ASE 2018

CANAL: A Cache Timing Analysis Framework via LLVM Transformation

Chungha Sung | Brandon Paulsen | Chao Wang

Software verification & analysis

```
while bytecount<0x20000:
    client.CChaltcpu();

    randcode=client.CCpeekiramword(0x08);
    randcount=client.CCpeekiramword(0x0A);

    if randcount!=lastcount and randcode==0xbeef:
        #New bytes are ready, halted in steady state.
        lastcount=randcount;
        for a in range(bytestart,bytestart+bytescount):
            file.write(chr(client.CCpeekirambyte(a)));
            bytecount=bytecount+1;
        print "Got 0x%06x bytes." % bytescount;
        print "%04x %04x: %02x%02x..." % (
            client.CCpeekiramword(0x08),
            client.CCpeekiramword(0x0a),
            client.CCpeekirambyte(0x0C),
            client.CCpeekirambyte(0x0d));

    client.CCreleasecpu();
```


- Model Checking
- Abstract Interpretation
- Symbolic Execution

Checking Functional properties

Ex)
assert(x > 1);

Software verification & analysis

```
while bytecount<0x20000:
    client.CChaltcpu();

    randcode=client.CCpeekiramword(0x08);
    randcount=client.CCpeekiramword(0x0A);

    if randcount!=lastcount and randcode==0xbeef:
        #New bytes are ready, halted in steady state.
        lastcount=randcount;
        for a in range(bytestart,bytestart+bytescount):
            file.write(chr(client.CCpeekirambyte(a)));
            bytecount=bytecount+1;
        print "Got 0x%06x bytes." % bytescount;
        print "%04x %04x: %02x%02x..." % (
            client.CCpeekiramword(0x08),
            client.CCpeekiramword(0x0a),
            client.CCpeekirambyte(0x0C),
            client.CCpeekirambyte(0x0d));

    client.CCreleasecpu();
```

Model Checking

Abstract Interpretation

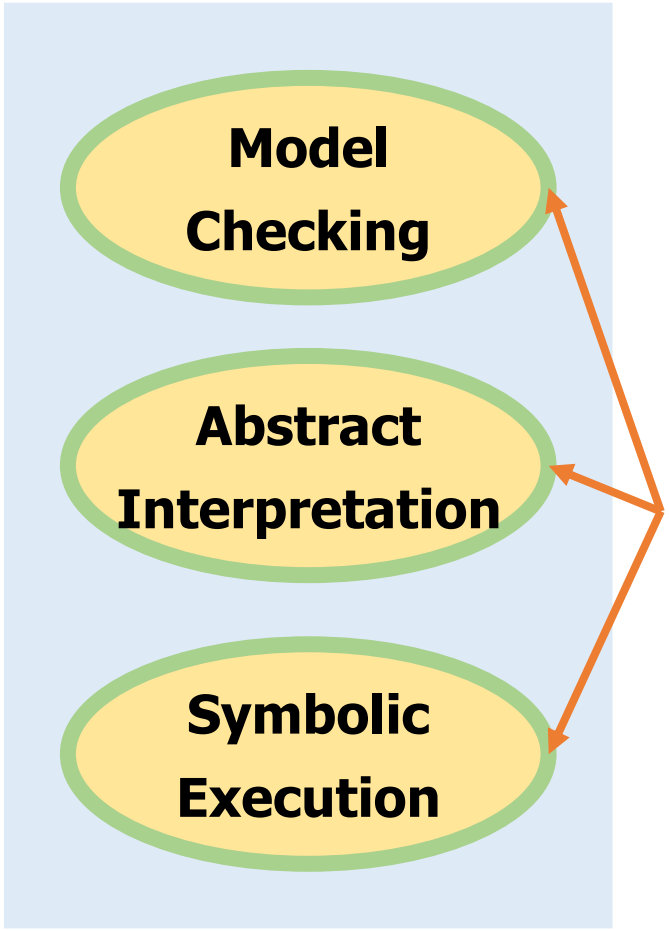
Symbolic Execution

Non-functional Properties
e.g. Cache behavior

Ex)
The number of cache misses?

