Formal verification of program obfuscations

joint work with Roberto Giacobazzi and Alix Trieu

IFIP WG 2.11, 2015-11-10
Background: verifying a compiler

Compiler + proof that the compiler does not introduce bugs

CompCert, a moderately optimizing C compiler usable for critical embedded software

• Fly-by-wire software, Airbus A380 and A400M, FCGU (3600 files): mostly control-command code generated from Scade block diagrams + mini. OS

• Commercially available since 2015 (AbsInt company)

• Formal verification using the Coq proof assistant
Methodology

- The compiler is written inside the purely functional Coq programming language.
- We state its correctness w.r.t. a formal specification of the language semantics.
- We interactively and mechanically prove this.
- We decompose the proof in proofs for each compiler pass.
- We extract a Caml implementation of the compiler.
The formally verified part of the compiler

**CompCert C**
- Side-effects out of expressions
  - Loop simplifications
- CFG construction
  - Expr. Decomp.
- Register allocation (IRC)
- Linearization of the CFG

**CLight**
- Type elimination
- Instruction selection
- Spilling, reloading calling conventions
- Asm code generation

**C#Minor**
- Stack allocation of «&» variables

**RTL**

**CminorSel**

**Cminor**

**LTL**

**LTLin**

**Linear**
- Layout of stack frames

**ASM**

**Mach**

Optimizations: constant prop., CSE, tail calls, (LCM), (software pipelining)
Let’s add some program obfuscations at the Clight source level and prove that they preserve the semantics of Clight programs.
Program obfuscation

\[ P \rightarrow \tau[P] \]
Recreational obfuscation

#define _ -F<00||--F-OO--; int F=00,OO=00;main(){F_OO();printf("%1.3f\n",4.*-F/0O/0O);}F_OO()
{


}

Winner of the 1988 International Obfuscated C Code Contest
Program obfuscation

Goal: protect software, so that it is harder to reverse engineer
→ Create secrets an attacker must know or discover in order to succeed

• Diversity of programs

• A recommended best practice
Program obfuscation: state of the art

• Trivial transformations: removing comments, renaming variables

• Hiding data: constant encoding, string encryption, variable encoding, variable splitting, array splitting, array merging, array folding, array flattening

• Hiding control-flow: opaque predicates, function inlining and outlining, function interleaving, loop transformations, control-flow flattening

```c
int original (int n) {
    return 0;
}
```

```c
int obfuscated (int n) {
    if (((n+1)*n%2==0)
        return 0;
    else return 1;
}
```
Program obfuscation: control-flow graph flattening

```c
int pc = 1;
while (pc != 0) {
    switch (pc) {
        case 1:
            i = 0;
            pc = 2;
            break;
        case 2:
            if (i <= 100)
                pc = 3;
            else
                pc = 0;
            break;
        case 3:
            i++;
            pc = 2;
            break;
    }
}
```
Program obfuscation: control-flow graph flattening

```c
i = 0;
while (i <= 100) {
    i++;
}

int pc = 1;
while (pc != 0) {
    switch (pc) {
    case 1 : {
        i = 0;
        pc = 2;
        break;
    }
    case 2 : {
        if (i <= 100)
            pc = 3;
        else
            pc = 0;
        break;
    }
    case 3 : {
        i++;
        pc = 2;
        break;
    }
    }
```
Obfuscation: issues

- Fairly widespread use, but cookbook-like use

No guarantee that program obfuscation is a semantics-preserving code transformation.

→ Formally verify some program obfuscations

- How to evaluate and compare different program obfuscations?

Standard measures: cost, potency, resilience and stealth.

→ Use the proof to evaluate and compare program obfuscations
The proof reveals the steps that are required to reverse the obfuscation.
Formal verification of control-flow-graph flattening
Clight semantics

Small-step style with continuations, supporting the reasoning on non-terminating programs.

Expressions: 17 rules (big-step)
Statements: 25 rules (small-step)
+ many rules for unary and binary operators, memory loads and stores

\[ k ::= \text{Kstop} | \text{Kseq2} \ k \ (* \text{after s1 in s1;s2} *) \]
\[ | \text{Kloop1} \ s1 \ s2 \ k \ | \text{Kloop2} \ s1 \ s2 \ k \ (* \text{after si in (loop s1 s2)} *) \]
\[ | \text{Kswitch} \ k \ (* \text{catches break statements} *) \]
\[ | \text{Kcall} \ oi \ f \ e \ le \ k \]

\[ \sigma ::= \text{C} \ f \ \text{args} \ k \ m \]
\[ | \text{R} \ \text{res} \ k \ m \]
\[ | \text{S} \ f \ s \ k \ e \ le \ m \]

(step \( \sigma_1 \ \sigma_1' \)) and also (plus \( \sigma_2 \ \sigma_2' \))
Correctness of control-flow flattening

