The formal verification of compilers and what it doesn’t say about security

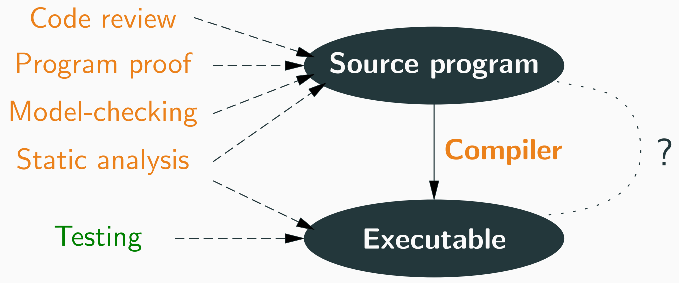
Xavier Leroy

Dagstuhl seminar on secure compilation, May 2018

Inria, Paris

Overview of compiler verification

Compiler verification



Prove that the compiled code behaves as prescribed by the semantics of the source program

- once and for all: compiler verification, or
- at every compilation run: translation validation.

“Behaves as prescribed” ???

Q1: what are the observable behaviors?

$\rightarrow$ impacts the semantics of the source and target languages.

Q2: which relation (“preservation”) between the behaviors of the source and compiled codes?

$\rightarrow$ impacts the statement and proof of compiler correctness.

Observable behaviors

For an imperative language, observables typically include

- Divergence / normal termination / termination on an error.
- Traces of input/output actions.

In the CompCert verified C compiler project, I/O actions are:

- calls to library functions, e.g. `printf`, `getchar`;
- loads and stores on global `volatile` variables, modeling memory-mapped hardware devices.

Other computation steps are not observable and can be optimized away during compilation: pure arithmetic, loads and stores on regular memory, etc.

Labeled transition systems

$$\text{state}_1 \xrightarrow{\ell} \text{state}_2 \qquad\qquad \ell ::= \tau \mid c?v \mid c!v$$

Very natural for machine languages and assembly languages, with state $=$ (registers, memory)

Also works for higher-level languages (C, Clight, Cminor), with

$$\begin{aligned}\text{state} \;=\; (&\text{statement under focus,}\\ &\text{environment for variables,}\\ &\text{context (a.k.a. continuation) (incl. call stack),}\\ &\text{memory state})\end{aligned}$$

Well-known proof techniques for LTS: bisimulations, etc.

From transitions to behaviors

Normal termination with trace $a_1 \ldots a_k$:

$$\text{initial} \ni s \xrightarrow{\tau} s_1 \xrightarrow{a_1} s_2 \xrightarrow{\tau} \cdots \xrightarrow{a_k} s_n \in \text{final}$$

Abnormal termination with trace $a_1 \ldots a_k$:

$$\text{initial} \ni s \xrightarrow{\tau} s_1 \xrightarrow{a_1} s_2 \xrightarrow{\tau} \cdots \xrightarrow{a_k} s_n \in \text{error}$$

Reactive divergence with infinite trace $a_1 \ldots a_k \ldots$:

$$\text{initial} \ni s \xrightarrow{\tau} \cdots \xrightarrow{a_i}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} \cdots \xrightarrow{a_j}\xrightarrow{\tau}\xrightarrow{\tau} \cdots$$

Silent divergence with trace $a_1 \ldots a_k$:

$$\text{initial} \ni s \xrightarrow{\tau} \cdots \xrightarrow{a_k} s_n \xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} \cdots$$

Notions of semantic preservation

Notions of semantic preservation: equivalence

Theorem
If the compiler produces code C from source S,
without reporting a compile-time error,
then C and S have the same observable behaviors.

Proof technique in LTS: bisimulations.

Appropriate for high-level languages with fully-defined, deterministic semantics (e.g. CakeML). But not for C. . .

Internal nondeterminism

C/C++ and some other languages have internal nondeterminism such as evaluation orders that are not fully specified. The compiler is allowed to choose one of the possible evaluation orders.

```
int a(void) { printf("a"); return 1; }
int b(void) { printf("b"); return 2; }
int c(void) { printf("c"); return 3; }
int main(void) { return a() + b() + c(); }
```

The subexpressions `a()` and `b()` and `c()` can be evaluated in any order that the compiler chooses.

$\Rightarrow$ any of the 6 permutations `abc`, `acb`, `bac`, `bca`, `cab`, `cba` is a valid output for this program.

Notions of semantic preservation: refinement

Theorem
If the compiler produces code C from source S,
without reporting a compile-time error,
then every observable behavior of C is a possible behavior of S.

Proof technique in LTS: “backward” simulations.

Backward simulation diagrams

Original program　　Transformed program

Simulation:

$$\begin{tikzcd}[column sep=huge, row sep=huge]
S_1 \arrow[r, dash, "\mathcal{R}"] \arrow[d, "+ \text{ or } \|S_2'\| < \|S_2\| \text{ and } *"', "t"] & S_2 \arrow[d, "t"] \\
S_1' \arrow[r, dash, "\mathcal{R}"] & S_2'
\end{tikzcd}$$

Progress:

$$\begin{tikzcd}[column sep=huge, row sep=large]
\texttt{final} \not\ni S_1 \arrow[r, dash, "\mathcal{R}"] & S_2 \notin \texttt{final} \arrow[d, "t"] \\
 & S_2'
\end{tikzcd}$$

Undefined behaviors

C-style undefined behaviors

Consider run-time errors such as integer division by zero, or accessing an array out of bounds.