Software verification & analysis

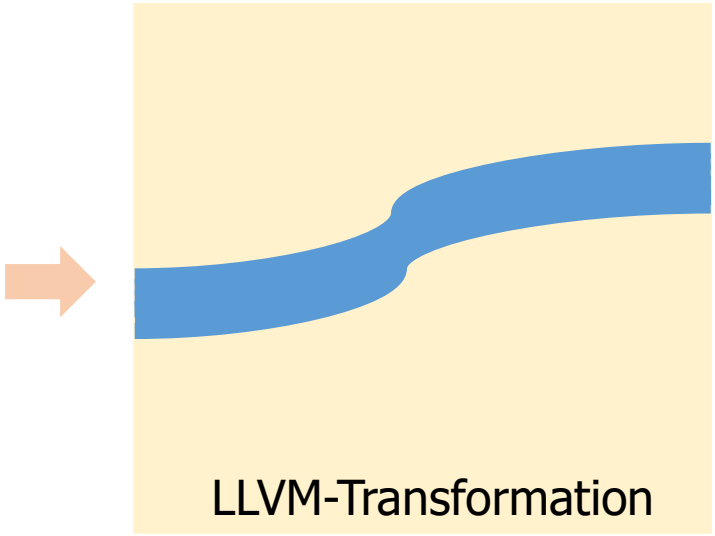
```
while bytecount<0x20000:
    client.CChaltcpu();

    randcode=client.CCpeekiramword(0x08);
    randcount=client.CCpeekiramword(0x0A);

    if randcount!=lastcount and randcode==0xbeef:
        #New bytes are ready, halted in steady state.
        lastcount=randcount;
        for a in range(bytestart,bytestart+bytescount):
            file.write(chr(client.CCpeekirambyte(a)));
            bytecount=bytecount+1;
        print "Got 0x%06x bytes." % bytescount;
        print "%04x %04x: %02x%02x..." % (
            client.CCpeekiramword(0x08),
            client.CCpeekiramword(0x0a),
            client.CCpeekirambyte(0x0C),
            client.CCpeekirambyte(0x0d));

    client.CCreleasecpu();
```



