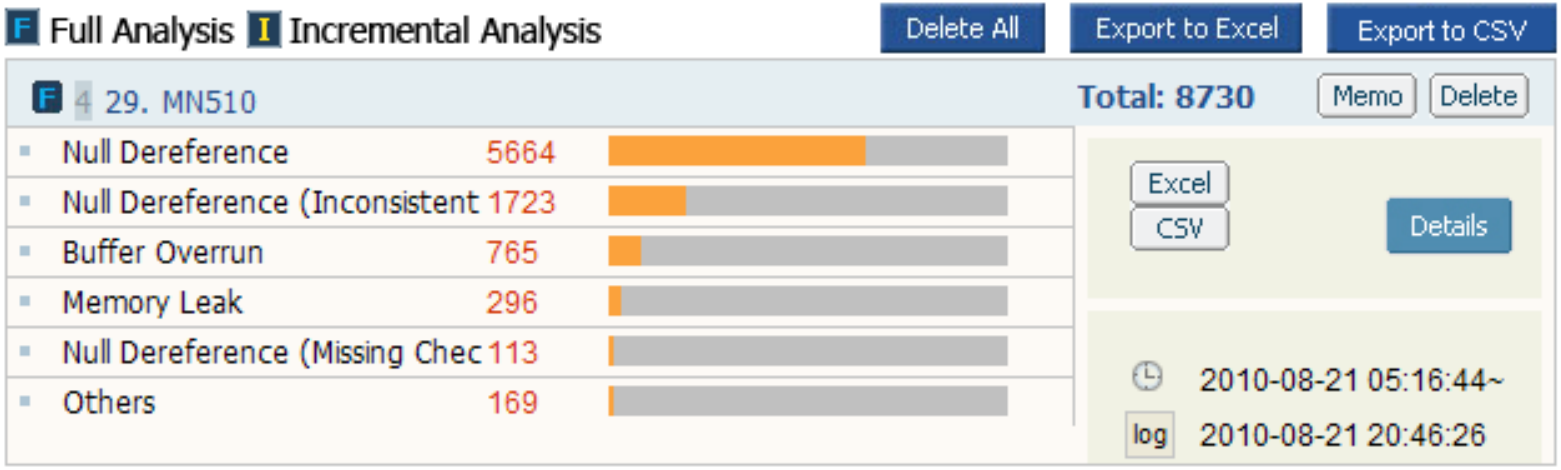
경로민감분석에서 알람군집구하기
Alarm clustering in path-sensitive analysis

