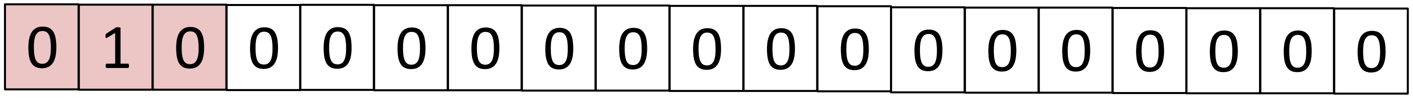
Improving Integer Security for Systems with KINT

Xi Wang, Haogang Chen, Zhihao Jia,
Nickolai Zeldovich, Frans Kaashoek
MIT CSAIL   Tsinghua IIIS

Integer error

- Expected result goes out of bounds
  - Math (∞-bit): $2^{30} \times 2^{3} = 2^{33}$
  - Machine (32-bit): $2^{30} \times 2^{3} = 0$



0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Can be exploited by attackers

Example: buffer overflow

- Array allocation
- `malloc(n * size)`
  - Overflow: $2^{30} \times 2^3 = 0$
  - Smaller buffer than expected

- Memory corruption
  - Privilege escalation
  - iPhone jailbreak (CVE-2011-0226)

Example: logical bug

- Linux kernel OOM killer (CVE-2011-4097)
  - Compute "memory usage score" for each process
  - Kill process with the highest score

- Score: **`nr_pages * 1000`** `/ nr_totalpages`
- Malicious process
  - Consume too much memory => a low score
  - Trick the kernel into killing innocent process

An emerging threat

- 2007 CVE survey:

  “Integer overflows, barely in the top 10 overall in the past few years, are **number 2** for OS vendor advisories, behind buffer overflows.”

- 2010 – early 2011 CVE survey: Linux kernel

  More than **1/3** of [serious bugs] are integer errors.

Hard to prevent integer errors

- Arbitrary-precision integers (Python/Ruby)
  - Performance: require dynamic storage
  - Their implementations (in C) have/had overflows
- Trap on every overflow
  - False positives: overflow checks intentionally incur overflow
  - Linux kernel requires overflow to boot up
- Memory-safe languages (C#/Java)
  - Performance concerns: runtime checks
  - Not enough: integer errors show up in logical bugs

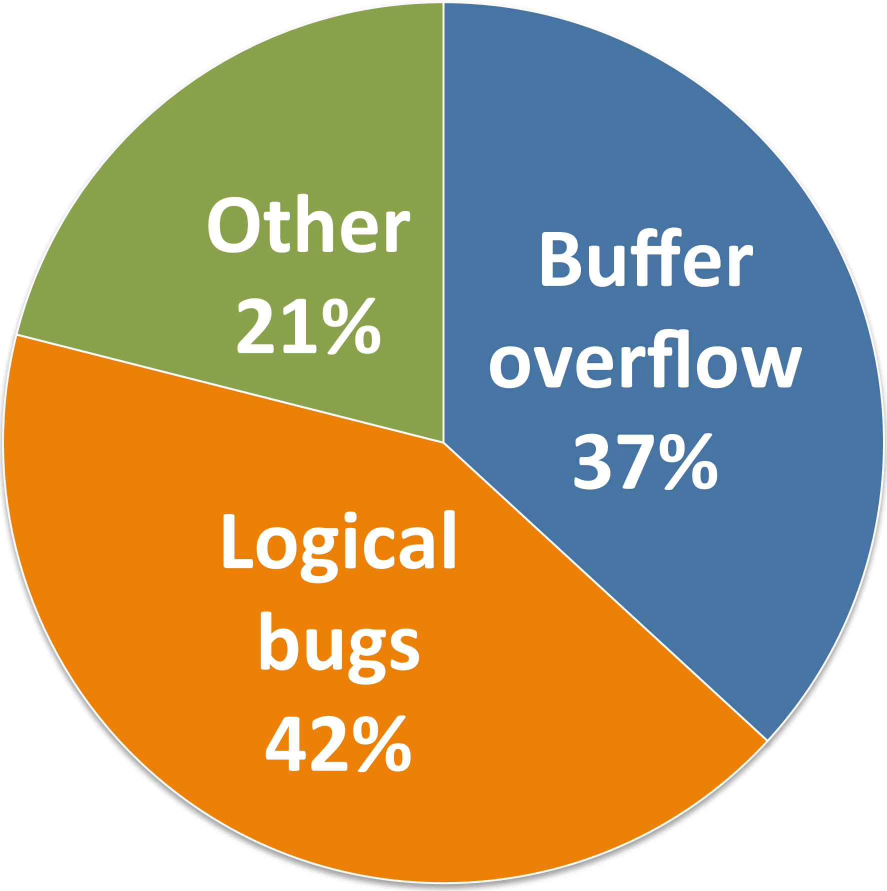
Contributions

- A case study of 114 bugs in the Linux kernel
- **KINT**: a static analysis tool for C programs
  - Used to find the 114 bugs
- **kmalloc_array**: overflow-aware allocation API
- **NaN integer**: automated overflow checking

