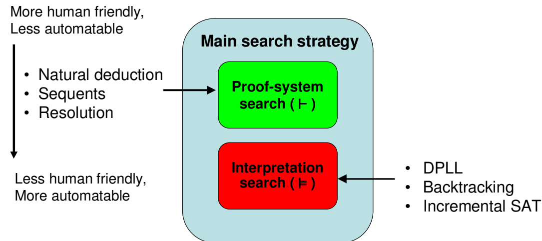
Main search strategy review



Comparison between the two domains

Comparison between the two domains

- Advantages of the interpretation domain
  - Don't have to deal with inference rules directly
    - less syntactic overhead, can build specialized data structures
  - Can more easily take advantage of recent advanges in SAT solvers
  - A failed search produces a counter-example, namely the interpretation that failed to make the formula true or false (depending on whether the goal is to show validity of unsatisfiability)

Comparison between the two domains

- Disadvantages of the interpretation domain
  - Search does not directly provide a proof
    - There are ways to get proofs, but they require more effort
  - Proofs are useful
    - Proof checkers are more efficient than proof finders (PCC)
    - Provide feedback to the human user of the theorem prover
      - Find false proofs cause by inconsistent axioms
      - See path taken, which may point to ways of formulating the problem to improve efficiency
    - Provide feedback to other tools
      - Proofs from a decision procedure can communicate useful information to the heuristic theorem prover

Comparison between the two domains

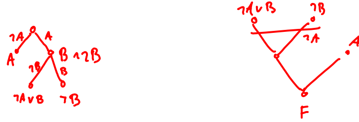
- Disadvantages of the interpretation domain (contd)
  - A lot harder to make the theorem prover interactive
  - Fairly simple to add user interaction when searching in the proof domain, but this is no the case in the interpretation domain
  - For example, when the Simplify theorem prover finds a false counter-example, it is in the middle of an exhaustive search. Not only is it hard to expose this state to the user, but it's also not even clear how the user is going to provide guidance

Connection between the two domains

- Are there connections between the techniques in the two domains?
- There is at least one strong connection, let's see what it is.

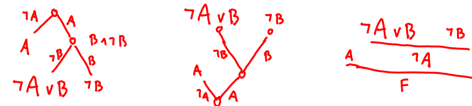
Let's go back to interpretation domain

- Show that the following in UNSAT (also, label each leaf with one of the original clauses that the leaf falsifies):
  - $A \wedge (\neg A \vee B) \wedge \neg B$



Let's go back to interpretation domain

- Show that the following in UNSAT (also, label each leaf with one of the original clauses that the leaf falsifies):
  - $A \wedge (\neg A \vee B) \wedge \neg B$



Parallel between DPLL and Resolution

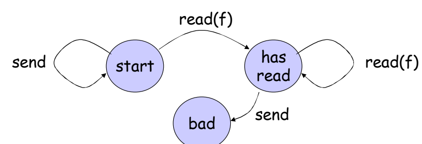
- A successful refutation DPLL search tree is isomorphic to a refutation based resolution proof
- From the DPLL search tree, one can build the resolution proof
  - Label each leaf with one of the original clauses that the leaf falsifies
  - Perform resolution based on the variables that DPLL performed a case split on
- One can therefore think of DPLL as a special case of resolution

Connection between the two domains

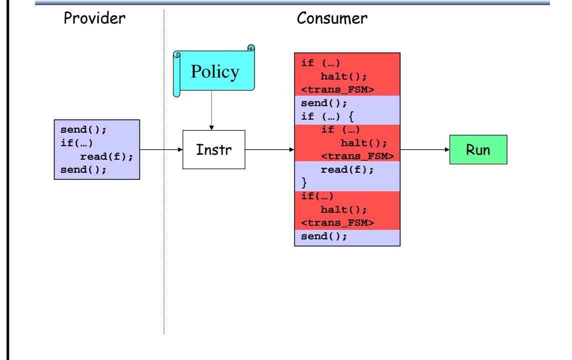
- Are there any other connections between interpretation searches and proof system searches?
  - Such connections could point out new search strategies (eg: what is the analog in the proof system domain of Simplify's search strategy?)
  - Such connections could allow the state of theorem prover to be switched back and forth between the interpretation domain and proof system domain, leading to a theorem prover that combines the benefits of the two search strategies

