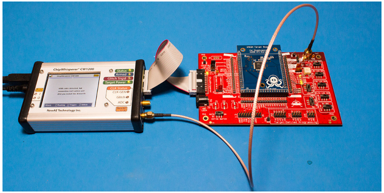
Formally verified hardening of C programs against fault injection

Sylvain Boulmé, David Monniaux, Basile Pesin, Marie-Laure Potet
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG

05/06/2024

Problem statement

Attacks by fault injection



```
#define PIN_LENGTH 4

int verify_pin(char *pin, char *entered) {
  int i, ok = 1; ⚡× 1
  for(i = 0; i < PIN_LENGTH; i++) {
    ~~if(pin[i] != entered[i]) ok = 0;~~
  } ← if(i != PIN_LENGTH) exit(1);
  return ok;
}
```

Inserting and verifying countermeasures

Definition (Countermeasure)

Redundant calculation used to catch the fault. May be inserted:

- in **hardware** (systematically)
- in the **source software** (selectively, by the programmer)
- at **compile-time** (systematically or selectively)

Properties

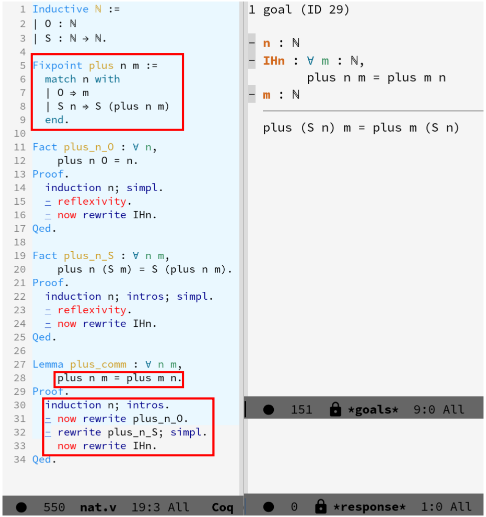
- *Correctness*: preserve the program semantics?
- *Adequacy*: protect from a given attacker model?

We use an interactive theorem prover (Coq)

The Coq Interactive Theorem Prover

[Coq Development Team (2020): The Coq proof assistant reference manual]

- A functional programming language
- ‘Extraction’ to OCaml programs
- A specification language
- Tactic-based interactive proof

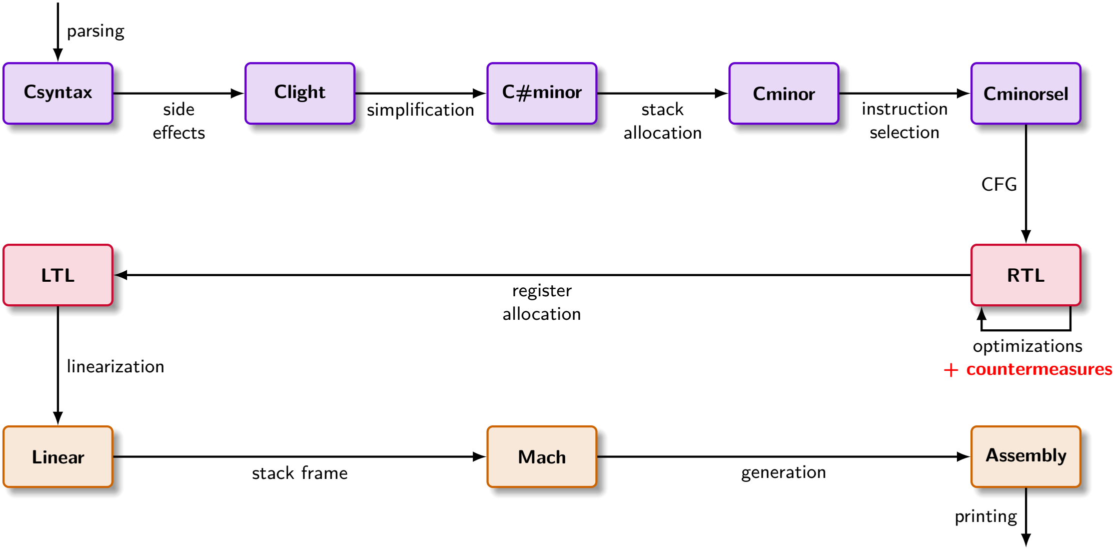


```
Inductive ℕ :=
| O : ℕ
| S : ℕ → ℕ.

Fixpoint plus n m :=
  match n with
  | O ⇒ m
  | S n ⇒ S (plus n m)
  end.

Fact plus_n_O : ∀ n,
    plus n O = n.
Proof.
  induction n; simpl.
  - reflexivity.
  - now rewrite IHn.
Qed.

Fact plus_n_S : ∀ n m,
    plus n (S m) = S (plus n m).
Proof.
  induction n; intros; simpl.
  - reflexivity.
  - now rewrite IHn.
Qed.

Lemma plus_comm : ∀ n m,
    plus n m = plus m n.
Proof.
  induction n; intros.
  - now rewrite plus_n_O.
  - rewrite plus_n_S; simpl.
    now rewrite IHn.
Qed.
```

```
1 goal (ID 29)

n : ℕ
IHn : ∀ m : ℕ,
      plus n m = plus m n
m : ℕ
______________________________________
plus (S n) m = plus m (S n)
```