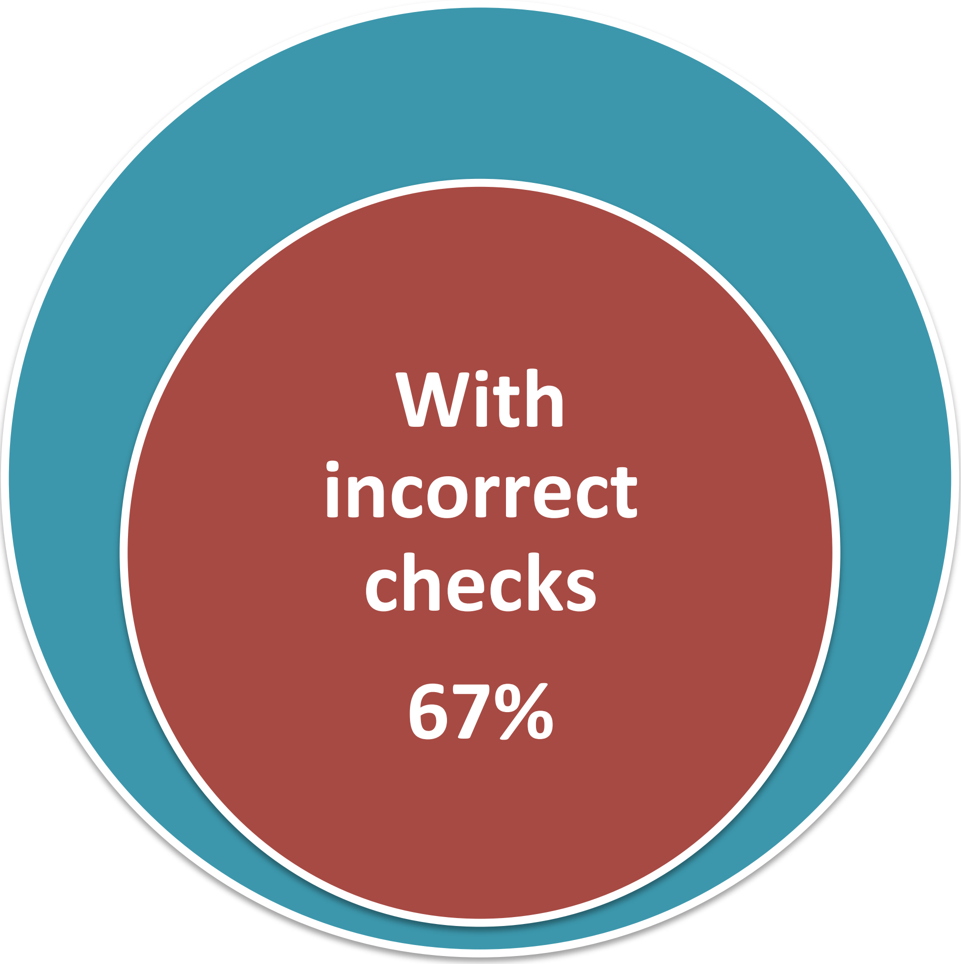
Case study: Linux kernel

- Applied KINT to Linux kernel source code
  - Nov 2011 to Apr 2012
  - Inspect KINT's bug reports & submit patches
- 114 bugs found by KINT
  - confirmed and fixed by developers
  - 105 exclusively found by KINT
  - 9 simultaneously found by other developers
- Incomplete: more to be discovered
  - No manpower to inspect all bug reports

Most are memory and logic bugs



2/3 of bugs have checks



With incorrect checks

67%

Example: wrong bounds

net/core/net-sysfs.c

```
struct flow_table {
    ...
    struct flow entries[0];
};
```

...
entries[0]
entries[…]
entries[n-1]

```
unsigned long n = /* from user space */;
if (n > 1<<30) return -EINVAL;
table = vmalloc(sizeof(struct flow_table) +
                n * sizeof(struct flow));
for (i = 0; i < n; ++i)
    table->entries[i] = …;
```

2^{30} × 8 (2^3) = 0

Example: wrong type

drivers/gpu/drm/vmwgfx/vmwgfx_kms.c

```
u32 pitch = /* from user space*/;
u32 height = /* from user space */;
```

Patch 1:

```
u32 size = pitch * height;
if (size > vram_size) return;
```

32-bit mul overflow

Patch 2: use 64 bits?

```
u64 size = pitch * height;
if (size > vram_size) return;
```

C spec: still 32-bit mul!

Patch 3: convert pitch and height to u64 first!

```
u64 size = (u64)pitch * (u64)height;
if (size > vram_size) return;
```