You'd have to change each of these tools to model cache behavior

CANAL

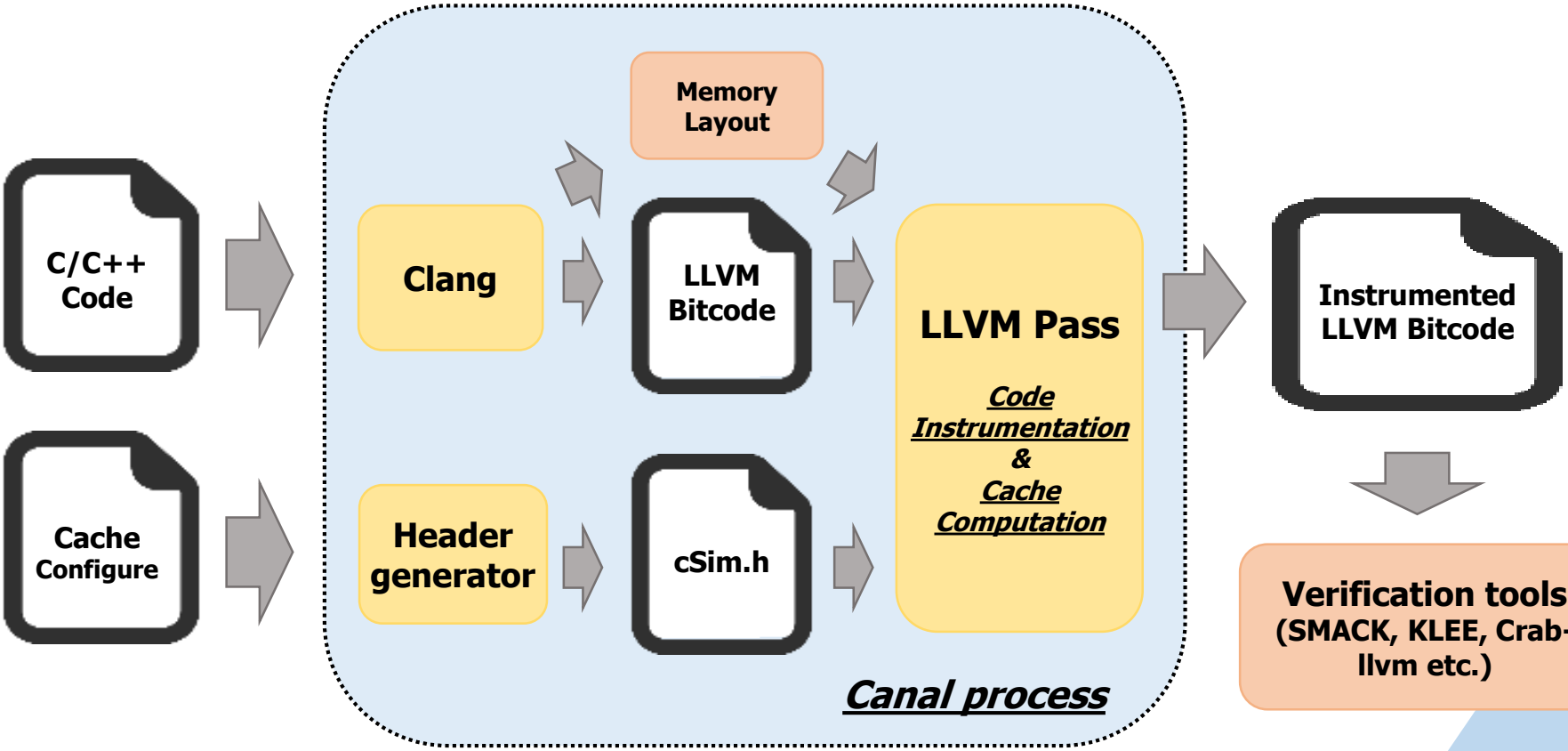
```
while bytecount<0x20000:
    client.CChaltcpu();

    randcode=client.CCpeekiramword(0x08);
    randcount=client.CCpeekiramword(0x0A);

    if randcount!=lastcount and randcode==0xbeef:
        #New bytes are ready, halted in steady state.
        lastcount=randcount;
        for a in range(bytestart,bytestart+bytescount):
            file.write(chr(client.CCpeekirambyte(a)));
            bytecount=bytecount+1;
        print "Got 0x%06x bytes." % bytescount;
        print "%04x %04x: %02x%02x..." % (
            client.CCpeekiramword(0x08),
            client.CCpeekiramword(0x0a),
            client.CCpeekirambyte(0x0C),
            client.CCpeekirambyte(0x0d));

    client.CCreleasecpu();
```



- Model Checking
- Abstract Interpretation
- Symbolic Execution

1. **Now, cache (and other non-functional) properties can be handled by existing verifiers**
2. **General (not tool-specific) cache modeling framework**

Overview


C/C++ Code
Cache Configure
Clang
Memory Layout
LLVM Bitcode
Header generator
cSim.h
LLVM Pass
Code Instrumentation & Cache Computation
Canal process
Instrumented LLVM Bitcode
Verification tools (SMACK, KLEE, Crab-llvm etc.)

Code instrumentation

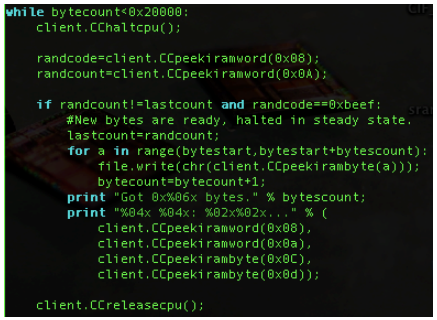
```
T = Y;
(Inserted function calls below)
__CSIM_Load(address set of “Y”, address tags of “Y”);
__CSIM_Store(address set of “T”, address tags of “T”);
```