Theorem simulation:
∀ (σ1 σ1':state), step σ1 σ1' ->
∀ (σ2:state), σ1 ≈ σ2 ->
(∃ σ2', plus σ2 σ2' ∧ σ1' ≈ σ2') ∨ (m(σ1') < m(σ1) ∧ σ1' ≈ σ2).

step (S f s1;s2 k e le m) (S f s1 (Kseq s2 k) e le m)
step (S f Skip (Kseq s k) e le m) (S f s k e le m)
Matching relation between semantic states

Starting from the AST of the flattened program, we need to explain how to rebuild the CFG from the generated switch cases.
Matching relations

\[
\frac{obf(f) = [f']}{k, k' \in \{Kstop, Kcall\}} \quad \frac{obf(f) = [f']}{R(v, k, m) \sim R(v', k', m)} \quad \frac{k \simeq k'}{C(f, a, k, m) \sim C(f', a, k', m)} \quad \frac{k \simeq k'}{R(v, k, m) \sim R(v', k', m)}
\]

(1) (2)

\[
\frac{\text{flatten}(pc_f, \text{body}(f), 1, 0) = [ls] \quad s' = \text{while}(pc_f = 0)(\text{switch}pc, ls)}{\forall s_1, s_2, k', \text{context}(k) \in \{\text{Kloop1} s_1 s_2 k'', \text{Kloop2} s_1 s_2 k''\} \implies \exists n_0 \text{ such that} pc_f, k'' \vdash (\text{loop} s_1 s_2) \sim ls[n_0] \land n_0, pc_f, k \vdash s \approx ls[n]}{\text{context}(k) \in \{\text{Kstop, Kcall}\} \implies pc_f, k \vdash s \sim ls[n]}
\]

(3)

Figure 8: Matching between states ($\sigma \sim \sigma'$ relation).

\[
\frac{ls[n] = (e_1 = e_2; pc = \text{next_stmt}; \text{break})}{pc, k \vdash \text{skip} \sim ls[\text{next_stmt}]}
\]

(4)

\[
\frac{ls[n] = (\text{skip}; pc = \text{next_stmt}; \text{break})}{pc, k \vdash s \sim ls[\text{next_stmt}]}
\]

(5)

\[
\frac{pc, k \vdash \text{skip} \sim ls[n]}{pc, k \vdash s \sim ls[n]}
\]

(6)

\[
\frac{pc, k \vdash \text{skip} \sim ls[n]}{pc, k \vdash s \sim ls[n]}
\]

(7)

\[
\frac{ls[n] = (\text{if} b \text{ then } pc = n + 1 \text{ else } pc = n + 1 + |s_1|; \text{break})}{pc, k \vdash s_1 \sim ls[n + 1]}
\]

(8)

\[
\frac{ls[n] = (pc = n + 1; \text{break})}{n, pc, Kloop1 s_1 s_2 k \vdash s_1 \sim ls[n + 1]}
\]

(9)

Figure 9: Matching between statements ($pc, k \vdash s \sim ls[n]$ relation).

\[
\frac{ls[n] = (s_1; s_2 \sim ls[n + 1])}{pc, k \vdash \text{loop} s_1 s_2 \sim ls[n]}
\]

\[
\frac{ls[n] = (pc = n + 1; \text{break})}{n, pc, Kloop2 s_1 s_2 k \vdash \text{skip} \sim ls[n + 1]}
\]

\[
\frac{ls[n] = (pc = n + 1; \text{break})}{n, pc, Kloop1 s_1 s_2 k \vdash \text{skip} \sim ls[n + 1]}
\]

(10) (11) (12)

\[
\frac{ls[n] = (pc = n + 1; \text{break})}{n, pc, Kloop2 s_1 s_2 k \vdash \text{skip} \sim ls[n + 1]}
\]

(13)

\[
\frac{ls[n] = (pc = n + 1; \text{break})}{n, pc, Kloop2 s_1 s_2 k \vdash \text{break} \sim ls[n]}
\]

(14)

Figure 10: Matching between statements ($n_0, pc, k \vdash s \approx ls[n]$ relation).
Implementation and experiments

1200 lines of spec + 4250 lines of proofs + reused CompCert libraries

The comparison with Obfuscator-LLVM revealed a slowdown in the execution of our obfuscated programs, due to a number of skip statements that are generated by the first pass of CompCert.

   Trick to facilitate the proof: use skip statements to materialize evaluation steps of non-deterministic expressions.