Writing correct checks is non-trivial

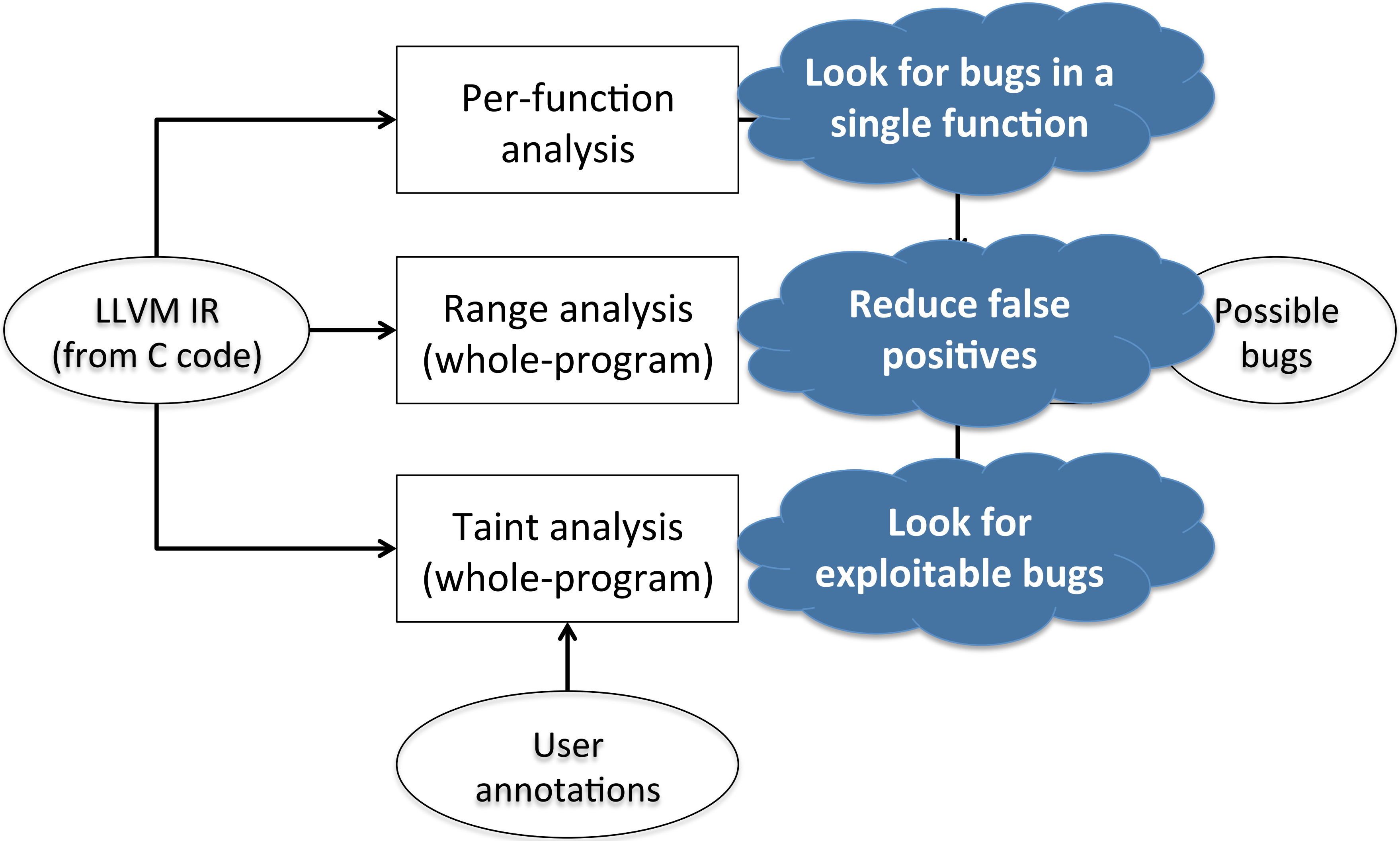
- 2/3 of the 114 integer errors have checks
- One check was fixed 3 times and still buggy
- Even two CVE cases were fixed incorrectly
  - Each received extensive review

- How do we find integer errors?

Finding integer errors

- Random testing
  - Low coverage: hard to trigger corner cases
- Symbolic model checking
  - Path explosion
  - Environment modeling
- KINT: static analysis for bug detection

KINT Overview



LLVM IR
(from C code)

Per-function
analysis

Range analysis
(whole-program)

Taint analysis
(whole-program)