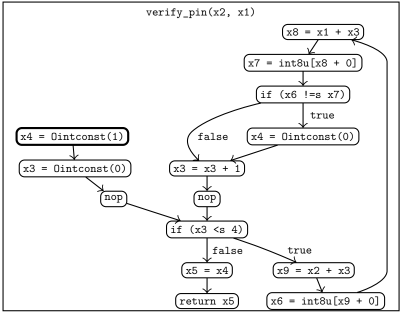
Verified program transformation

The Chamois-CompCert verified compiler



The RTL intermediate language

```
#define PIN_LENGTH 4

int verify_pin(char *pin, char *entered) {
  int i, ok = 1;
  for(i = 0; i < PIN_LENGTH; i++) {
    if(pin[i] != entered[i]) ok = 0;
  }
  return ok;
}
```

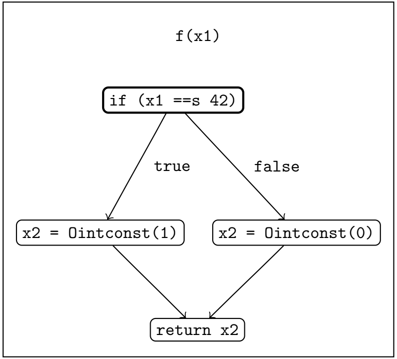


$$
\begin{array}{rcl}
i & ::= & \\
 & | & \textbf{nop}\,(l) \\
 & | & \textbf{op}\,(op, \vec{r}, r, l) \\
 & | & \textbf{load}\,(k, addr, \vec{r}, r, l) \\
 & | & \textbf{store}\,(k, addr, \vec{r}, r, l) \\
 & | & \textbf{call}\,(sig, regid, \vec{r}, r, l) \\
 & | & \textbf{tailcall}\,(sig, regid, \vec{r}) \\
 & | & \textbf{cond}\,(cond, \vec{r}, l_1, l_2) \\
 & | & \textbf{return}\,(r)
\end{array}
$$

$$
\begin{array}{rcl}
g & ::= & \\
 & | & l \mapsto i
\end{array}
$$

Example CM: Control-Flow Checking

[Ferrière (2019): A compiler approach to Cyber-Security]



f(x1)

if (x1 ==s 42)