Proof Carrying Code

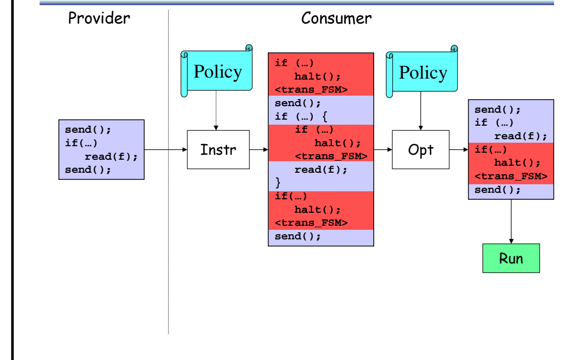
Security Automata



Example



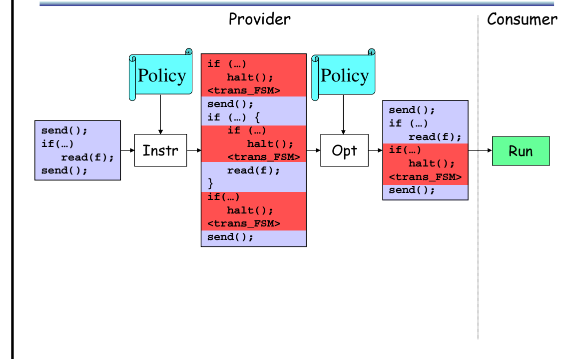
Example



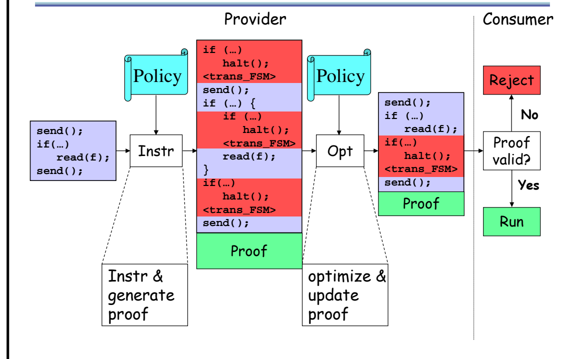
Optimize: how?

- Use a dataflow analysis
  - Determine at each program point what state the security automata may be in
- Based on this information, can remove checks that are known to succeed

Example

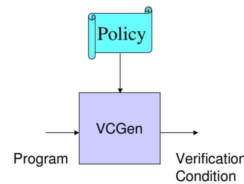


Example

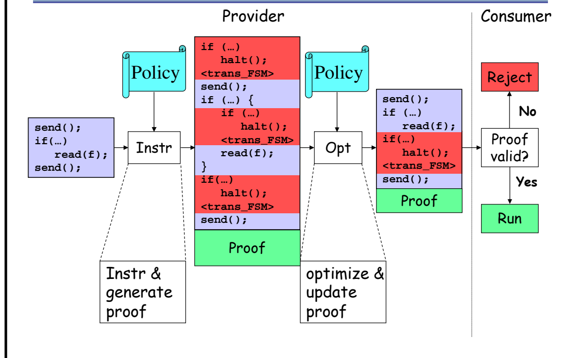


Proofs: how?

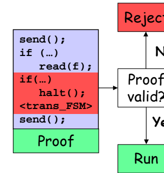
- Generate verification condition
  - Include proof of verification in binary



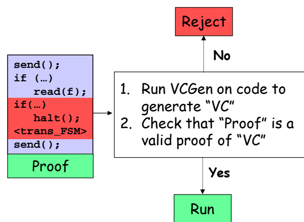
Example



Example



Example



VCGen

- Where have we seen this idea before?

VCGen

- Where have we seen this idea before?
  - ESC/Java
- For a certain class of policies, we can use a similar approach to ESC/Java
- Say we want to guarantee that there are no NULL pointer dereferences
  - Add assert(p != NULL) before every dereference of p
  - The verification condition is then the weakest precondition of the program with respect to TRUE

Simple Example

```
a := &b
if (x >= 0)
   if (x < 0)
      a := NULL
assert(a != NULL)
c := *a
```

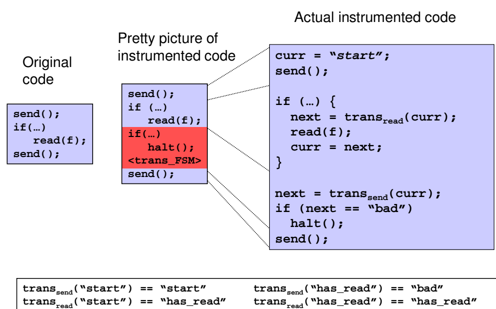
Simple Example

```
(x >= 0 ∧ x < 0) => FALSE ∧
!(x >= 0 ∧ x < 0) => TRUE
                        a := &b
                        if (x >= 0)
                           if (x < 0)
                              a := NULL
a ≠ NULL
                        assert(a != NULL)
                        c := *a
```