Most programming languages define exactly the semantics of run-time errors: abort the program, raise an exception, etc.

In contrast, C and C++ treat run-time errors and many other dark corners of the language as undefined behaviors: anything can happen, from aborting the program to computing the wrong result to mounting a security attack.

Undefined behaviors and optimization

Moderate interpretation of the C standards:

Since “undefined behavior” $\Rightarrow$ “now anything can happen”, compilers can remove computations that cause undefined behaviors, e.g.

```
z = x / y;  z = 1;      →     z = 1;
```

This is valid whether $\mathtt{y} = 0$ or not.
(In Java such an optimization would be invalid if $\mathtt{y} = 0$.)

Radical interpretation of the C standards:

Source programs are assumed free of undefined behaviors.

Compilers can generate any code (e.g. “terminate immediately” or “launch missile now”) if the source program contains any u.b.

Undefined behaviors and optimization

The radical interpretation allows the compiler to reorder undefined behaviors with observable computations, causing the undefined behavior to occur observably earlier.

For example, some versions of GCC reorder as follows:

```
printf("dividing");     →     z = x / y;
z = x / y;                    printf("dividing");
```

This is undesirable in practice and cannot be explained just by "undefined behavior" $\Rightarrow$ "now anything can happen".

Semantics of undefined behaviors

Assume undefined behaviors transition to a special state Ω.

"Anything can happen" semantics: $\Omega \xrightarrow{\ell} \Omega$ for all ℓ.

Compiler correctness = refinement of non-determinism.

$$\forall b \in \mathcal{B}(\text{compiled}), \quad b \in \mathcal{B}(\text{source})$$

Semantics of undefined behaviors

Assume undefined behaviors transition to a special state Ω.

"Anything can happen" semantics: $\Omega \xrightarrow{\ell} \Omega$ for all ℓ.

"Getting stuck" semantics: $\Omega \in$ errors, and typically $\Omega \not\to$

Compiler correctness = refinement of non-determinism and improvement of "getting stuck" behaviors.

$$\forall b \in \mathcal{B}(\text{compiled}), \quad \exists b' \in \mathcal{B}(\text{source}), \quad b' \preceq b$$

where $b' \preceq b$ ("b improves on b'") is

- either $b = b'$,
- or b' is "getting stuck after performing an I/O trace t", and b is any behavior whose trace starts with t.

Backward simulation diagrams with refinement of u.b.

Original program Transformed program

Simulation:

$$\begin{tikzcd}
\texttt{safe} \ni S_1 \arrow[r, dash, "\mathcal{R}"] \arrow[d, "t", "{\substack{+ \\ \text{or} \\ \|S_2'\| < \|S_2\| \text{ and } *}}"'] & S_2 \arrow[d, "t"] \\
S_1' \arrow[r, dash, "\mathcal{R}"] & S_2'
\end{tikzcd}$$

Progress:

$$\begin{tikzcd}
\texttt{safe} \ni S_1 \arrow[r, dash, "\mathcal{R}"] & S_2 \notin \texttt{final} \arrow[d, "t"] \\
& S_2'
\end{tikzcd}$$

$S \in \texttt{safe}$ means $\neg\, S \xrightarrow[\tau]{*} \Omega$

(cannot cause undefined behavior silently).

Connections (or lack thereof) with compiler security

What CompCert-style proofs say

What is proved: preservation of

- safety properties (nothing wrong happens)
- liveness properties (something good eventually happens)

in a normal, non-adversarial execution context.

What CompCert-style proofs do not say

Nothing is proved about

- The behavior of the compiled code after the source program runs into undefined behavior.
- More generally: unconditional safety of the compiled code.
- Linking with code not compiled by CompCert.
- More generally: uses in adversarial contexts.
- Preservation of non-functional properties (time, etc).
- More generally: side channels (leaks) in the compiled code.

Is CompCert a bad C compiler, security-wise?

No! It’s certainly no worse than GCC or Clang in this respect…

CompCert refrains from stupid optimizations of the kind that breaks security countermeasures, and tries to present programmers with a predictable model of optimizations.

However this is all best efforts, verified only by testing: CompCert’s proofs gives no formal guarantees about this.

Could they? See examples next.

Constant-time code

Constant-time cryptographic code

To avoid obvious timing leaks:

- Secret data is manipulated only by operations whose execution time is independent on the value of their arguments (e.g. integer addition, and, or, xor).
- Other operations, e.g. conditional branches, array indexing, integer division, are never applied to secret data.

Compiling constant-time code

If the source code is “constant time”, is the compiled code too?

What prevents the compiler from introducing a case analysis on secret data? E.g. for optimization purposes:

```
if (secret > 0) {
  /* faster code */
} else {
  /* original code */
}
```

Assuming this case analysis is functionally correct, a CompCert-style semantic preservation proof still holds, yet “constant-time-ness” is broken.

