Scalable Approximation of Quantitative Information Flow in Programs

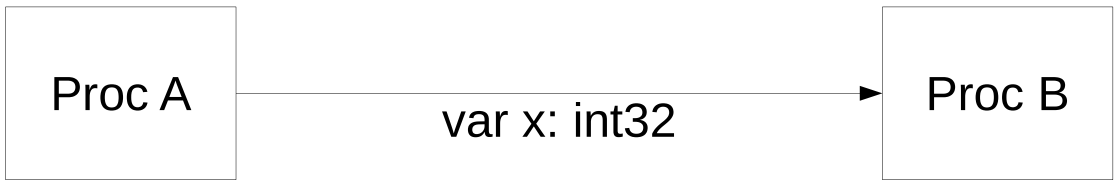
Fabrizio Biondi, Michael A. Enescu, Annelie Heuser, Axel Legay, Kuldeep S.Meel, Jean Quilbeuf

EQUIPE **TAMIS**

CENTRE DE RENNES - BRETAGNE ATLANTIQUE

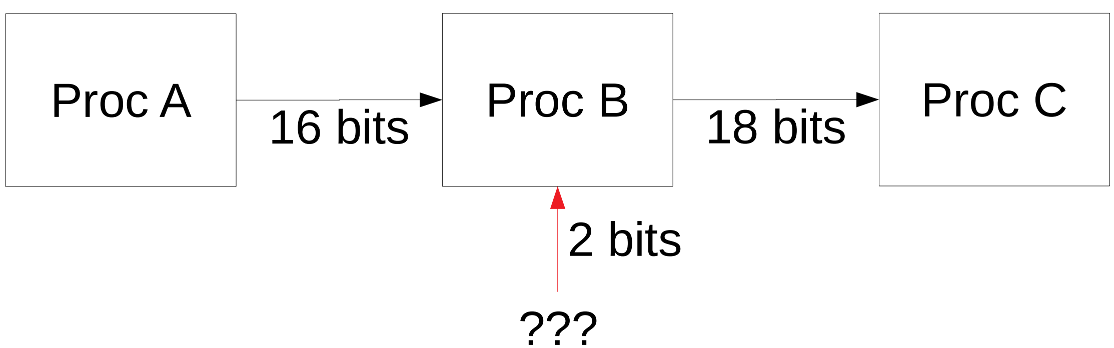
7 Jan 2018

Information Flow Quantification



Information flow from A to B = $\log_2$(possible values(x)) bits

8 values → 3 bits 64 values → 6 bits
4294967296 values → 32 bits



Heartbleed leaks memory...

```
int dtls1_process_heartbeat(SSL *s) {

  unsigned char *p = &s->s3->rrec.data[0], *pl;
  unsigned short hbtype;
  unsigned int payload;
  unsigned int padding = 16;
  //...
  hbtype = *p++;
  n2s(p, payload);

  if (1+2 + payload+16 > s->s3->rrec.length)
    return 0; /* missing in bugged version */

  if (hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp;
    unsigned int write_length =
       1 + 2 + payload + padding;
    //..
    buffer = OPENSSL_malloc(write_length);
    bp = buffer;
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
    //send buffer ...
  }
}
```

This is the fix

Will send kernel memory

...can we detect it?

```
int dtls1_process_heartbeat(SSL *s) {

  unsigned char *p = &s->s3->rrec.data[0], *pl;
  unsigned short hbtype;
  unsigned int payload;
  unsigned int padding = 16;
  //...
  hbtype = *p++;
  n2s(p, payload);

  if (1+2 + payload+16 > s->s3->rrec.length)
    return 0; /* missing in bugged version */

  if (hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp;
    unsigned int write_length =
       1 + 2 + payload + padding;
    //..
    buffer = OPENSSL_malloc(write_length);
    bp = buffer;
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
    //send buffer ...
  }
}
```

Approach*:

1) get SAT constraints on variables you care about

2) use projected SAT model counter to count possible values

* extremely simplified, details in paper

...can we detect it?

```
int dtls1_process_heartbeat(SSL *s) {

  unsigned char *p = &s->s3->rrec.data[0], *pl;
  unsigned short hbtype;
  unsigned int payload;
  unsigned int padding = 16;
  //...
  hbtype = *p++;
  n2s(p, payload);

  if (1+2 + payload+16 > s->s3->rrec.length)
    return 0; /* missing in bugged version */

  if (hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp;
    unsigned int write_length =
       1 + 2 + payload + padding;
    //..
    buffer = OPENSSL_malloc(write_length);
    bp = buffer;
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
    //send buffer ...
  }
}
```