Solution: add a pass that eliminates skip statements in skip;s sequences
## Experimental results

<table>
<thead>
<tr>
<th>Program</th>
<th>LoC (2)</th>
<th>Original (3)</th>
<th>Obfuscated (4)</th>
<th>Ratio (5)</th>
<th>Obfuscated (6)</th>
<th>Ratio (7)</th>
<th>LLVM / NO SKIPS + OBF. (8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes.c</td>
<td>1453</td>
<td>1.015</td>
<td>3.256</td>
<td>3.207</td>
<td>2.290</td>
<td>2.256</td>
<td>0.703</td>
</tr>
<tr>
<td>almbench.c</td>
<td>351</td>
<td>0.452</td>
<td>0.781</td>
<td>1.727</td>
<td>0.600</td>
<td>1.327</td>
<td>0.768</td>
</tr>
<tr>
<td>binarytrees.c</td>
<td>164</td>
<td>5.001</td>
<td>6.007</td>
<td>1.201</td>
<td>5.387</td>
<td>1.077</td>
<td>0.896</td>
</tr>
<tr>
<td>bisect.c</td>
<td>377</td>
<td>4.675</td>
<td>10.127</td>
<td>2.166</td>
<td>24.893</td>
<td>5.324</td>
<td>2.457</td>
</tr>
<tr>
<td>chomp.c</td>
<td>368</td>
<td>1.393</td>
<td>4.308</td>
<td>3.092</td>
<td>4.654</td>
<td>3.340</td>
<td>1.080</td>
</tr>
<tr>
<td>fannkuch.c</td>
<td>154</td>
<td>0.265</td>
<td>3.306</td>
<td>12.475</td>
<td>6.504</td>
<td>24.543</td>
<td>1.967</td>
</tr>
<tr>
<td>fft.c</td>
<td>191</td>
<td>0.095</td>
<td>0.161</td>
<td>1.694</td>
<td>0.302</td>
<td>3.178</td>
<td>1.876</td>
</tr>
<tr>
<td>fftsp.c</td>
<td>196</td>
<td>0.001</td>
<td>0.002</td>
<td>2.000</td>
<td>0.004</td>
<td>4.000</td>
<td>2.000</td>
</tr>
<tr>
<td>fftw.c</td>
<td>89</td>
<td>2.059</td>
<td>17.817</td>
<td>8.653</td>
<td>6.419</td>
<td>3.117</td>
<td>0.360</td>
</tr>
<tr>
<td>fib.c</td>
<td>19</td>
<td>0.164</td>
<td>0.395</td>
<td>2.408</td>
<td>0.662</td>
<td>4.036</td>
<td>1.676</td>
</tr>
<tr>
<td>integr.c</td>
<td>32</td>
<td>0.052</td>
<td>0.167</td>
<td>3.211</td>
<td>0.148</td>
<td>2.846</td>
<td>0.886</td>
</tr>
<tr>
<td>knucleotide.c</td>
<td>369</td>
<td>0.080</td>
<td>0.152</td>
<td>1.900</td>
<td>0.138</td>
<td>1.725</td>
<td>0.907</td>
</tr>
<tr>
<td>lists.c</td>
<td>81</td>
<td>0.386</td>
<td>5.047</td>
<td>13.075</td>
<td>1.214</td>
<td>3.145</td>
<td>0.240</td>
</tr>
<tr>
<td>mandelbrot.c</td>
<td>92</td>
<td>1.302</td>
<td>5.559</td>
<td>4.269</td>
<td>24.173</td>
<td>18.566</td>
<td>4.349</td>
</tr>
<tr>
<td>nbody.c</td>
<td>174</td>
<td>4.981</td>
<td>17.062</td>
<td>3.425</td>
<td>17.489</td>
<td>3.511</td>
<td>1.025</td>
</tr>
<tr>
<td>nsieve.c</td>
<td>57</td>
<td>0.113</td>
<td>0.548</td>
<td>4.849</td>
<td>1.497</td>
<td>13.247</td>
<td>2.731</td>
</tr>
<tr>
<td>nsievebits.c</td>
<td>76</td>
<td>0.101</td>
<td>0.389</td>
<td>3.851</td>
<td>0.929</td>
<td>9.198</td>
<td>2.388</td>
</tr>
<tr>
<td>perlin.c</td>
<td>75</td>
<td>8.241</td>
<td>32.876</td>
<td>3.989</td>
<td>53.520</td>
<td>6.494</td>
<td>1.627</td>
</tr>
<tr>
<td>qsort.c</td>
<td>50</td>
<td>0.293</td>
<td>1.342</td>
<td>4.580</td>
<td>2.703</td>
<td>9.225</td>
<td>2.014</td>
</tr>
<tr>
<td>sha3.c</td>
<td>2233</td>
<td>5.202</td>
<td>34.601</td>
<td>6.651</td>
<td>8.321</td>
<td>1.599</td>
<td>0.240</td>
</tr>
</tbody>
</table>
Conclusion

Competitive program obfuscator operating over C programs, integrated in the CompCert compiler

Semantics-preserving code transformation

Future work

- Combine CFG flattening with other simple obfuscations
- The proof measures the difficulty of reverse engineering the obfuscated code.
  - Study how to count the size of lambda-terms
  - Semantics of proofs as independent objects (focused proof systems)
Questions ?