Security Properties

Information Flow

- A class of security policies that constrain the ways in which information can be manipulated during program execution
- Usually formulated in terms of *non-interference* between *confidential inputs* (high-security variables) and *public outputs* (low-security variables)
- Typically specified using some augmented type system

Example (indirect flow)

```
void fibonacci ()
{
  while (n > 0) {
    f1 = f1 + f2;
    f2 = f1 - f2;
    n--;
  }
  if (f1 > k) l = 1;
  else l = 0;
}
```

Let n be high and l be low
(others we don't care)

the program is *insecure*
and can be typed as so

Example (direct flow)

```
n = 1;
if (k)  x = 1;
if (!k) x = n;
l = x+y;
```

Let n be high and l be low
(others we don't care)

is the program secure?

Non-Interference (Semantic Formulation)

A semantics- (rather than type-) based formulation of non-interference treats cases like the previous example correctly

$$\sigma \overset{V_L \cup V_?}{=} \tau \;\wedge\; (C,\sigma) \Downarrow \sigma' \;\wedge\; (C,\tau) \Downarrow \tau' \quad \Longrightarrow \quad \sigma' \overset{V_L}{=} \tau'$$

(termination-insensitive formulation)

Self-composition

- A technique that allows to specify non-interference using axiomatic semantics (Hoare logic)
- *Informally:* Given program C, we construct the program C; C' where C' is a copy of C with every variable *x* renamed to *x'*.
- C is secure if,
  when the program C;C' is run from a state in which *x*=*x'* for all non-high security *x*,
  then when (if!) it stops, *l*=*l'* for every low security variable *l*

Self-composition

- Can be formalized using contracts and JML / ACSL

$$\left\{ \bigwedge_{x \in V_L \cup V_?} x = x^s \right\} C; C^s \left\{ \bigwedge_{x \in V_L} x = x^s \right\}$$

```
/*@ requires n>=0 && ns>=0 && l == ls && f1 == f1s
             && f2 == f2s && k == ks && n==ns
  @ ensures l == ls
  @*/
void fibonacci_verif ()
{
  while (n > 0) {
    f1 = f1 + f2;
    f2 = f1 - f2;
    n--;
  }
  if (f1 > k) l = 1;
  else l = 0;

  while (ns > 0) {
    f1s = f1s + f2s;
    f2s = f1s - f2s;
    ns--;
  }
  if (f1s > ks) ls = 1;
  else ls = 0;
}
```

n=ns is required to get *l=ls* thus information flows from n to l

How can this be annotated to achieve automatic proofs?

Usual (functional) invariants will not do…

(paper *Deductive Verification of Cryptographic Software, NFM'09*)

Program Equivalences

```
/*@ requires 0<=i<=j && 0<=is<=js &&
  @          i==is && j==js && k==ks && a==as && b==bs;
  @ ensures  i==is && j==js && k==ks && a==as && b==bs;
  @*/
void main_verif ()
{

  for (k=i ; k<=j; k++) b += k;
  for (k=i ; k<=j; k++) a *= k;

  for (ks=is ; ks<=js; ks++) {
    bs += ks;
    as *= ks;
  }
}
```

How can this be annotated to achieve automatic proofs?


High Assurance Compilation

High Assurance Compilation

What is a correct compiler?
Let $C = Comp(S)$

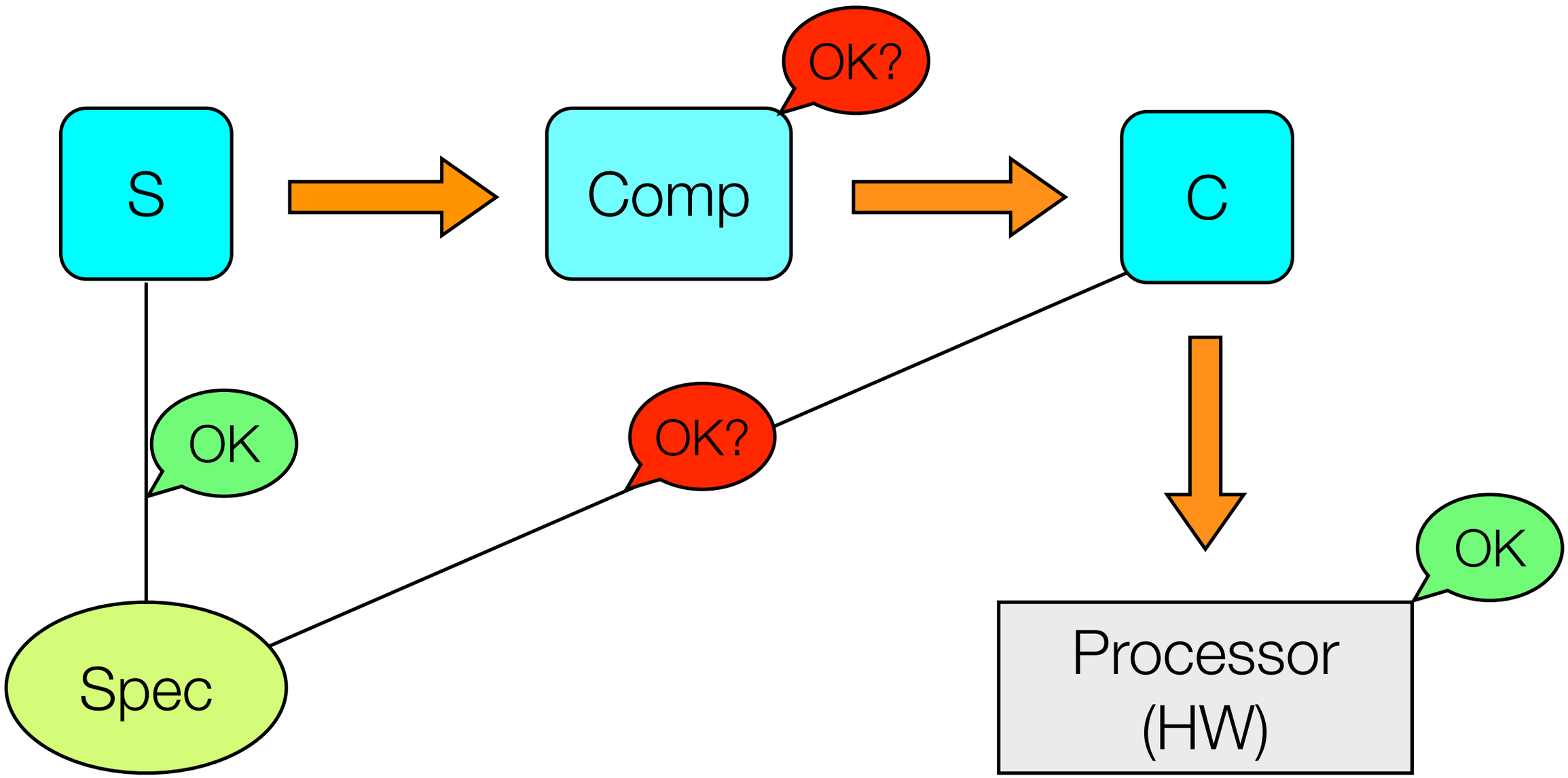
- Semantic transparency: the behaviour of C should be the one specified by the semantics of S… or,