Woosuk Lee (ROPAS, Seoul National University)
wslee@ropas.snu.ac.kr

Motivation

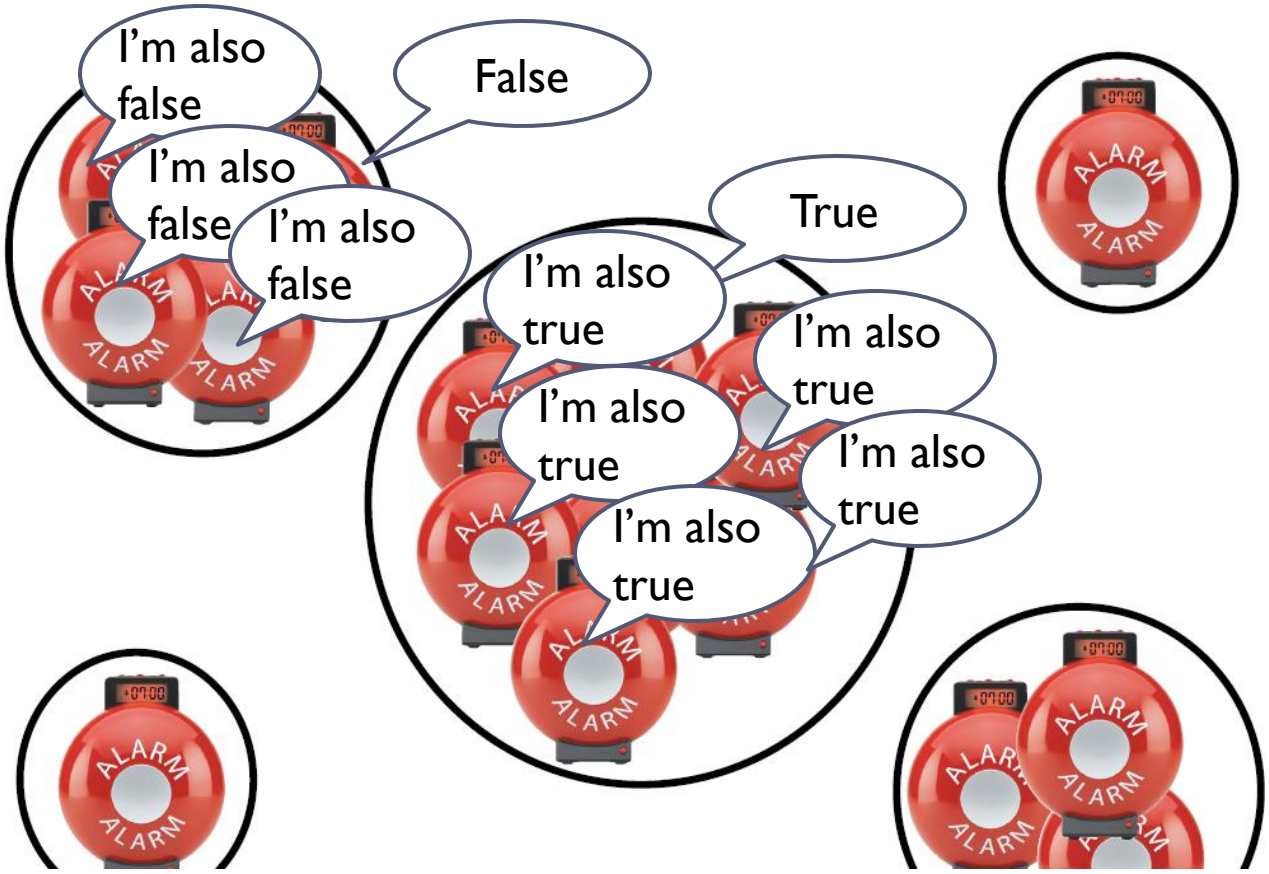
- Static analyzer produces overwhelmingly large number of alarms.



- Many of them look similar and share common causes.

- To relieve users' pain, grouping those similar alarms is needed.

- To relieve users' pain, grouping those similar alarms is needed.



Goal

- General solution
  - Not stick to specific kinds of errors
    - (Just for memory leak, buffer overrun, etc.)
  - Not stick to specific patterns of programs.

