Packet Filters

A very simple view of packet filters:
- Some bits come in off the wire
- Some application(s) want the “packet” and some do not (e.g., port number)
- For safety, only the O/S can access the wire
- For extensibility, the applications accept/reject packets

Conventional solution goes to user-space for every packet and app that wants (any) packets

Faster solution: Run app-written filters in kernel-space

What we need

Now the O/S writer is defining the packet-filter language!

Properties we wish of (untrusted) filters:
1. Do not corrupt kernel data structures
2. Terminate (within a time bound)
3. Run fast (the whole point)

Should we download some C/assembly code? (Get 1 of 3)

Should we make up a language and “hope” it has these properties?
Language-based approaches

1. Interpret a language
   + clean operational semantics, + portable, - may be slow (+ filter-specific optimizations), - unusual interface

2. Translate a language into C/assembly
   + clean denotational semantics, + employ existing optimizers, - upfront cost, - unusual interface

3. Require a conservative subset of C/assembly
   + normal interface, - too conservative w/o help

IMP has taught us about (1) and (2) — we’ll get to (3)

A General Pattern

Packet filters move the code to the data rather than data to the code

General reasons: performance, security, other?

Other examples:
  ▶ Query languages
  ▶ Active networks
  ▶ Client-side web scripts (Javascript)

Equivalence motivation

- Program equivalence (we change the program):
  ▶ code optimizer
  ▶ code maintainer

- Semantics equivalence (we change the language):
  ▶ interpreter optimizer
  ▶ language designer
    ▶ (prove properties for equivalent semantics with easier proof)

Note: Proofs may seem easy with the right semantics and lemmas
  ▶ (almost never start off with right semantics and lemmas)

Note: Small-step operational semantics often has harder proofs, but models more interesting things
What is equivalence?
Equivalence depends on what is observable!
▶ Partial I/O equivalence (if terminates, same ans)
  ▶ **while 1 skip** equivalent to everything
  ▶ not transitive
▶ Total I/O equivalence (same termination behavior, same ans)
▶ All (almost all?) variables have the same value
▶ Equivalence plus complexity bounds
▶ Is $O(2^n)$ really equivalent to $O(n)$?
▶ Is "runs within 10ms of each other" important?
▶ Syntactic equivalence (perhaps with renaming)
  ▶ Too strict to be interesting?
In PL, equivalence most often means total I/O equivalence
What is equivalence?

Equivalence depends on what is observable!

- Partial I/O equivalence (if terminates, same ans)
  - while 1 skip equivalent to everything
  - not transitive
- Total I/O equivalence (same termination behavior, same ans)
- Total heap equivalence (same termination behavior, same heaps)
  - All (almost all?) variables have the same value

- Equivalence plus complexity bounds
  - Is $O(2^n)$ really equivalent to $O(n)$?
  - Is “runs within 10ms of each other” important?

- Syntactic equivalence (perhaps with renaming)
  - Too strict to be interesting?

In PL, equivalence most often means total I/O equivalence

Zach Tatlock
CSE-505 2016, Lecture 6
8
Program Example: Strength Reduction

Motivation: Strength reduction
  ▶ A common compiler optimization due to architecture issues

Theorem: \( H \; e \star 2 \downarrow c \) if and only if \( H \; e + e \downarrow c \)

Proof sketch:
  ▶ Prove separately for each direction
  ▶ Invert the assumed derivation, use hypotheses plus a little math to derive what we need
  ▶ Hmm, doesn’t use induction. That’s because this theorem isn’t very useful...