User
annotations

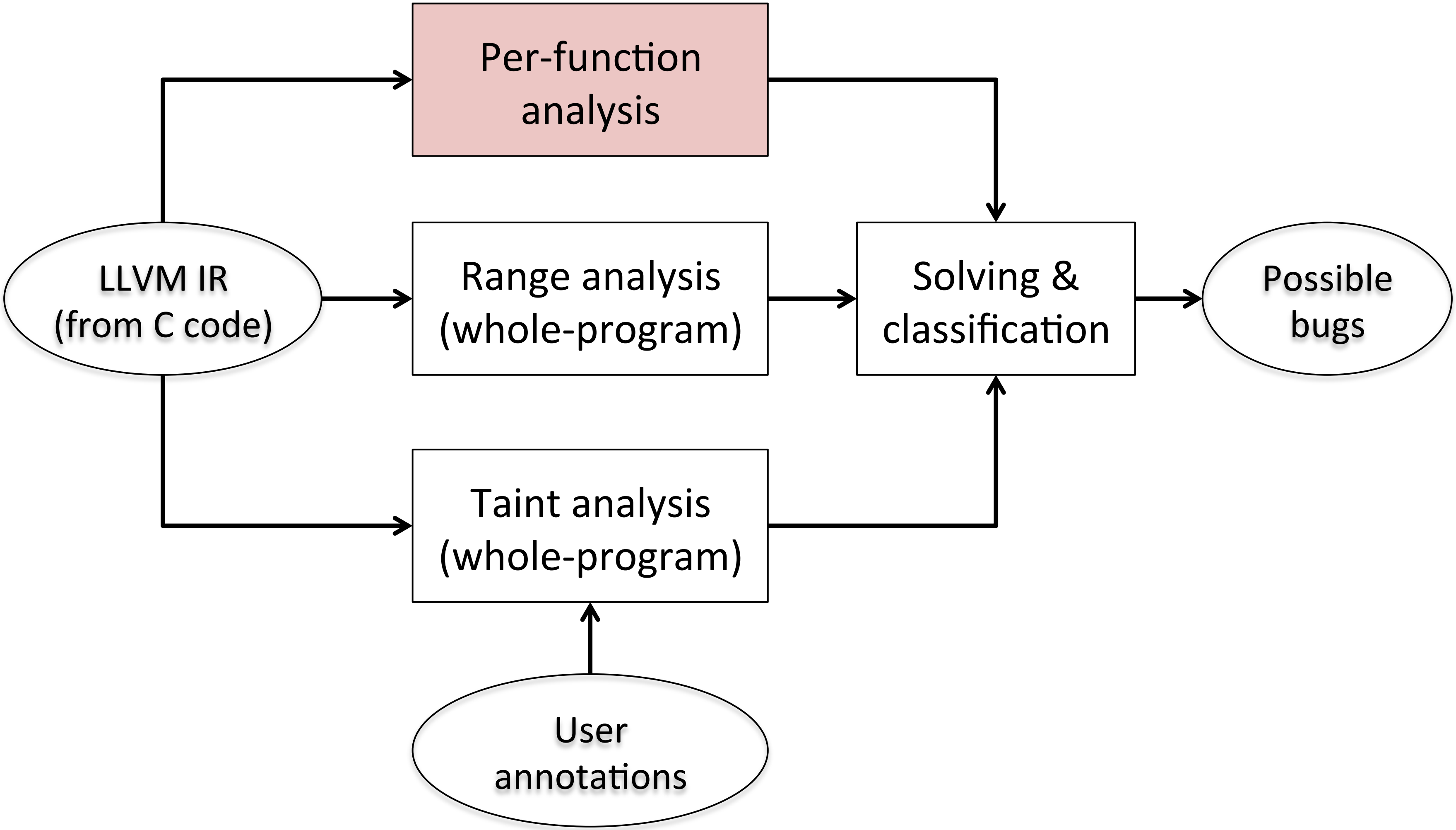
Possible
bugs

Look for bugs in a single function

Reduce false positives

Look for exploitable bugs

KINT Overview



Per-function analysis

LLVM IR (from C code)

Range analysis (whole-program)

Solving & classification

Possible bugs

Taint analysis (whole-program)

User annotations

Per-function analysis

```
int foo(unsigned long n)
{
        if (n > 1<<30) return -EINVAL;
    void *p = vmalloc(n * 8);
    ...
}
```

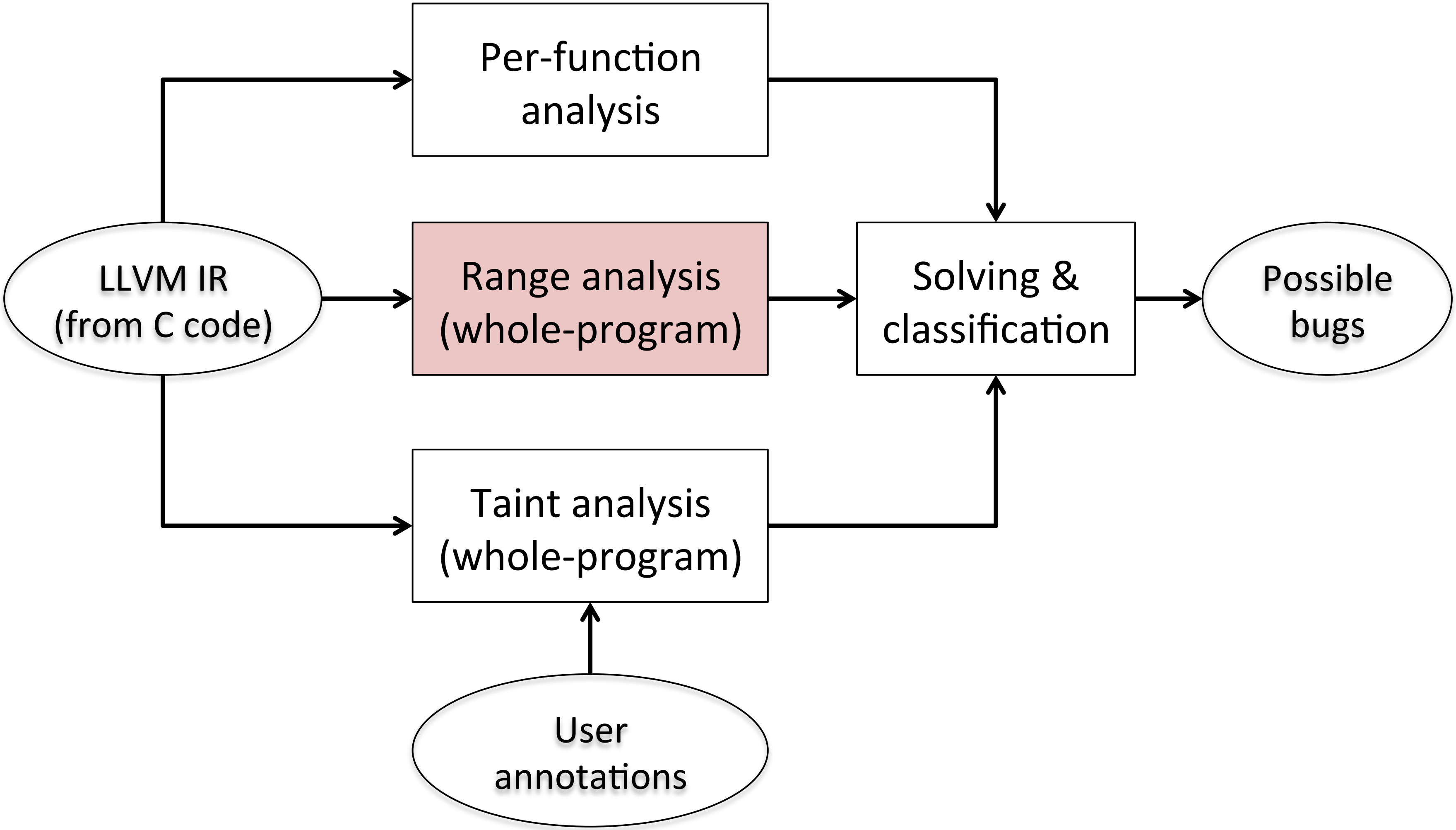
- Under what condition will n * 8 overflow?
  - *Overflow condition*: **n > MAX / 8**
- Under what condition will n * 8 execute?
  - Bypass existing check "`if (n > 1<<30)`"
  - *Path condition*: **n ≤ 1<<30**