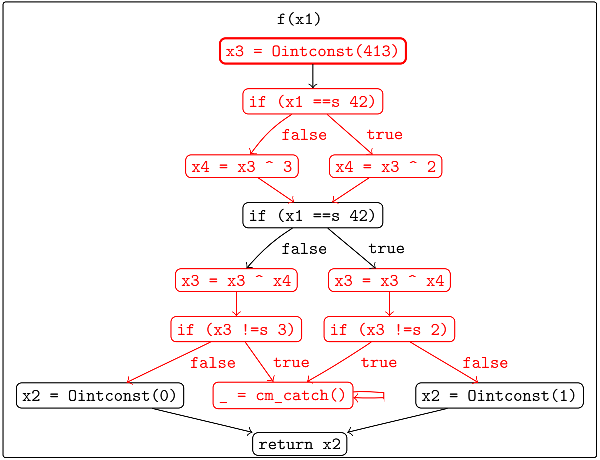
true

false

x2 = Ointconst(1)

x2 = Ointconst(0)

return x2



f(x1)

x3 = Ointconst(413)

if (x1 ==s 42)

false

true

x4 = x3 ^ 3

x4 = x3 ^ 2

if (x1 ==s 42)

false

true

x3 = x3 ^ x4

x3 = x3 ^ x4

if (x3 !=s 3)

if (x3 !=s 2)

false

true

true

false

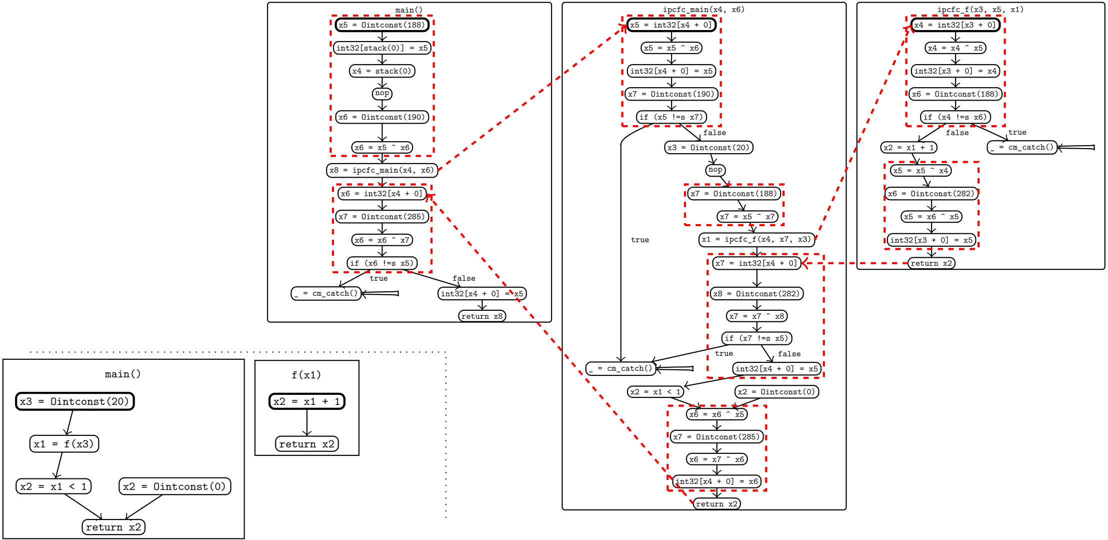
x2 = Ointconst(0)

_ = cm_catch()

x2 = Ointconst(1)

return x2

Example CM: Inter-procedural Control-Flow Checking

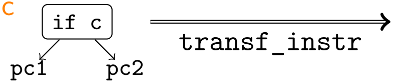


Implementing program transformations

V1: global graph transformation (using state monad) ▶ Tedious reasoning

V2: local instruction-to-sequence rewriting

Specific



if c

pc1 pc2

transf_instr

```
if c { rts = gsr ^ sig1 }
else { rts = gsr ^ sig2 };
if c {
  gsr = gsr ^ rts;
  if (gsr != sig1) { cm_catch(); }
  goto pc1;
} else {
  gsr = gsr ^ rts;
  if (gsr != sig2) { cm_catch(); }
  goto pc2;
}
```

pc1 pc2

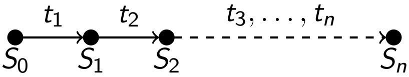
Generic

- → $G \xRightarrow[\texttt{transf}_{TR}]{} G'$
- → $\texttt{spec}_{TR} : \texttt{code} \to \texttt{Prop}$
- → $\forall G,\ \texttt{spec}_{TR}(\texttt{transf}_{TR}(G))$