Code instrumentation is done at the LLVM-Bitcode level

Outline

- **Motivation**
- **Code Instrumentation**
- ***Usages***
  - ***Use CANAL as a simulator (omitted)***
  - ***Use CANAL with Symbolic execution tool***
  - ***Use CANAL with Static analysis tool***
  - ***Use CANAL with Software verification tool***
- **Conclusion**

Usage 1 – Symbolic execution tool



```
while bytecount<0x20000:
    client.CChaltcpu();

    randcode=client.CCpeekiramword(0x08);
    randcount=client.CCpeekiramword(0x0A);

    if randcount!=lastcount and randcode==0xbeef:
        #New bytes are ready, halted in steady state.
        lastcount=randcount;
        for a in range(bytestart,bytestart+bytescount):
            file.write(chr(client.CCpeekirambyte(a)));
            bytecount=bytecount+1;
        print "Got 0x%06x bytes." % bytescount;
        print "%04x %04x: %02x%02x..." % (
            client.CCpeekiramword(0x08),
            client.CCpeekiramword(0x0a),
            client.CCpeekirambyte(0x0C),
            client.CCpeekirambyte(0x0d));

    client.CCreleasecpu();
```

CANAL

Instrumented LLVM Bitcode

Symbolic execution tools (e.g Klee)

Check if there exist two inputs that lead to different cache stats
(Side-channel leakage)

Usage 1 – Symbolic execution tool (Cont'd)

```
klee_make_symbolic(&input1);
klee_make_symbolic(&input2);

__CSIM_init_cache();

call_program1(input1);
h1  = __CSIM_num_hit;
m1 = __CSIM_num_miss;

__CSIM_init_cache();

call_program1(input2);
h2  = __CSIM_num_hit;
m2 = __CSIM_num_miss;

assert(h1 == h2 && m1 == m2);
```

Usage 1 – Symbolic execution tool (Cont'd)

```
klee_make_symbolic(&input1);
klee_make_symbolic(&input2);
```

Define symbolic inputs

```
__CSIM_init_cache();

call_program1(input1);
h1  = __CSIM_num_hit;
m1 = __CSIM_num_miss;

__CSIM_init_cache();

call_program1(input2);
h2  = __CSIM_num_hit;
m2 = __CSIM_num_miss;

assert(h1 == h2 && m1 == m2);
```

Usage 1 – Symbolic execution tool (Cont'd)

```
klee_make_symbolic(&input1);
klee_make_symbolic(&input2);

__CSIM_init_cache();                    Cache status initialization

call_program1(input1);
h1  = __CSIM_num_hit;                   Run program and get cache stats
m1 = __CSIM_num_miss;

__CSIM_init_cache();

call_program1(input2);
h2 = __CSIM_num_hit;
m2 = __CSIM_num_miss;

assert(h1 == h2 && m1 == m2);
```

Input 1

Usage 1 – Symbolic execution tool (Cont'd)

```
klee_make_symbolic(&input1);
klee_make_symbolic(&input2);

__CSIM_init_cache();

call_program1(input1);
h1 = __CSIM_num_hit;
m1 = __CSIM_num_miss;
```

Input 2

```
__CSIM_init_cache();

call_program1(input2);
h2  = __CSIM_num_hit;
m2 = __CSIM_num_miss;
```

Cache status initialization

Run program and get cache stats

```
assert(h1 == h2 && m1 == m2);
```

Usage 1 – Symbolic execution tool (Cont'd)

```
klee_make_symbolic(&input1);
klee_make_symbolic(&input2);

__CSIM_init_cache();

call_program1(input1);
h1  = __CSIM_num_hit;
m1 = __CSIM_num_miss;

__CSIM_init_cache();

call_program1(input2);
h2  = __CSIM_num_hit;
m2 = __CSIM_num_miss;

assert(h1 == h2 && m1 == m2);
```