Zach Tatlock
CSE-505 2016, Lecture 6
Program Example: Nested Strength Reduction

Theorem: If \( e' \) has a subexpression of the form \( e \ast 2 \),
then \( H ; e' \Downarrow c' \) if and only if \( H ; e'' \Downarrow c' \)
where \( e'' \) is \( e' \) with \( e \ast 2 \) replaced with \( e + e \)

First some useful metanotation:
\[
C ::= [\cdot] \mid C + e \mid e + C \mid C \ast e \mid e \ast C
\]

\( C[e] \) is “\( C \) with \( e \) in the hole” (inductive definition of “stapling”)

Crisper statement of theorem:
\( H ; C[e \ast 2] \Downarrow c' \) if and only if \( H ; C[e + e] \Downarrow c' \)

Proof sketch: By induction on structure (“syntax height”) of \( C \)
\>
\>

Proof reuse

As we cannot emphasize enough, proving is just like programming

The proof of nested strength reduction had nothing to do with 
\( e \ast 2 \) and \( e + e \) except in the base case where we used our previous theorem

A much more useful theorem would parameterize over the base 
case so that we could get the “nested \( X \)” theorem for any 
appropriate \( X \):

If \( (H ; e_1 \Downarrow c \text{ if and only if } H ; e_2 \Downarrow c) \),
then \( (H ; C[e_1] \Downarrow c' \text{ if and only if } H ; C[e_2] \Downarrow c') \)

The proof is identical except the base case is “by assumption”
Proof, part 1
First assume \( H \downarrow c \) and show \( \exists n. H ; e \rightarrow^n c \)

Lemma (prove it!): If \( H ; e \rightarrow^n e' \), then \( H ; e + e_2 \rightarrow^n e' + e_2 \)

- Proof by induction on \( n \)
- Inductive case uses SLEFT and SRIGHT
First assume $H; e \downarrow c$ and show $\exists n. H; e \rightarrow^n c$

Lemma (prove it!): If $H; e \rightarrow^n e'$, then $H; e_1 + e \rightarrow^n e_1 + e'$ and $H; e + e_2 \rightarrow^n e' + e_2$.

- Proof by induction on $n$
- Inductive case uses SLEFT and SRIGHT

Given the lemma, prove by induction on derivation of $H; e \downarrow c$

- CONST: Derivation with CONST implies $e = c$, and we can derive $H; c \rightarrow^0 c$
- VAR: Derivation with VAR implies $e = x$ for some $x$ where $H(x) = c$, so derive $H; e \rightarrow^1 c$ with SVAR

First assume $H; e \downarrow c$ and show $\exists n. H; e \rightarrow^n c$

Lemma (prove it!): If $H; e \rightarrow^n e'$, then $H; e_1 + e \rightarrow^n e_1 + e'$ and $H; e + e_2 \rightarrow^n e' + e_2$.

- Proof by induction on $n$
- Inductive case uses SLEFT and SRIGHT

Given the lemma, prove by induction on derivation of $H; e \downarrow c$

- CONST: Derivation with CONST implies $e = c$, and we can derive $H; c \rightarrow^0 c$
- VAR: Derivation with VAR implies $e = x$ for some $x$ where $H(x) = c$, so derive $H; e \rightarrow^1 c$ with SVAR
- ADD: ...
Part 1, continued

First assume \( H ; e \Downarrow c \) and show \( \exists n. H ; e \Rightarrow^n c \)

Lemma (prove it!): If \( H ; e \Rightarrow^n e' \), then \( H ; e_1 + e \Rightarrow^n e_1 + e' \) and \( H ; e + e_2 \Rightarrow^n e' + e_2 \).

Given the lemma, prove by induction on derivation of \( H ; e \Downarrow c \)

\[ \begin{align*}
\text{• ADD: Derivation with ADD implies } e &= e_1 + e_2, c = c_1 + c_2, \\
&H ; e_1 \Downarrow c_1, \text{ and } H ; e_2 \Downarrow c_2 \text{ for some } e_1, e_2, c_1, c_2. \\
&\text{By induction (twice), } \exists n_1, n_2. H ; e_1 \Rightarrow^{n_1} c_1 \text{ and } \\
&H ; e_2 \Rightarrow^{n_2} c_2. \\
&\text{So by our lemma } H ; e_1 + e_2 \Rightarrow^{n_1} c_1 + e_2 \text{ and } \\
&H ; c_1 + e_2 \Rightarrow^{n_2} c_1 + c_2.
\end{align*} \]
Proof, part 2

Now assume ∃n. H; e →^n c and show H ; e ⇓c.