Solving boolean constraints

```
int foo(unsigned long n)
{
        if (n > 1<<30) return -EINVAL;
    void *p = vmalloc(n * 8);
    ...
}
```

- Symbolic query: combine overflow & path conditions
  - (n > MAX / 8) AND (n ≤ 1<<30)
- Constraint solver: n = 1<<30
  - KINT: a possible bug

KINT Overview



LLVM IR
(from C code)

Per-function
analysis

Range analysis
(whole-program)

Taint analysis
(whole-program)

User
annotations

Solving &
classification

Possible
bugs

Checks in caller

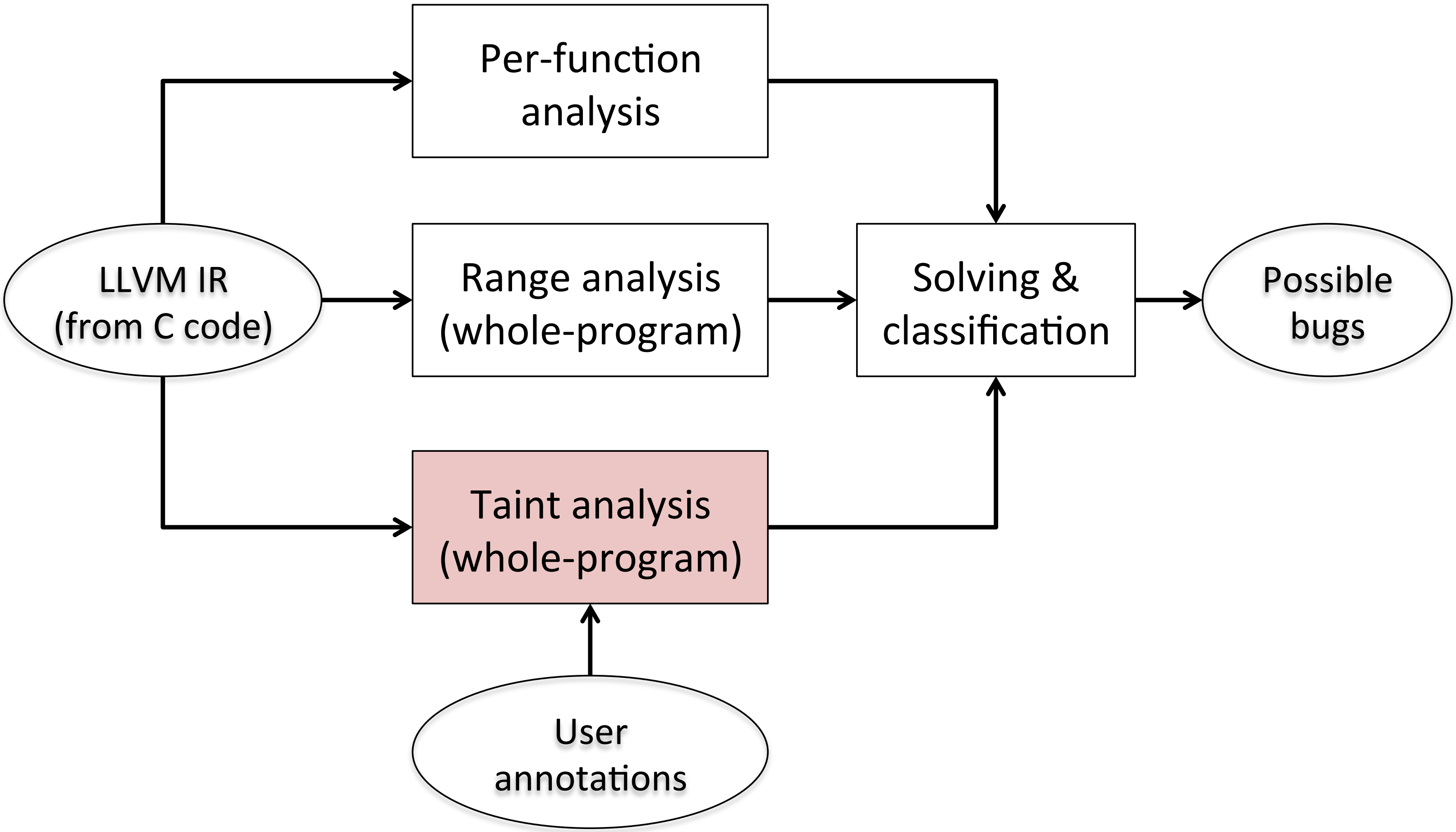
```
int foo(unsigned long n)
{
    if (n > 1<<30) return -EINVAL;
    void *p = vmalloc(n * 8);
    ...
}
void bar()
{
    if (x >= 0 && x <= 100)
           foo(x);
}
```