CompCert's constant-time story

CompCert's optimizations never introduce a conditional branch or a memory indexing (that wasn't already in the source).

CompCert's instruction selection phase sometimes must introduce a conditional branch to work around a limitation of the target ISA.

Example

PowerPC 32 bits has a `fctiwz` instruction to convert from FP to 32-bit signed integer. For conversion to 32-bit unsigned integer, CompCert uses the formula

$$\texttt{if } (x < 2^{31}) \texttt{ then fctiwz}(x) \texttt{ else fctiwz}(x - 2^{31}) + 2^{31}$$

Proving something about CompCert and constant-time

A posteriori verification of conditional branches:

(Almeida, Barbosa, Barthe, Dupressoir; CCS 2013)

- Trace source-level conditionals down to the generated assembly code, using CompCert's annotation mechanism.
- Statically analyze the generated assembly code to check that all conditional branches trace back to source conditionals.

A priori proof of constant-time preservation?

(pure speculation)

- Avoid generating tests in instruction selection (focus on an ISA that don't need them, e.g. RISC-V 64).
- Observe non-constant-time ops in the semantics to prove their preservation.

Observing more to preserve more

Change the semantics so that non-constant-time operations (conditionals, etc) are observable: they produce events in the trace.

If we can still prove that compilation preserves traces, we known that compilation preserves non-constant-time operations.

Consequently, no NCT operation over secret arguments was generated.

This prevents a number of safe optimizations:

- Introducing a case analysis over non-secret data.
- Deleting useless conditional branches or memory accesses.

The latter could be accommodated by proving that compilation improves traces (including removal of some traced operations) instead of just preserving traces.

Making optimizations robust w.r.t. undefined behaviors

When optimization opens a security hole: CVE-2009-1879

Linux kernel 2.6.30:

```
struct sock *sk = tun->sk;
if (tun == NULL)
    return POLLERR;
/* write to address based on tun */
```

GCC removed the `tun == NULL` safety check, reasoning that if `tun` is `NULL` the memory access `tun->sk` is undefined behavior and the generated code can do anything.

However, this code ran in the Linux kernel, and the read `tun->sk` can succeed (without a kernel panic) even if `tun` is `NULL`.

Removing the `tun == NULL` check therefore opened an exploitable security hole, CVE-2009-1897.

Generic optimizations at play

GCC was not applying a specific “remove NULL checks” optimization designed to annoy security people.

Rather, the poor optimization is a natural consequence of two standard compilation passes:

- Value analysis: approximates the values that a variable or expression can or cannot take.
- Constant propagation: replace expressions having known values by those values.

```
struct sock *sk = tun->sk;
                    // value analysis: now tun != NULL
if (tun == NULL)
                    // value analysis: tun == NULL is 0
                    // constant propagation: rewrite to if(0)
                    // constant propagation: remove if(0)
```

The fine line between desirable and dangerous optimizations

Compare the following two optimizations:

```
if (p != NULL) {                   if (p != NULL) {
  ...                     -->        ...
  if (p == NULL) ...                 /*nothing*/
}

x = *p;                            x = *p;
...                       -->      ...
if (p == NULL) ...                 /* nothing */
```

Both optimizations rely on the same value analysis + constant propagation. The first optimization is clearly desirable and secure. Not performing the second optimization requires special effort! (GCC's `-fno-delete-null-pointer-checks` option)

Undesirable optimizations in CompCert

CompCert's value analysis is currently too weak to eliminate NULL checks CVE-2009-1879-style. However, similar issues were found with its points-to analysis and bit-level manipulation of pointers (which is undefined behavior in the CompCert formal semantics).

```
x = 1; y = 2;
if (cond) {
  p = (int *) ((uintptr_t) &x / 4 * 4);
                // undefined behavior, hence p ↦ ∅
} else {
  p = &y;       // p ↦ {y}
}
return *p;      // p ↦ {y} hence *p can be replaced by 2
```

I had to weaken the points-to analysis so that $p \mapsto \{\mathtt{prov}(x), y\}$ in the example above. There is no proof this is the right weakening.

Proving more about undefined behaviors

Hard to prove anything without first making u.b. more defined!

- Completely defined whenever it's easy.
  E.g. CompCert defines overflow in signed integer arithmetic.
- Defined nondeterministically.
  E.g. `(uintptr_t) &x` is any integer.
  (Kang, Hur, et al; PLDI 2015)
- Defined symbolically.
  E.g. `(uintptr_t) &x` is any integer but
  `(uintptr_t)&x - (uintptr_t)&x` is always zero.
  (Besson, Blazy, Wilke; ITP 2017)
- Defined as a run-time trap or abort.
  E.g. hardware support for pointer and bounds checking.

Conclusions

In closing. . .

Formal compiler verification in the style of CompCert or CakeML gives many guarantees relevant to safety, but few guarantees relevant to security-beyond-safety.

CompCert tries to handle security code with care, but it's a best effort without confirmation by the proof.

Expecting to get new ideas by the end of this meeting!