RTL semantics

Small-step semantics

$$
\begin{array}{lll}
st & ::= & \\
 & | & \mathbf{S}\,(\Sigma, f, \sigma, pc, R, M) \\
 & | & \mathbf{Call}\,(\Sigma, fd, \vec{v}, M) \\
 & | & \mathbf{Return}\,(\Sigma, v, M)
\end{array}
$$



$$\frac{f.\mathbf{code}(pc) = \lfloor \mathbf{op}\,(op, \vec{r}, r, l) \rfloor \qquad \text{eval\_op}(G, \sigma, op, R(\vec{r})) = \lfloor v \rfloor}{G \vdash \mathbf{S}\,(\Sigma, f, \sigma, pc, R, M) \xrightarrow{\epsilon} \mathbf{S}\,(\Sigma, f, \sigma, l, R\{r \leftarrow v\}, M)}$$

$$\frac{f.\mathbf{code}(pc) = \lfloor \mathbf{cond}\,(cond, \vec{r}, l_1, l_2) \rfloor \qquad \text{eval\_condition}(cond, R(\vec{r}), M) = \lfloor b \rfloor}{G \vdash \mathbf{S}\,(\Sigma, f, \sigma, pc, R, M) \xrightarrow{\epsilon} \mathbf{S}\,(\Sigma, f, \sigma, \mathbf{if}\; b\; \mathbf{then}\; l_1\; \mathbf{else}\; l_2, R, M)}$$

Proving a CompCert pass

Each CompCert pass must satisfy a forward simulation:

$$\begin{tikzcd}
S_1 \arrow[r, dash, "\sim"] \arrow[d, "t"'] & S_1' \arrow[d, dashed, "t"', "+"] \\
S_2 \arrow[r, dash, dashed, "\sim"] & S_2'
\end{tikzcd}$$

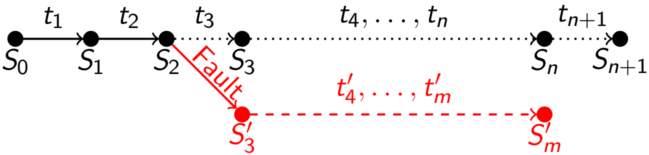
Formally stated:

$$\begin{array}{rl}
\textbf{if} & G \vdash S_1 \xrightarrow{t} S_2 \\
\textbf{and} & \text{match\_states } S_1\ S_1' \\
\textbf{then} & \exists S_2', \quad \text{compile}(G) \vdash S_1' \xrightarrow{t}{}^{+} S_2' \quad \textbf{and} \quad \text{match\_states } S_2\ S_2'
\end{array}$$

Proof of security

RTL semantics with faults

Semantic model extended with fault transitions

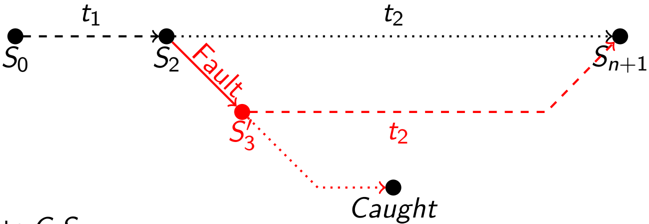


Example (invert conditional):

$$\frac{f.\mathbf{code}(pc) = \lfloor \mathbf{cond}(cond, \vec{r}, l_1, l_2) \rfloor \qquad \text{eval\_condition}(cond, R(\vec{r}), M) = \lfloor b \rfloor}{G \vdash_{\mathrm{F}} \mathbf{S}(\Sigma, f, \sigma, pc, R, M) \xrightarrow{\textbf{[Fault InvertCond]}} \mathbf{S}(\Sigma, f, \sigma, \textbf{if } b \textbf{ then } \mathbf{l_2} \textbf{ else } \mathbf{l_1}, R, M)}$$

Security theorem

We say that a program G is secure against a **single-fault** attack with fault F if:



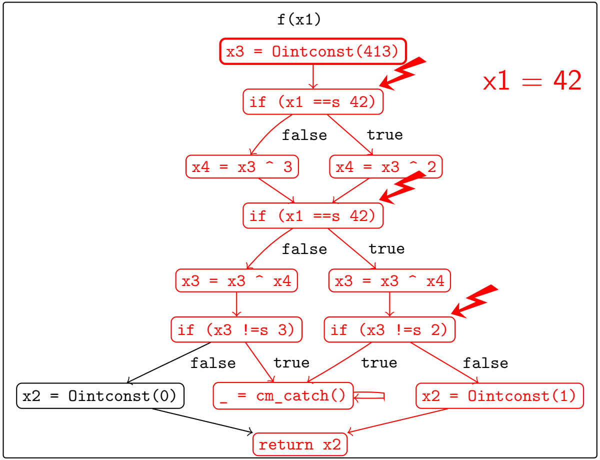
if initial-state $G\ S_0$
and $G \vdash_{\mathrm{F}} S_0 \xrightarrow{t}^{\star} S'_3$
and $t = t_1 + [\textbf{Fault}\ F]$ **and** nofault t_1

then $G \vdash_{\mathrm{F}} S'_3 \xrightarrow{\epsilon}^{\star}$ **Caught**
or $\exists\ S_{n+1}\ t_2,$ nofault t_2 **and** $G \vdash_{\mathrm{F}} st \xrightarrow{t_2}^{\star} S_{n+1}$ **and** $G \vdash S_0 \xrightarrow{t_1+t_2}^{\star} S_{n+1}$

Example: CFC Security proof

- consider all possible points of attacks
- for each attack, reach `catch`
- `gsr`, `rts` depend on past steps...
- invert $G \vdash_{\mathrm{F}} st_0 \xrightarrow{t}{}^{\star} st$
- complex (but reusable?) lemmas

Hypothesis: well-formedness of CFG



f(x1)

x3 = Ointconst(413)

if (x1 ==s 42)

false

true

x4 = x3 ^ 3

x4 = x3 ^ 2

if (x1 ==s 42)

false

true

x3 = x3 ^ x4

x3 = x3 ^ x4

if (x3 !=s 3)

if (x3 !=s 2)

false

true

true

false

x2 = Ointconst(0)

_ = cm_catch()

x2 = Ointconst(1)

return x2

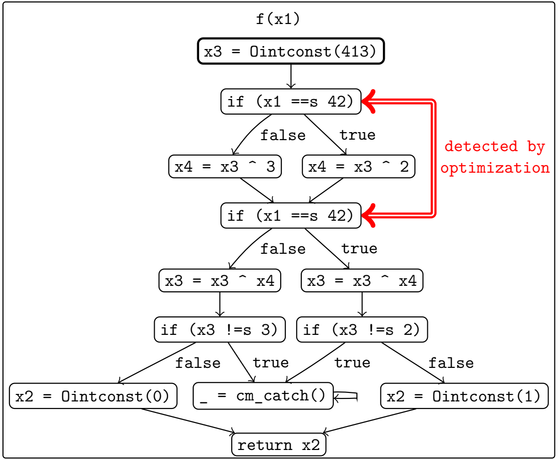
x1 = 42

Tedious proof: 250 reusable LoC + 1100 specific LoC

Experimental Evaluation

Countermeasures and Optimizations

Does the program stay protected ?



f(x1)

x3 = 0intconst(413)

if (x1 ==s 42)

false

true

x4 = x3 ^ 3

x4 = x3 ^ 2

if (x1 ==s 42)

detected by optimization

false

true

x3 = x3 ^ x4

x3 = x3 ^ x4

if (x3 !=s 3)

if (x3 !=s 2)

false

true

true

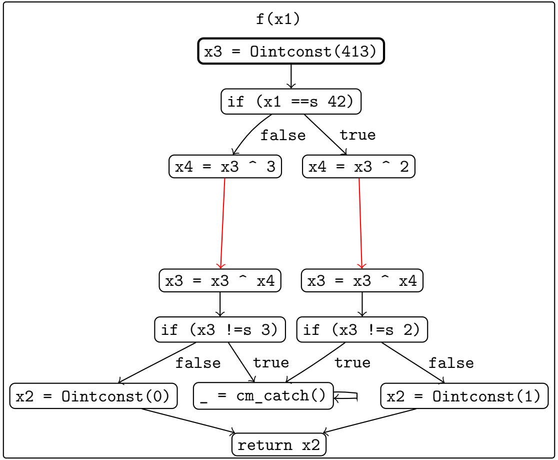
false

x2 = 0intconst(0)