Approach*:

1) get SAT constraints on variables you care about

2) use projected SAT model counter to count possible values

Result:

- Generated SAT formula with 39272 clauses in <1s
- … model counter timeout :(

* extremely simplified, details in paper

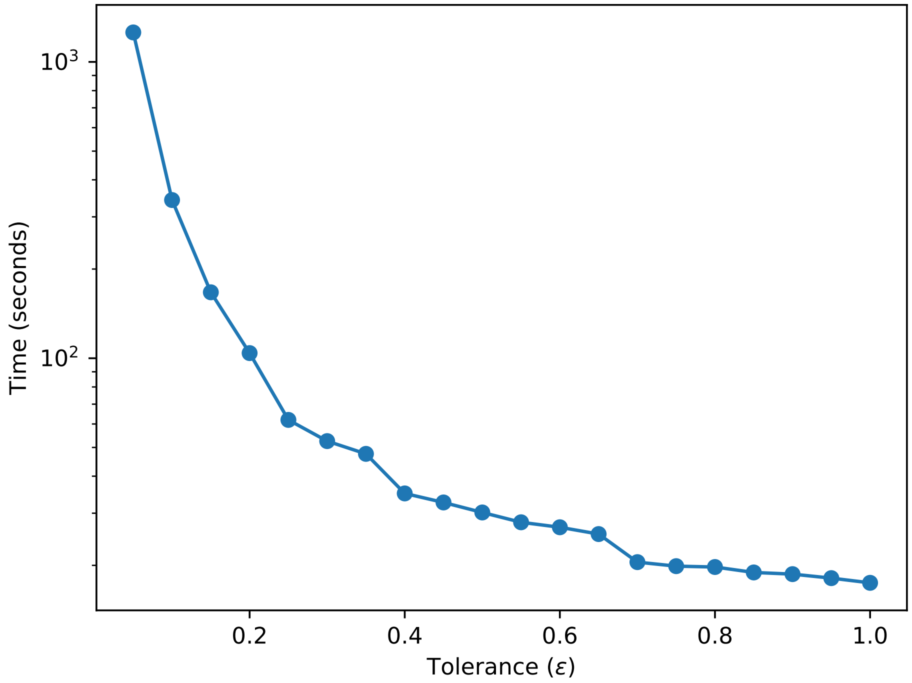
Solution: approximate SAT counting

- Counting precision not very important (we are taking a log anyway)
- Can we trade precision for speed by approximating?
- Klebanov et al.[1] already did it!
- Except…
  - We couldn't reproduce their results (authors were nice and helpful)
  - Error in Theorem 2.12 overestimates termination probability
  - Error in Theorem 2.6 overestimates precision
    (details in paper)
- So we used ApproxMC instead (v2 has projected model counting[2])

[1]V. Klebanov, A. Weigl, and J. Weisbarth. Sound probabilistic #SAT with projection. In QAPL 2016
[2]S. Chakraborty, K. S. Meel, and M. Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In IJCAI 2016

Performance-precision trade-off

Preprocessed AppleTalk benchmark:



Tolerance of 1: approximation is from 0.5 to 2 times the real value
→ exactly +/- 1 bit of information

Benchmarks

Benchmarks from [24, 5, 23, 19]

Experiment name	sharpCDCL leakage	ApproxMC2 leakage	Relative error	sharpCDCL time	ApproxMC2 time	Speedup factor
e-purse	**5.00**	**5.00**	**0%**	**0.06**	**0.28**	**-4.67**
pw-checker	**1.00**	**1.00**	**0%**	**0.00**	**0.00**	—
sum-query	>22.49	32.00	*	t/o	0.87	*
10random	**3.32**	**3.32**	**0%**	**0.00**	**0.00**	—
bsearch16	**16.00**	**16.00**	**0%**	**3.40**	**0.49**	**6.90**
bsearch32	>22.87	32.00	*	t/o	2.13	*
mix-dupl	**16.00**	**16.00**	**0%**	**5.91**	**0.20**	**29.60**
sum32	>22.48	32.00	*	t/o	0.89	*
illustr.	**4.09**	**4.09**	**0%**	**0.00**	**0.01**	—
mask-cpy	**16.00**	**16.00**	**0%**	**6.02**	**0.20**	**30.1**
sanity-1	>22.82	31.04	*	t/o	0.94	*
sanity-2	>22.92	31.00	*	t/o	1.07	*
check-cpy	>22.51	32.00	*	t/o	0.88	*
copy	>22.49	32.00	*	t/o	0.84	*
div-by-2	22.79	31.00	*	t/o	1.06	*
implicit	**>2.81**	**2.81**	**0%**	**0.00**	**0.01**	—
mul-by-2	>22.46	31.00	*	t/o	0.89	*
popcnt	**5.04**	**5.04**	**0%**	**0.00**	**0.01**	—
simp-mask	**8.00**	**8.00**	**0%**	**0.00**	**0.05**	—
switch	**4.25**	**4.25**	**0%**	**0.00**	**0.00**	—
tbl-lookup	>22.45	32.00	*	t/o	0.88	*