- n in [0, 100]
  - n * 8 cannot overflow

A whole-program range analysis

- Goals
  - Reduce false positives
  - Scale to large programs with many functions
- Use two constants as bounds for each variable
  - Example: n in [0, 100]
  - Simpler to solve than overflow & path conditions
- Iteratively propagate ranges across functions

KINT Overview



Taint analysis for bug classification

- Users can provide annotations to classify bugs
  - Optional
- Users annotate untrusted input
  - Example: `copy_from_user()`
  - KINT propagates and labels bugs derived from untrusted input
- Users annotate sensitive sinks
  - Example: `kmalloc()` size
  - KINT labels overflowed values as allocation size

KINT Implementation

- LLVM compiler framework
- Boolector constraint solver

KINT usage

```
$ make CC=kint-gcc         # generate LLVM IR *.ll
$ kint-range-taint *.ll    # whole program
$ kint-checker *.ll        # solving & classifying bugs
```

```
==========================================
Unsigned multiplication overflow (32-bit)
fs/xfs/xfs_acl.c:199:3
Untrusted source: struct.posix_acl.a_count
Sensitive sink:   allocation size
==========================================
```

Evaluation

- Effectiveness in finding new bugs
- False negatives (missed errors)
- False positives (not real errors)
- Time to analyze Linux kernel

KINT finds new bugs

- 114 in the Linux kernel shown in case study
- 5 in OpenSSH
- 1 in the lighttpd web server
- All confirmed and fixed

KINT finds most known integer errors

- Test case: all 37 CVE integer bugs in past 3 yrs
  - Excluding those found by ourselves using KINT

- KINT found 36 out of 37 bugs
  - 1 missing: overflow happens due to loops
  - KINT unrolls loops once for path condition

False positives (CVE)

- Test case: patches for 37 CVE bugs (past 3 yrs)
- Assumption: patched code is correct

- KINT reports 1 false error (out of 37)
- Also found 2 incorrect fixes in CVE
  - Useful for validating patches

False positives (whole kernel)

- Linux kernel 3.4-rc1 in April 2012
- 125,172 possible bugs in total
- 741 ranked as “risky”
  - Allocation size computed from untrusted input
- Skimmed the 741 bugs in 5 hours
- Found 11 real bugs
- We don’t know if the rest are real bugs

KINT analysis time

- Linux 3.4-rc1: 8,915 C files
- 6 CPU cores (w/ 2x SMT)
- Total time: 3 hours

Summary of finding bugs with KINT

- 100+ bugs in real-world systems
  - Linux kernel, OpenSSH, lighttpd
- Could have many more bugs
  - Difficult to inspect all possible bugs

- How to mitigate integer errors?

Mitigating allocation size overflow

- kmalloc(n * size)
  - Frequently used in the Linux kernel
  - Can lead to buffer overflow

- **kmalloc_array**(n, size)
  - Return `NULL` if n * size overflows
  - Since Linux 3.4-rc1

Generalized approach: NaN integer

- Semantics
  - Special “NaN” value: Not-A-Number
  - Any overflow results in NaN
  - Any operation with NaN results in NaN
- Easy to check for overflow
  - Check if final result is NaN
- Implementation: modified Clang C compiler
  - Negligible overhead on x86: FLAGS register checks

Verbose manual check (had 3 bugs)

```
size_t symsz     = /* input */;
size_t nr_events = /* input */;
size_t histsz, totalsz;
if (symsz > (SIZE_MAX - sizeof(struct hist)) / sizeof(u64))
    return -1;
histsz = sizeof(struct hist) + symsz * sizeof(u64);
if (histsz > (SIZE_MAX - sizeof(void *)) / nr_events)
    return -1;
totalsz = sizeof(void *) + nr_events * histsz;
void *p = malloc(totalsz);
if (p == NULL)
    return -1;
```

