user pointers before dereferencing them
  - Corrupt kernel memory
  - Read kernel memory
  - Elevate privileges
  - Crash system
- `copy_{to,from}_user` do sanity checks and copies

User-kernel: GOOD!

User code

```
int main ()
{
  struct foo *p;
  ...
  ioctl (fd, SIOCGFOO, p);
  ...
}
```

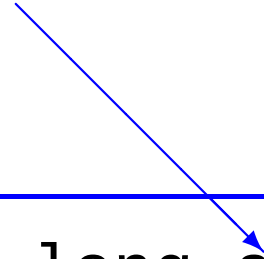

Kernel code

```
int dev_ioctl (int cmd, long arg)
{
  struct foo *q;
  ...
  copy_to_user (arg, q, n);
  ...
}
```

User-kernel: BAD!

User code

```
int main ()
{
  struct foo *p;
  ...
  ioctl (fd, SIOCGFOO, p);
  ...
}
```



Kernel code

```
int dev_ioctl (int cmd, long arg)
{
  struct foo *q;
  ...
  memcpy (arg, q, n);
  ...
}
```

User-kernel: Type qualifiers

```
int dev_ioctl (int cmd, long $user arg)
{
  struct foo * $kernel q;
  ...
  memcpy (arg, q, n);
  ...
}
```

- Annotate everything from user-space as `$user`
- Only allow dereferencing of `$kernel` pointers
- Use type qualifier inference

User-kernel: Results

- Run file-by-file on Linux kernel
- Found 2 new bugs
- Found many (20-40) bugs that were already fixed
- About 200 false positives

`__init` functions and data

- Linux places some kernel functions and data in special "`__init`" sections
- `__init` sections are deleted after kernel initialization
- Thus non-`__init` functions must not
  - call `__init` functions
  - dereference pointers to `__init` data
- `__init` functions may use non-`__init` functions and data

__init: GOOD!

```
int y __init;

void dev_reset(int *x)
{
  *x = 0;
}

void dev_init() __init
{
  dev_reset(&y);
}
```

__init: BAD!

```
int y __init;

void dev_reset(void)
{
  y = 0;
}

void dev_init() __init
{
  dev_reset();
}
```

__init: Effect qualifiers

```
int y $init;

void dev_reset(void) $noninit
{
  y = 0;
}

void dev_init() $init
{
  dev_reset();
}
```

- Model sections as effects
- Perform effect inference

`__init`: Results

- Run file-by-file on Linux kernel
- Found 2 functions which could be declared `__init`
- About 6 false positives

Integrating with Linux build process

- Easier for CQUAL than MOPS
  - MOPS inherently whole-program analysis
  - CQUAL can do whole-program or file-by-file
  - Annotations can make file-by-file analysis sound
- Linux 2.6 Makefile has hooks for file-by-file checkers
  - `make C=1 CHECK=kqual bzImage`
  - `$CHECK` called with same args as `gcc`
- `kqual` drop-in replacement for Linus' Sparse
  - Run `gcc` as preprocessor
  - Run CQUAL on results

Whole-program vs. file-by-file

- Advantages of whole-program analysis
  - Fewer annotations
  - Soundness
- Advantages of file-by-file analysis
  - More annotations (programmers like them!)
  - Can be sound
  - Easy (don't have to emulate `cc1`, `ld`, etc.)
  - Supports incremental recompilation
  - Whole-program analysis impractical for large programs

Other checkers

- Sparse (Linus)
  - Only checks, no inference
  - Requires lots of casts
  - Supposed to be sound, but apparently has bugs
  - CQUAL found bugs in code that Sparse passed
- MECA (Stanford)
  - Very precise (flow-sensitive, path-sensitive)
  - Unsound
  - CQUAL found bugs in code that MECA passed
- H.U.M.A.N.S.
  - Very precise
  - Sound, but buggy
  - CQUAL found bugs in code that humans had audited

How CQUAL got (a little) street cred

- We found bugs that
  - Were real
  - Were exploitable
  - Were non-obvious
  - Were missed by all other tools / manual audits
- Explained why other tools missed these bugs
- Showed interest in working with developers
- Got lucky (Greg KH)

Lessons

- Developers want tools
- Developers like annotations
- Tools should work the way developers work
- Soundness sells
- Get credibility by finding bugs

Type qualifiers

- Idea: decorate language's built-in types with qualifiers
- E.g.

  ref (ref (int))

  becomes

  α *ref (*β *ref (*γ *int))*

- Perform type inference on qualifiers to find solutions for α, β, and γ
- CQUAL is a type qualifier inference engine for C
  - Reduces program to constraint graph
  - Uses CFL-reachability to achieve context-sensitivity

Working with C: int/pointer casts

```
int *x;
int *w;
int y;
int z;
y = (int)x;
z = y;
w = (int*)z;
```

x ref (x' int)
w ref (w' int)
y int
z int

Inferred constraints:

- $x \leq y \leq z \leq w$
- What about x' and y'?

Working with C: int/pointer casts

```
int *x;
int *w;
int y;
int z;
y = (int)x;
z = y;
w = (int*)z;
```

x ref (x' int)
w ref (w' int)
y int
z int

Inferred constraints:

- $x \leq y \leq z \leq w$
- A hack: $x' = y$, $z = w'$

Working with C: int/pointer casts

```
int *x;
int *w;
int y;
int z;
y = (int)x;
z = y;
w = (int*)z;
```

x ref (x' int)
w ref (w' int)
y int
z int

Inferred constraints:

- $x \leq y \leq z \leq w$
- A hack: $x' = y$, $z = w'$
- So $x' \leq w'$ (WRONG! should be $x' = w'$)
- Also causes lots of imprecision

Working with C: int/pointer casts

```
int *x;
int *w;
int y;
int z;
y = (int)x;
z = y;
w = (int*)z;
```

x ref (x' int)
w ref (w' int)
y int (y' void)
z int (z' void)

Inferred constraints:

- $x \leq y \leq z \leq w$
- $x' = y' = z' = w'$
- Sound and more precise