For the security automata example

- We will do this example differently
- Instead of providing a proof of WP for the whole program, the provider annotates the code with predicates, and provides proofs that the annotations are locally consistent
- The consumer checks the proofs of local consistency
- We use the type system of David Walker, POPL 2000, for expressing security automata compliance

For the security automata example



Secure functions

- Each security relevant operation requires pre-conditions, and guarantees post-conditions.
- For any "alphabet" function "**func**":
  - P_1: `in(current_state)`
  - P_2: `next_state == trans_func(current_state)`
  - P_3: `next_state != "bad"`

  Pre: `{P1; P2; P3}`

  Execute `func`

  Post: `{in(next_state)}`

Secure functions

- Example for function send()

```
{in(curr); next == trans_send(curr); next != "bad"}
send();
{in(next)}
```

- Normal WP rules apply for other statements, for example:

```
{in(curr)}
next = trans_send(curr);
{in(curr); next == trans_send(curr)}
```

Example

```
curr = "start";
send();

if (…) {
  next = trans_read(curr);
  read(f);
  curr = next;
}

next = trans_send(curr);
if (next == bad)
  halt();
send();
```

Example

```
curr = "start";
send();

if (…) {
  next = trans_read(curr);
  read(f);
  curr = next;
}

next = trans_send(curr);
if (next == bad)
  halt();
send();
```

```
...
next = trans_send(curr);
if (next == bad)
  halt();
send();
```

```
...
{in(curr)}
next = trans_send(curr);
{in(curr); next == trans_send(curr)}
if (next == bad) {
   {in(curr); next == trans_send(curr); next == "bad"}
   halt();
   {}
}
{in(curr); next == trans_send(curr); next != "bad"}
send();
```

```
curr = start;
send();

if (…) {
  next = trans_read(curr);
  read(f);
  curr = next;
}

next = trans_send(curr);
if (next == bad)
  halt();
send();
```

```
{in("start")}
curr = "start";
{curr == "start";
 in(curr); curr == trans_send(curr); curr != "bad"}
send();
{in(curr); curr == "start"}

if (…) {
   {in(curr); curr == "start"}
   next = trans_read(curr);
   {curr == "start"; next == "has_read";
    in(curr); next == trans_read(curr); next != "bad"}
   read(f);
   {in(next); next == "has_read"}
   curr = next;
   {in(curr); curr == "has_read"}
}

{in(curr)}
next = trans_send(curr);
if (next == bad)
  halt();
send();
```

```
Recall:
   trans_send("start") == "start"        trans_read("start") == "has_read"
```

What to do with the annotations?

- The code provider:
  - Send the annotations with the program
  - For each statement: `{ P } S { Q }`
  - Send a proof of P $\Rightarrow$ wp(S, Q)
- The code consumer
  - For each statement: `{ P } S { Q }`
  - Check that the provided proof of P $\Rightarrow$ wp(S, Q) is correct

---

PCC issues: Generating the proof

- Cannot always generate the proof automatically
- Techniques to make it easier to generate proof
  - Can have programmer provide hints
  - Can automatically modify the code to make the proof go through
  - Can use type information to generate a proof at the source code, and propagate the proof through compilation (TIL: Type Intermediate Language, TAL: Typed Assembly Language)

Object P
int op

PCC issues: Representing the proof

- Can use the LF framework
  - Proofs can be large
- Techniques for proof compression
  - Can remove steps from the proof, and let the checker infer them
  - Tradeoff between size of proof and speed of checker

PCC issues: Trusted Computing Base

- What's trusted?

PCC issues: Trusted Computing Base

- What's trusted?
  - Proof checker
  - VCGen (encodes the semantics of the language)
  - Background axioms

Foundational PCC

- Try to reduce the trusted computing base as much as possible
- Express semantics of machine instructions and safety properties in a foundational logic
  - This logic should be suitably expressive to serve as a foundation for mathematics
  - Few axioms, making the proof checker very simple
- No VCGen. Instead just provide a proof in the foundational proof system that the safety property holds
- Trusted computed base an order of magnitude smaller than regular PCC

The big questions

(which you should ask yourself when you review a paper for this class)

- What problem is this solving?
- How well does it solve the problem?
- What other problems does it add?
- What are the costs (technical, economic, social, other) ?
- Is it worth it?
- May this eventually be useful?