Proof by induction on n:

▶ n = 0: e is c and const lets us derive H ; c ⇓c.

Lemma (prove it!): If H; e →^n e′, then H; e1 + e →^n e1 + e1 + e2.

By induction (twice), H; e1 + e2 →^n1+n2 c.

So by our lemma H; e1 + e2 →^n1+n2+1 c.

Proof, part 2

Now assume ∃n. H; e →^n c and show H ; e ⇓c.

Proof by induction on n:

▶ n = 0: e is c and const lets us derive H ; c ⇓c
Proof, part 2

Now assume \( \exists n. H; e \rightarrow^n c \) and show \( H; e \downarrow c \).

Proof by induction on \( n \):

▶ \( n = 0 \): \( e \) is \( c \) and \textsc{const} lets us derive \( H; c \downarrow c \)

▶ \( n > 0 \): (Clever: break into \textit{first} step and remaining ones)
  \( \exists e'. H; e \rightarrow e' \) and \( H; e' \rightarrow^{n-1} c \).
  
  By induction \( H; e' \downarrow c \).
  
  So this lemma suffices: If \( H; e \rightarrow e' \) and \( H; e' \downarrow c \), then \( H; e \downarrow c \).
Part 2, key lemma

Lemma: If $H; e \rightarrow e'$ and $H; e' \downarrow c$, then $H; e \downarrow c$.

Prove the lemma by induction on derivation of $H; e \rightarrow e'$:

- **Svar**: Derivation with `svar` implies $e$ is some $x$ and $e' = H(x) = c$, so derive, by `var`, $H; x \downarrow H(x)$.
- **Sadd**: Derivation with `sadd` implies $e$ is some $c_1 + c_2$ and $e' = c_1 + c_2 = c$, so derive, by `add` and two `const`, $H; c_1 + c_2 \downarrow c_1 + c_2$.

Part 2, key lemma

Lemma: If $H; e \rightarrow e'$ and $H; e' \downarrow c$, then $H; e \downarrow c$.

Prove the lemma by induction on derivation of $H; e \rightarrow e'$:

- **Svar**: Derivation with `svar` implies $e$ is some $x$ and $e' = H(x) = c$, so derive, by `var`, $H; x \downarrow H(x)$.
- **Sadd**: Derivation with `sadd` implies $e$ is some $c_1 + c_2$ and $e' = c_1 + c_2 = c$, so derive, by `add` and two `const`, $H; c_1 + c_2 \downarrow c_1 + c_2$.
- **Sleft**: Derivation with `sleft` implies $e = e_1 + e_2$ and $e' = e'_1 + e_2$ and $H; e_1 \rightarrow e'_1$ for some $e_1, e_2, e'_1$.  

Zach Tatlock
CSE-505 2016, Lecture 6
17

Zach Tatlock
CSE-505 2016, Lecture 6
17
Part 2, key lemma

Lemma: If $H; e \rightarrow e'$ and $H; e' \downarrow c$, then $H; e \downarrow c$.

Prove the lemma by induction on derivation of $H; e \rightarrow e'$:

- **SVAR**: Derivation with SVAR implies $e$ is some $x$ and $e' = H(x) = c$, so derive, by VAR, $H; x \downarrow H(x)$.
- **SADD**: Derivation with SADD implies $e$ is some $c_1 + c_2$ and $e' = c_1 + c_2 = c$, so derive, by ADD and two CONST, $H; c_1 + c_2 \downarrow c_1 + c_2$.
- **SLEFT**: Derivation with SLEFT implies $e = e_1 + e_2$ and $e' = e_1' + e_2$ and $H; e_1 \rightarrow e_1'$ for some $e_1, e_2, e_1'$. Since $e' = e_1' + e_2$ inverting assumption $H; e' \downarrow c$ gives $H; e_1' \downarrow c_1$, $H; e_2 \downarrow c_2$ and $c = c_1 + c_2$.