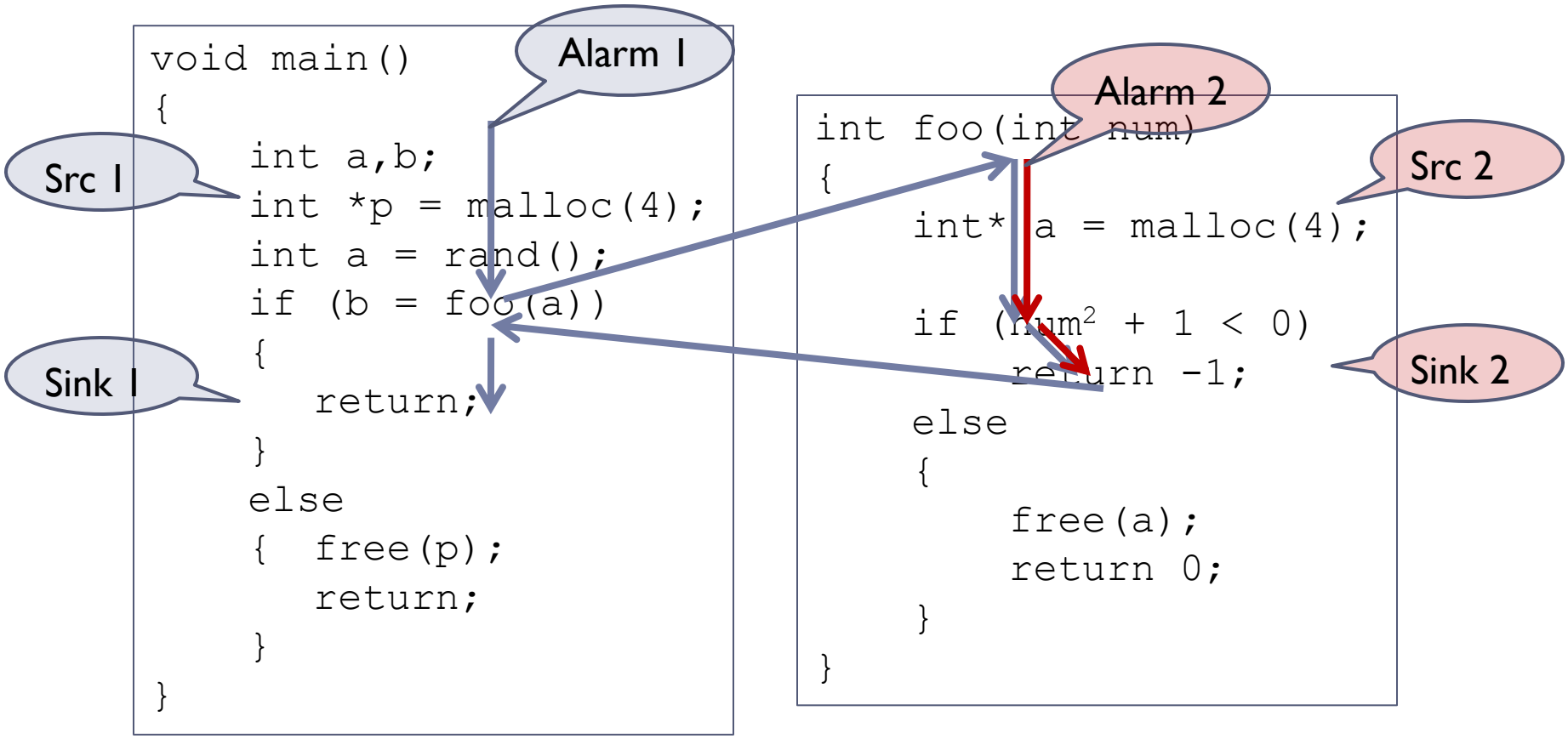
- Not based on statistical approach, but on logical reasoning.
  - It is certain that there's **no need to inspect another alarms in the same group**.

Analyzer

- Opus1
  - Commercialized static analyzer (Fasoo.com)
  - Path-sensitive
  - Procedural summary based
  - Mixed up with abstract interpretation and intuitive and practical approach (not sound)
  - Good performance
    - 300,000 LOC within 30mins.
    - Similar false alarm ratio with Klockwork (4hrs).