NaN integer example

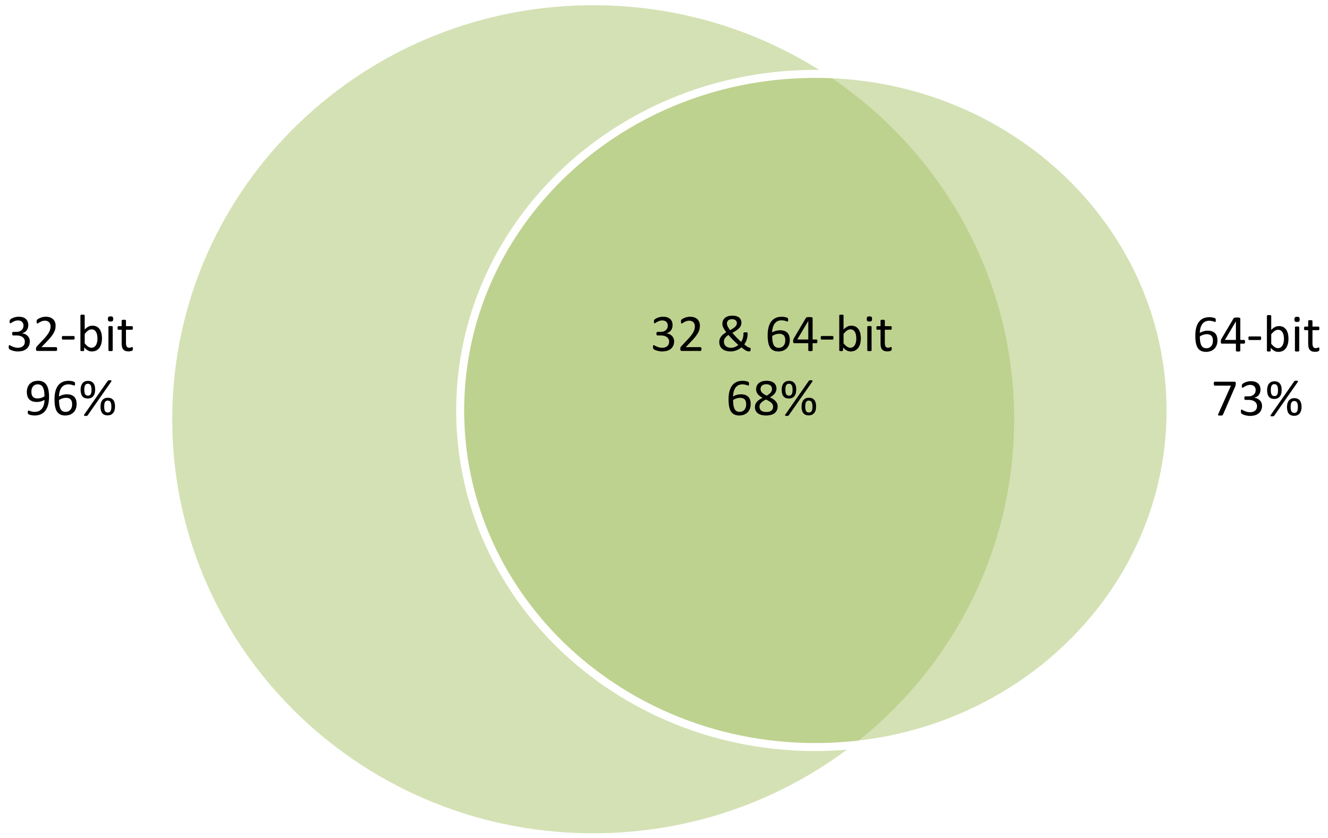
```
nan size_t symsz     = /* input */;
nan size_t nr_events = /* input */;
nan size_t histsz, totalsz;
if (symsz > (SIZE_MAX - sizeof(struct hist)) / sizeof(u64))
    return -1;
histsz = sizeof(struct hist) + symsz * sizeof(u64);
if (histsz > (SIZE_MAX - sizeof(void *)) / nr_events)
    return -1;
totalsz = sizeof(void *) + nr_events * histsz;
void *p = malloc(totalsz);
if (p == NULL)
    return -1;
```

```
void *malloc(nan size_t size)
{
    if (isnan(size)) return NULL;
    return libc_malloc((size_t)size);
}
```

Conclusion

- Case study of integer errors in the Linux kernel
  - Writing correct checks is non-trivial
- KINT: static detection of integer errors for C
  - Scalable analysis based on constraint solving
  - 100+ bugs confirmed and fixed upstream
- `kmalloc_array`: safe array allocation
- NaN integer: automated bounds checking
- http://pdos.csail.mit.edu/kint/

