Memory simulations, security and optimization in a verified compiler

David Monniaux

VERIMAG

December 9, 2024

Contents

CompCert

Formally verified C compiler, effort led by Xavier Leroy & Sandrine Blazy

"If compilation succeeds, then the assembly program matches the C program."

Formally verified: compiler written in Coq
+ correctness theorem proved in Coq, a proof assistant
(mathematical proof, machine-checked)

Rationale for CompCert

Certain industries (avionics, nuclear...) must demonstrate that the object code is equivalent to the source.

Conventional approach

Disable optimizations
"Human" comparisons
"This compiler worked in other safety-critical projects"

CompCert

Use the mathematical proof

Versions under discussion

“Official” releases

`https://github.com/AbsInt/CompCert`

“Chamois” branch

for our own agile development 

`https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert`

Correctness theorem

execution = trace of “externally visible **events**” (calls to external functions, volatile variables accesses)

The trace at assembly matches the C trace.

Obtained by “forward simulation” (assembly simulates C) through “match” relations

Forward simulations

Lockstep "One step of the program before transformation maps to one step after transformation."
$\sigma_1 \rightarrow_e \sigma_2$ and $m(\sigma_1, \sigma'_1)$ then there exists σ'_2 such that $\sigma'_1 \rightarrow_e \sigma'_2$ and $m(\sigma_3, \sigma'_2)$
e = "observable events"
e.g. "replace $x \times y$ by a move from a register already containing that expression"

Plus "One step maps to several steps."
e.g. function call from one instruction to many (move operands to registers / stack etc.)

Star "Several steps map to several steps."

Matching and definedness

m match relation between **states**:

- program counter
- abstract stack (list of blocks and return addresses on the call stack)
- value in set of "pseudo registers"
- values in addressable memory

"Definedness": special "undefined" value that can be refined during program transformtions

Most matches: "s' most defined than s"

Memory model

Memory divided in **blocks** ($\simeq$ memory objects in C standard)

Memory address = pair (b, o) block identifier + offset

Operations:

- allocate a block
- free a block
- read
- write
- decrease permissions

Memory extension

m' **extends** m = same block numbers, blocks in m' extend the index ranges of blocks in m, content is more defined

e.g. “add extra workspace to the end of the stackframe (spills)”

Memory injection

m **injects** into m': each block b in m maps to a sub-range of a block b' in m'

e.g. proof of function inlining: “Stackframes of inlined functions are portions of the stackframes of the target program”

External call axiomatization

External calls can do “anything” to memory
...but must respect memory injections and extensions!

Intuition for extensions: “If the external call succeeded with smaller memory blocks it must still succeed with bigger blocks.”

Intuition for injections: “The behavior of external calls does not depend on the actual memory addresses as long as those within the same blocks respect the layout.”

Bouquetin

Countermeasures against attacks

Contents

Buffer overflow attack

Stackframe

Local array	Return address

Attack

Feed incorrect data into program.

Trigger a buffer overflow bug, overflow the array, choose return address.

Possibly return into the array itself.

(Other countermeasures)

Countermeasures

On processors with a MMU + an operating system

1. address space layout randomization (ASLR): the attacker cannot guess the value to put in the return address
2. make the stack non-executable: the attacker cannot execute arbitrary code overflowing the stack
3. make normal data non-executable: the attacker cannot execute arbitrary code in the memory heap

(Other countermeasures)

Counterattacks

- can guess even with ASLR
- 2,3: return-oriented programming (use code from application and libc)

A simple countermeasure: stack canaries

arrays	(padding)		return address

Overflowing a local buffer **clobbers the canary**.
If the canary contains an incorrect value, kill the program.

Countermeasure

- extend allocation size
- function prologue (at function start): install the canary
- function epilogue (before return or tail call): check that the canary is still there, branch to trap if not

Proof

Proof of **correctness** by memory extension: does not perturb legal executions.

Currently no proof of **adequation**: “the countermeasure blocks certain attacks”, would need an **attacker model** (nonstandard execution semantics? expressed by code transformation?)

Contents

Tail call elimination

(Already in CompCert, by Leroy)

```
unsigned g(unsigned);

unsigned f(unsigned x) {
  if (x==1) return x;
  else return g(x);
}

unsigned g(unsigned x) {
  if (x % 2) return f(3*x+1);
  else return f(x/2);
}
```

The tail calls are just “jumps” to the head of the other function. During a tail call, the stack frame of the current function is destroyed.

Tail call elimination

Tail call eliminated on recursive function: tail call sequence is

- deallocate stack frame (and restore callee-saved registers)
- jump to head of function
- allocate stack frame (and save callee-saved registers)

Why restore just to save again in the same place?

Tail recursion elimination

Just put the arguments in the correct pseudo registers and branch to top of function.
Do not save and restore.

Turn tail call into a normal loop

Simulation proof

Original program

- "f executes with stackframe s_1"
- deallocate, jump, allocate
- "f executes with stackframe s_2"

Transformed program

- "f executes with stackframe s'_1"
- jump
- "f executes with stackframe s'_1"

s'_1 successively **simulates** s_1 then s_2

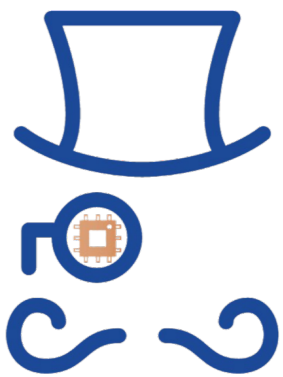
Contents

Conclusion

Illustrates how security and optimization features dealing with memory can be proved through memory injections and extensions.

Caveat: current implementation of pointer authentication (not discussed due to limited time) does not commute with injections, not an issue since very late in compilation chain and no more injections

`https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert`



"PEPR" Cybersecurity "Arsene" project `https://www.pepr-cyber-arsene.fr/`

ASK ME ABOUT OUR OPEN PROFESSORSHIP POSITIONS!