Check stats are the same

Usage 2 – Software verification tool

```
while bytecount<0x20000:
    client.CChaltcpu();

    randcode=client.CCpeekiramword(0x08);
    randcount=client.CCpeekiramword(0x0A);

    if randcount!=lastcount and randcode==0xbeef:
        #New bytes are ready, halted in steady state.
        lastcount=randcount;
        for a in range(bytestart,bytestart+bytescount):
            file.write(chr(client.CCpeekirambyte(a)));
            bytecount=bytecount+1;
        print "Got 0x%06x bytes." % bytescount;
        print "%04x %04x: %02x%02x..." % (
            client.CCpeekiramword(0x08),
            client.CCpeekiramword(0x0a),
            client.CCpeekirambyte(0x0C),
            client.CCpeekirambyte(0x0d));

    client.CCreleasecpu();
```

CANAL

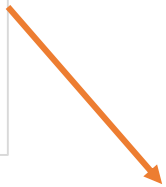
Instrumented LLVM Bitcode

Software verification tool (e.g SMACK)

Check if a memory read or write always leads to cach hit/miss **(MUST hit/miss analysis)**

Usage 2 – Software verification tool (Con'd)

```
if (cond)
    buffer[0] = 1;
else
    buffer[16] = 1;

x = buffer[2];

h = __CSIM_Load_ret;

assert (h == true);
```



Check: Read of buffer[2]
always leads to cache hit?

Usage 2 – Software verification tool (Con'd)

```
if (cond)
    buffer[0] = 1;
else
    buffer[16] = 1;
```

buffer[0] and buffer[16] are in different cache line

```
x = buffer[2];

h = __CSIM_Load_ret;

assert (h == true);
```

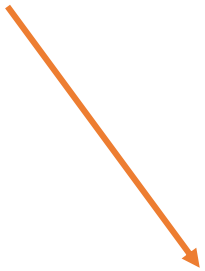
Usage 2 – Software verification tool (Con'd)

```
if (cond)
    buffer[0] = 1;
else
    buffer[16] = 1;

x = buffer[2];

h = __CSIM_Load_ret;

assert (h == true);
```

buffer[2] will be the first cache line access when the branch was not taken.

Usage 2 – Software verification tool (Con'd)

```
if (cond)
    buffer[0] = 1;
else
    buffer[16] = 1;

x = buffer[2];

h = __CSIM_Load_ret;

assert (h == true);
```



Read the cache status of the last Load/Store operation

Usage 3 – Static analysis tool

```
while bytecount<0x20000:
    client.CChaltcpu();

    randcode=client.CCpeekiramword(0x08);
    randcount=client.CCpeekiramword(0x0A);

    if randcount!=lastcount and randcode==0xbeef:
        #New bytes are ready, halted in steady state.
        lastcount=randcount;
        for a in range(bytestart,bytestart+bytescount):
            file.write(chr(client.CCpeekirambyte(a)));
            bytecount=bytecount+1;
        print "Got 0x%06x bytes." % bytescount;
        print "%04x %04x: %02x%02x..." % (
            client.CCpeekiramword(0x08),
            client.CCpeekiramword(0x0a),
            client.CCpeekirambyte(0x0C),
            client.CCpeekirambyte(0x0d));

    client.CCreleasecpu();
```

CANAL

Instrumented LLVM Bitcode

Static analysis tool (e.g Crab-llvm)

Compute invariants over cache stats

(e.g., min/max of cache hits/misses)

Usage 3 – Static analysis tool (Con'd)

```
if (cond)
    buffer[0] = 1;
else
    buffer[16] = 1;

buffer[2] = 1;

s_h = __CSIM_num_Store_hit;
s_m = __CSIM_num_Store_miss;

assert (s_h > 1);
assert (s_m < 3);
assert (s_h + s_m == 2);
```

Usage 3 – Static analysis tool (Con'd)

```
if (cond)
    buffer[0] = 1;
else
    buffer[16] = 1;

buffer[2] = 1;
```

```
s_h  = __CSIM_num_Store_hit;
s_m = __CSIM_num_Store_miss;

assert (s_h > 1);
assert (s_m < 3);
assert (s_h + s_m == 2);
```

Check invariants over the number of cache hits and misses.

Conclusions

- Proposed a **unified** framework for modeling cache behaviors through LLVM-transformation
- *CANAL* can be used as a simulator without losing accuracy
- *CANAL* can be used tougher with various software verification tools

Thank you!

https://github.com/canalcache/canal