Shifting to 64-bit systems helps a little

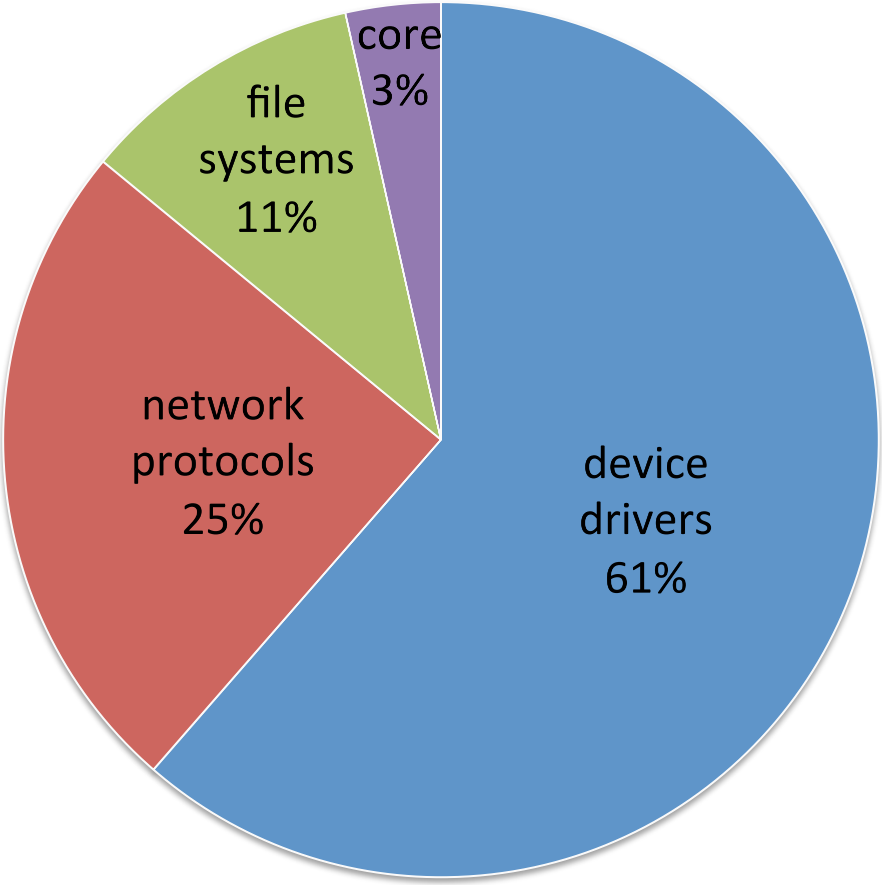


32-bit
96%

32 & 64-bit
68%

64-bit
73%

Bugs found in major components



Example: undefined behavior in ext4

```
log_groups_per_flex = /* from disk */;
groups_per_flex = 1 << log_groups_per_flex;
if (groups_per_flex == 0)
    return 1;
... = ... / groups_per_flex;
```

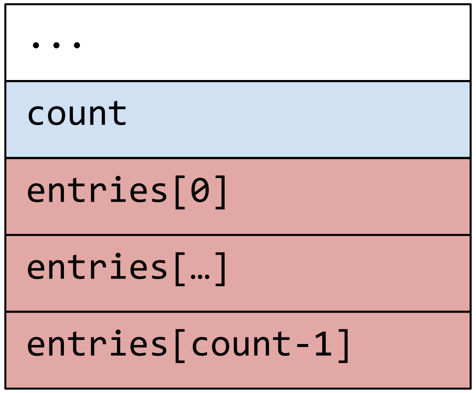
1 << 32 = 0

PowerPC	• 1 << 32 = 0
x86	• 1 << 32 = 1
C/C++	• Undefined behavior

CVE-2012-0038: XFS

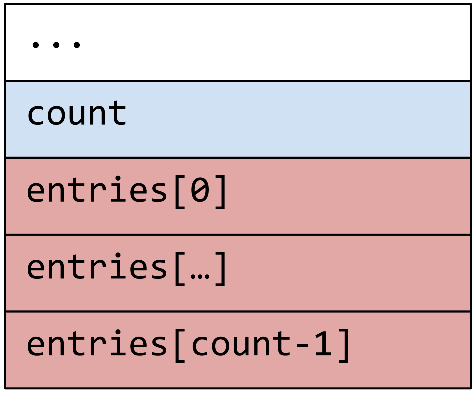
```
/* Access control list */
struct posix_acl { ...
  unsigned int            count;
  struct posix_acl_entry  entries[0];
};

struct posix_acl *acl;
int count = /* read from disk */;
acl = kmalloc(sizeof(struct posix_acl) +
              count * sizeof(struct posix_acl_entry), GFP_KERNEL);
acl->count = count;
for (i = 0; i < acl->count; ++i) {
    /* write to acl->entries[i] */
}
```



CVE-2012-0038: XFS

```
/* Access control list */
struct posix_acl { ...
  unsigned int            count;
  struct posix_acl_entry  entries[0];
};
```



```
...
count
entries[0]
entries[…]
entries[count-1]
```

```
struct posix_acl *acl;
int count = 0x80000000;
acl = kmalloc(sizeof(struct posix_acl) +
              count * sizeof(struct posix_acl_entry), GFP_KERNEL);
acl->count = count;
for (i = 0; i < acl->count; ++i) {
    /* write to acl->entries[i] */
}
```

$$2^{31} \times 8\ (2^3) = 0$$

Fixing integer error is non-trivial

count < 0

0x80000000

```
struct posix_acl *acl;
int count = /* read from disk */;
if (count > 25)
    return ERR_PTR(-EFSCORRUPTED);
acl = kmalloc(sizeof(struct posix_acl) +
              count * sizeof(struct posix_acl_entry), GFP_KERNEL);
acl->count = count;
/* ... */
for (i = 0; i < acl->count; ++i) {
    /* write to acl->entries[i] */
}
```

NaN checks faster than manual

- Experiment: safely allocate n*size bytes

- Manual multiply overflow check: 21-25 cycles
- NaN multiply overflow check: 1-3 cycles
  - Compiler emits code to use hardware overflow flag

Example bad fix: CVE-2008-3526 (sctp)

`key_len > INT_MAX – sizeof(…)`

```
/* u32 key_len */
if (INT_MAX - key_len < sizeof(struct sctp_auth_bytes))
    return NULL;
key = kmalloc(sizeof(struct sctp_auth_bytes) + key_len, gfp);
```

- `key_len = 0xffffffff (UINT_MAX)`
- LHS: negative?
- C 101: unsigned type promotion
- KINT: LHS is large positive 2^{31}

Broken error handling example

```
/* drivers:media */
uchar i2c_read_demod_bytes(…) {
    if (…)
            return -EIO;
}

int err = i2c_read_demod_bytes(…);
if (err < 0)
    return err;
```