_ = cm_catch()

x2 = 0intconst(1)

return x2



f(x1)

x3 = 0intconst(413)

if (x1 ==s 42)

false

true

x4 = x3 ^ 3

x4 = x3 ^ 2

x3 = x3 ^ x4

x3 = x3 ^ x4

if (x3 !=s 3)

if (x3 !=s 2)

false

true

true

false

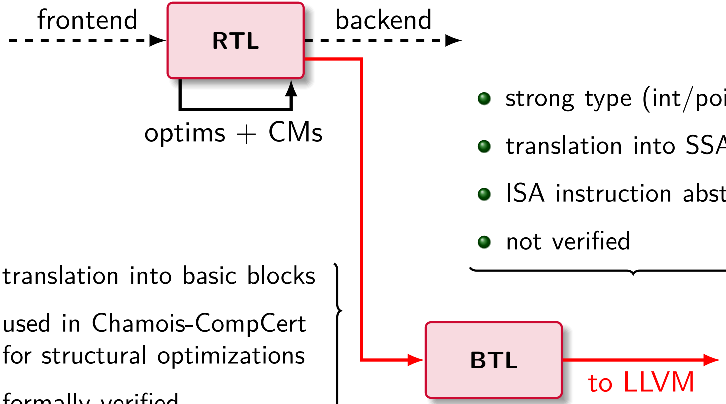
x2 = 0intconst(0)

_ = cm_catch()

x2 = 0intconst(1)

return x2

Interfacing with evaluation tools



- strong type (int/pointer) + int size re-inference
- translation into SSA form
- ISA instruction abstraction into LLVM instructions
- not verified

- translation into basic blocks
- used in Chamois-CompCert for structural optimizations
- formally verified (by translation validation)

LAZART

simulating fault injection
by symbolic execution

Preliminary experimental results

We tested CompCert CMs on some Lazart test cases:

		No CM with -O0			CM with -O0			CM with -O1		
Program	Type	#IP	1F	2F	#IP	1F	2F	#IP	1F	2F
`aes_round_key`	TI	1	16	0	4	**0**	32	3	**16**	0
`verify_pin`	TI	4	3	3	16	**0**	6	15	**1**	4
`memcmps`	Data Load	4	2	4	6	**0**	2	2	**2**	4

Optimizations do break our countermeasures!

Conclusion

Contributions

We proposed a methodology to formally verified software countermeasures

- a framework for local graph rewriting
- a scheme for defining attacker models
- definitions and tactics for proving the adequacy of a countermeasure
- a methodology for experimental evaluation of the compilation chain

We applied this methodology to two countermeasures

- Intra-procedural control-flow checking
- Inter-procedural control-flow checking

Perspectives

- Develop attacker model and adequacy proof for Inter-Procedural CFC
  - skip call
  - call to the wrong function
- Apply our evaluation technique to more examples
- Test by simulation at the binary level
  - RTL is only the middle of the compiler: later passes may break CMs
  - using BINSEC? [David, Bardin, Ta, Mounier, Feist, Potet, and Marion (2016): BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis]
- Develop a methodology to protect the CMs from optimizations
  - following [Vu, Heydemann, Grandmaison, and Cohen (2020): Secure delivery of program properties through optimizing compilation]
  - mechanized as a hyper-property of the semantics ?

Thank You! Questions?

Please visit our GitLab repository:

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

Some PhD/Postdoc positions are available!

`https://www-verimag.imag.fr/`