Solution at a glance

- Just compare paths of alarms



- Path : Ordered set of

  <Guard, Bool, Reason>

  - Ex) <b != 0, False, PRUNED>

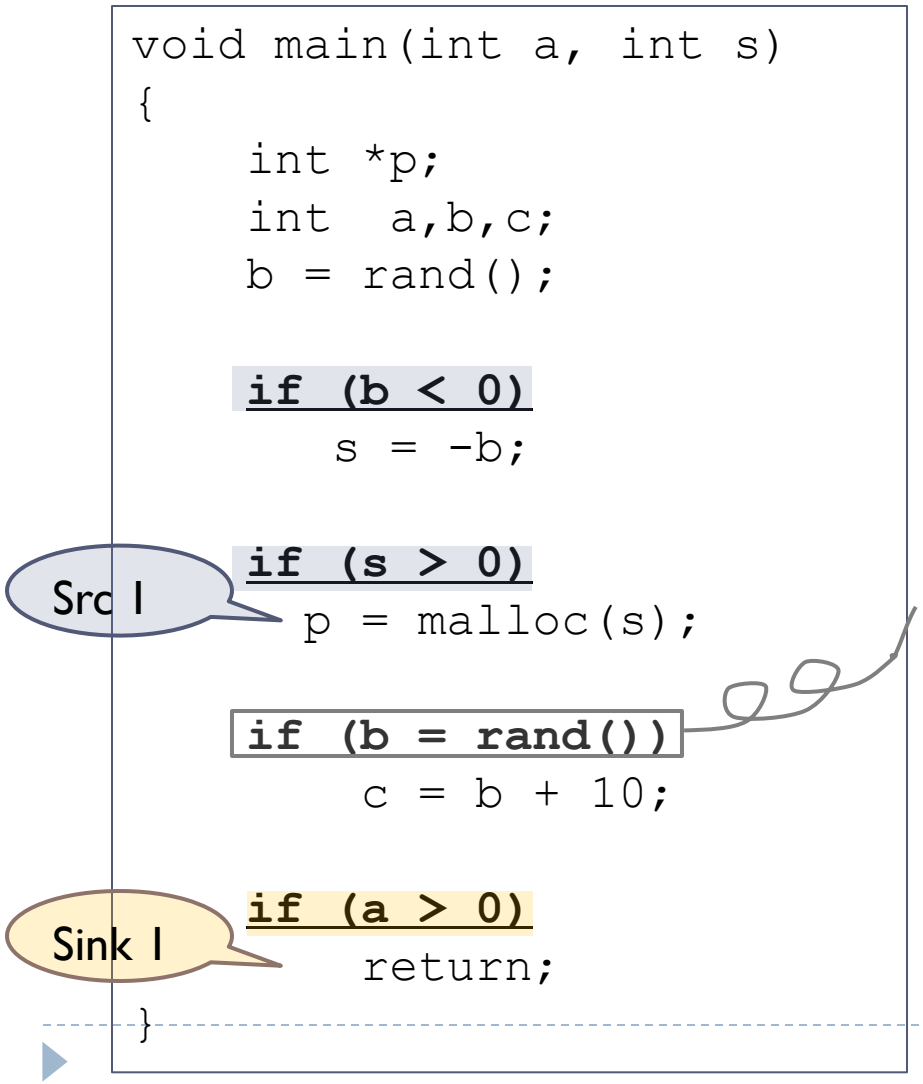
- If Path1 $\sqsubseteq$ Path2,
  - Alarm 1 is false $\Rightarrow$ Alarm 2 is false.
  - Alarm 2 is true $\Rightarrow$ Alarm 1 is true.

However…

- There are few groups produced by above approach.

- To produce many groups, compare only guards which are **critical to alarms**.

Dependence Analysis



- A variable **x** depends on a variable **y**
  - if **y** change ⇒ **x** may also change.
- Using summary-based dependence analysis, extract guards on which source and sink of an alarm depend.

Experimental result

Program	LOC	Total Alarm	Alarm Group	Grouped Alarms	Grouped/ Total(%)	Alarms to inspect(min)
spell	2213	21	6	17	81%	10
wget	5513	19	2	5	26%	16
jwhois	9344	35	4	14	40%	25
bc	14370	32	10	24	75%	17
fftw	184026	107	15	90	84%	31
cvs	116080	117	14	47	40%	83
SGCrypto	116566	33	5	11	33%	27

Example

Alarm A

```
str = str_make(NULL);                          Allocate  str
while (1)                                                          (True)
{
      str = str_make (str);
      add_line_ret = str_add_line (str, stream);    line++;
      if (add_line_ret == ADD_LINE_EOF && !str->len)       (True)
                  return;                          Leak!
}
```

Alarm B

```
str = str_make(NULL);
while (1)                                                          (True)
{
      str = str_make (str);                    Allocate  *(str).str
      add_line_ret = str_add_line (str, stream);    line++;
      if (add_line_ret == ADD_LINE_EOF && !str->len)       (True)
                  return;                          Leak!
}
```

Example

Alarm A (src/lookup.c:178)

```
struct re_pattern_buffer      rpb;
…
if (error = (char*)re_compile_pattern(&rpb)) (True)   //ALLOCATE : rpb->buffer
{
            return NULL;                        //LEAK!
}
```

- There are 3 alarms similar to this.

Example

Alarm B (src/regex.c:1132)

```
static reg_errcode_t
regex_compile (struct_re_pattern buffer* bufp)
{
compile_stack = malloc (32);
if (compile_stack == NULL)                (False)
  return -1;
bufp->buffer = malloc(64);
while (fact1)                             (True)
{
      if (fact2)                          (True)
        return 1;
    …
}
```

- ‘fact1’ and ‘fact2’ can’t be true concurrently.
- Then we know another 4 alarms(including Alarm A) which take this path as sub-path are also false.

Conclusion

- There are alarms which share common causes so that can be grouped together.
- In path-sensitive analyzer, we can group alarms just by comparing paths alarms took.
- To produce many groups, comparing only paths that are critical to alarms is effective.