Applying the induction hypothesis to $H; e_1 \rightarrow e_1'$ and $H; e_1' \downarrow c_1$ gives $H; e_1 \downarrow c_1$.

So use ADD, $H; e_1 \downarrow c_1$, and $H; e_2 \downarrow c_2$ to derive $H; e_1 + e_2 \downarrow c_1 + c_2$.
The cool part, redux

Step through the `sleft` case more visually:

By assumption, we must have derivations that look like this:

\[
\begin{align*}
H; e_1 & \rightarrow e'_1 \\
H; e_1 + e_2 & \rightarrow e'_1 + e_2 \\
H; e_1 & \rightarrow c_1 \\
H; e_2 & \rightarrow c_2 \\
H; e_1 + e_2 & \downarrow c_1 + c_2
\end{align*}
\]

Grab the hypothesis from the left and the left hypothesis from the right and use induction to get \( H; e_1 \downarrow c_1 \).

Now go grab the one hypothesis we haven’t used yet and combine it with our inductive result to derive our answer:

\[
\begin{align*}
H; e_1 & \downarrow c_1 \\
H; e_2 & \downarrow c_2 \\
H; e_1 + e_2 & \downarrow c_1 + c_2
\end{align*}
\]

A nice payoff

Theorem: The small-step semantics is deterministic:
if \( H; e \rightarrow^* c_1 \) and \( H; e \rightarrow^* c_2 \), then \( c_1 = c_2 \)

Not obvious (see `sleft` and `sright`), nor do I know a direct proof

▷ Given \(((1 + 2) + (3 + 4)) + (5 + 6)) + (7 + 8)\) there are many execution sequences, which all produce 36 but with different intermediate expressions

Proof:

▷ Large-step evaluation is deterministic (easy induction proof)
▷ Small-step and and large-step are equivalent (just proved that)
▷ So small-step is deterministic
▷ Convince yourself a deterministic and a nondeterministic semantics cannot be equivalent
Conclusions
▶ Equivalence is a subtle concept
▶ Proofs “seem obvious” only when the definitions are right
▶ Some other language-equivalence claims:

Replace \textit{while} rule with
\[
\frac{H; e \Downarrow c \quad c \leq 0}{H; \text{while } e \ s \rightarrow H; \text{skip}} \quad \frac{H; e \Downarrow c \quad c > 0}{H; \text{while } e \ s \rightarrow H; s; \text{while } e \ s}
\]

Equivalent to our original language

Replace \textit{while} rule with
\[
\frac{H; e \Downarrow c \quad c \leq 0}{H; \text{while } e \ s \rightarrow H; \text{skip}} \quad \frac{H; e \Downarrow c \quad c > 0}{H; \text{while } e \ s \rightarrow H; s; \text{while } e \ s}
\]

Equivalent to our original language

Change syntax of heap and replace \texttt{assign} and \texttt{var} rules with
\[
\frac{H; x := e \rightarrow H, x \mapsto e; \text{skip}}{H; x \Downarrow c}
\]
Conclusions

▶ Equivalence is a subtle concept
▶ Proofs “seem obvious” only when the definitions are right
▶ Some other language-equivalence claims:

Replace \textsc{while} rule with

\[
\frac{H; e \Downarrow c \quad c \leq 0}{H; \text{while } e \ s \rightarrow H; \text{skip}} \quad \frac{H; e \Downarrow c \quad c > 0}{H; \text{while } e \ s \rightarrow H; s; \text{while } e \ s}
\]

Equivalent to our original language

Change syntax of heap and replace \textsc{assign} and \textsc{var} rules with

\[
\frac{H; x := e \rightarrow H, x \mapsto e; \text{skip}}{H; \downarrow e} \quad \frac{H; x \Downarrow c}{H; H(x) \Downarrow c}
\]

\textit{NOT} equivalent to our original language