Benchmarks

Benchmarks from [24, 5, 23, 19]

Experiment name	sharpCDCL leakage	ApproxMC2 leakage	Relative error	sharpCDCL time	ApproxMC2 time	Speedup factor
ddp	error	128.00	*	error	23.50	*
ddp.pp	error	128.00	*	error	19.55	*
popcount	**5.04**	**5.04**	**0%**	**0.00**	**0.01**	—
sanitize	**4.00**	**4.00**	**0%**	**0.00**	**0.00**	—
openssl.1	**8.00**	**8.00**	**0%**	**1.44**	**70.66**	**-49.10**
openssl.2	**16.00**	**16.00**	**0%**	**4.63**	**75.39**	**-16.30**
openssl.3	>22.24	24.00	*	t/o	92.47	*
openssl.4	>22.91	32.00	*	t/o	86.32	*
openssl.5	>23.10	40.00	*	t/o	87.74	*
openssl.6	error	48.00	*	error	89.60	*
openssl.7	error	56.00	*	error	91.98	*
openssl.8	error	64.00	*	error	98.04	*
openssl.9	error	72.00	*	error	97.41	*
openssl.10	error	80.00	*	error	112.71	*
openssl.15	error	t/o	*	error	t/o	*
openssl.20	error	160.00	*	error	142.48	*
swirl	>12.82	t/o	*	t/o	t/o	—
10random	**3.32**	**3.32**	**0%**	**0.00**	**0.01**	—
bsearch16	**16.00**	**16.00**	**0%**	**4.16**	**0.68**	**6.12**
bsearch16.pp	**16.00**	**16.00**	**0%**	**3.73**	**0.35**	**10.70**
bsearch32	>22.79	32.00	*	t/o	3.21	*
bsearch32.pp	>22.90	32.00	*	t/o	6.93	*
fx	**16.00**	**16.00**	**0%**	**5753.42**	**7307.61**	**-1.27**
mixdup	**16.00**	**16.00**	**0%**	**8.44**	**0.22**	**38.40**
sum.32	>22.78	32.00	*	t/o	0.98	*

So how about Heartbleed?

```
int dtls1_process_heartbeat(SSL *s) {

  unsigned char *p = &s->s3->rrec.data[0], *pl;
  unsigned short hbtype;
  unsigned int payload;
  unsigned int padding = 16;
  //...
  hbtype = *p++;
  n2s(p, payload);

  if (1+2 + payload+16 > s->s3->rrec.length)
    return 0; /* missing in bugged version */

  if (hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp;
    unsigned int write_length =
       1 + 2 + payload + padding;
    //..
    buffer = OPENSSL_malloc(write_length);
    bp = buffer;
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
    //send buffer ...
  }
}
```

Approach*:

1) get SAT constraints on variables you care about

2) use *approximate* projected SAT model counter to *estimate* possible values

Result:

- Generated SAT formula with 39272 clauses in <1s
- Computed flow of ~15 bytes in 25s
- Reducing confidence gives ~15.1 bytes in 2s
- Normal flow should be 1 byte*
- **Bug found!**

* extremely simplified, details in paper

Conclusions

- Information flow quantification can detect interesting bugs

- Approximate quantification is significantly faster than precise
  - Large performance increase for negligible precision loss

- Approximate quantification scales to real-world code and bugs
  - Modeling still complex (but mostly engineering problem)

- Future work:
  - Lower-bound estimation is sufficient and faster
  - Multiple upwards/downwards passes for refinement in large programs
  - ...mostly engineering?

Thank you for your attention!