- If S is correct w.r.t. to a given spec. then so should C

- *Testing* C will not differentiate bugs in S and in $Comp$, and may of course miss bugs

Critical Software

Formal methods used



Compiler Correctness Properties (Leroy, POPL'06)

We write

OK(S) for "S does not go wrong" and

OK(S,spec) for "S satisfies the specification spec"

Compiler correctness as *preservation of specifications*:

If OK(S) and OK(S,spec) then OK(C,spec)

A particular case is

(type, memory) safety of S implies safety of C

Compiler Correctness Properties

Most work assumes that specs. depend exclusively on the *observable behaviour* of programs

Under this assumption, preservation of specifications is a consequence of the following property:

If OK(S) then S and C are observationally equivalent.

A compiler for which a monolithic proof of this property can be produced is a *verified compiler*

Verifying Compiler

A *verifying compiler* produces, *for each source S*, not only executable C but also a proof that one of the previous notions of correspondence between S and C holds:

- if OK(S) then S and C are observationally equivalent.

- If OK(S) and OK(S,spec) then OK(C,spec)

Proofs not as difficult as verifying the compiler monolithically

Automation (little user intervention) should be a goal

Verifying Compiler

- A challenge since 1967 (Floyd)

- Proposed by Hoare as a grand challenge for computing research (2003)

- Many recent advances

Pike et al (ACL2'06) have produced a verifying core for the μCryptol cryptographic language

Transformation Verification

- Constructing a verifying compiler is *compositional*

- For each transformation step S -> S', both S and the generated code S' must be somehow *embedded* in the language of the selected proof tool
  (the semantics of S, S' are described in that language)

- Following this, a set of theorems is generated corresponding to, say, obs. equivalence of S,S'

- Finally, proofs of these theorems are automatically constructed

Certifying Compiler

Certificate: a representation of a proof as a (type theory) term. Proofs can thus be *checked* against the theorems they prove!

It is unclear (to me!) if the name *certifying compiler* appeared only to designate a verifying compiler based on certificates.

Its use is also associated with properties that regard the target code only:

- OK(C,spec)

- C is (type, memory) safe

Proof-carrying Code

- Certificates generated by a certifying compiler can be *checked* before execution (by trusted proof checker)

- Trust in compiled code becomes independent of source code and compilation process

- For example, for a memory safety policy, it would suffice to trust a VCGen that produces the appropriate safety conditions from the compiled code

Certificate Translation (Barthe et al, SAS'06)

For properties like

- OK(C,spec)

for which interactive proofs are required, it is much more convenient to conduct such proofs (and produce the corresponding certificates) at source level.

The verification conditions for spec may be modified by transformations performed by the compiler.

Certificate translation refers to the production of certificates by translation from certificates constructed at source level

Translation Validation

A translation validator is a program that takes source S and executable C and determines by static analysis that they are related in the desired way, e.g. they are observationally equivalent.

This approach requires *verification of the validator* Verif: it must be proved that when

$$\text{Verif}(S,C) = \text{true}$$

the correspondence is indeed valid.

Leroy's Unified View

Verifier can be generalized by taking as arguments not only S and C (as in translation validation), but also a certificate A

Verifier Correctness:

Verif (S,C,A) should return true only if S and C are related in the desired way [obs. equiv / spec. preserv. / OK(C,spec)]

In PCC A is a proof term and S is not used.
In TV A is empty and S is required.

Other notions of certificate (partial annotations allowing for the reconstruction of proof) stand midway

Certification Steps (Leroy, POPL'06)

- Constructing and proving a certifying compiler is a compositional process

- Different techniques may be used for each step, roughly divided into

  - certified transformations (as before), and

  - a posteriori checking with a certified verifier (as in TV)

- Not only the optimising steps are difficult to prove!