Thoughts on Retrofitting Legacy Code for Security

Somesh Jha

University of Wisconsin

Science of Security, ITI (April 2, 2015)

Kaminsky scores 23, No. 5 Wisconsin beats Illinois 68-49

Feb 15, 2015


GOOD NEWS/
BAD NEWS

Threat Landscape:
Summary of
Symantec Threat Report 2014

Symantec. Confidence in a connected world.

Key Findings

- 91% increase in targeted attacks campaigns in 2013
- 62% increase in the number of breaches in 2013
- Over 552M identities were exposed via breaches in 2013
- 23 zero-day vulnerabilities discovered
- 38% of mobile users have experienced mobile cybercrime in past 12 months

Key Findings (Contd.)

- Spam volume dropped to 66% of all email traffic
- 1 in 392 emails contain a phishing attacks
- Web-based attacks are up 23%
- 1 in 8 legitimate websites have a critical vulnerability

What I feel like?

News is Grim

- See talks at
  - DARPA Cyber Colloqium
  - http://www.darpa.mil/Cyber_Colloqium_Presentations.aspx

- What do we do?

Clean-slate Design

- Rethink the entire system stack

- Networks
  - NSF program
    - See http://cleanslate.stanford.edu
  - See DARPA Mission Resilient Clouds (MRC) program
- Hosts
  - DARPA CRASH program

Some Interesting Systems

- Operating systems with powerful capabilities
  - Asbestos, HiStar, Flume
  - Capsicum
  - ….
- Virtual-machine based
  - Proxos
  - Overshadow
- Possible to build applications with strong guarantees
  - *Web server*: No information flow between threads handling different requests

Two Guiding Principles

- Provide powerful primitives at lower levels in the “system stack”
  - *Example:* HiStar (information flow labels at the OS level)

- Systems will be compromised, but limit the damage
  - *Example:* Process can be compromised, but sensitive data cannot be exfiltrated

What happens to all the code?

- Should we implement all the code from scratch?
- Can we help programmers adapt their code for these new platforms?

- Analogy
  - We have strong foundation
  - Can we build a strong house on top of it?

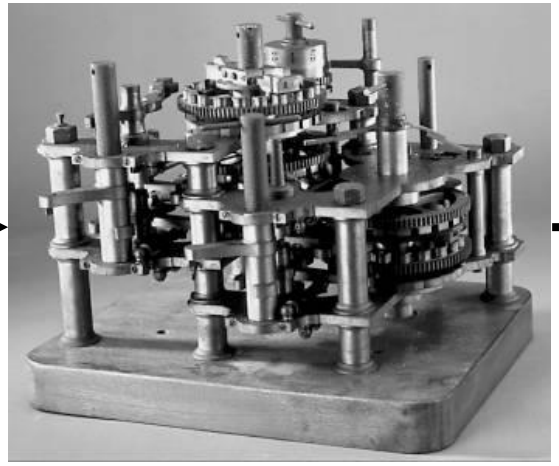
Ideal Functionality

- Input: functionality/security policy
  - Output: functional/secure code

- Proving safety is “undecidable”
  - Rice’s theorem (proving any non-trivial property is undecidable)
- I think
  - Synthesis is “relatively hard”
    - Even if provided with an oracle to prove safety

Retrofitting legacy code

Need systematic techniques to retrofit legacy code for security

Legacy code →  → Retrofitted code

INSECURE

SECURE

Premise

- Techniques and ideas from
  - Verification
  - Static Analysis
  - …
- Can help with this problem

Collaborators and Funding



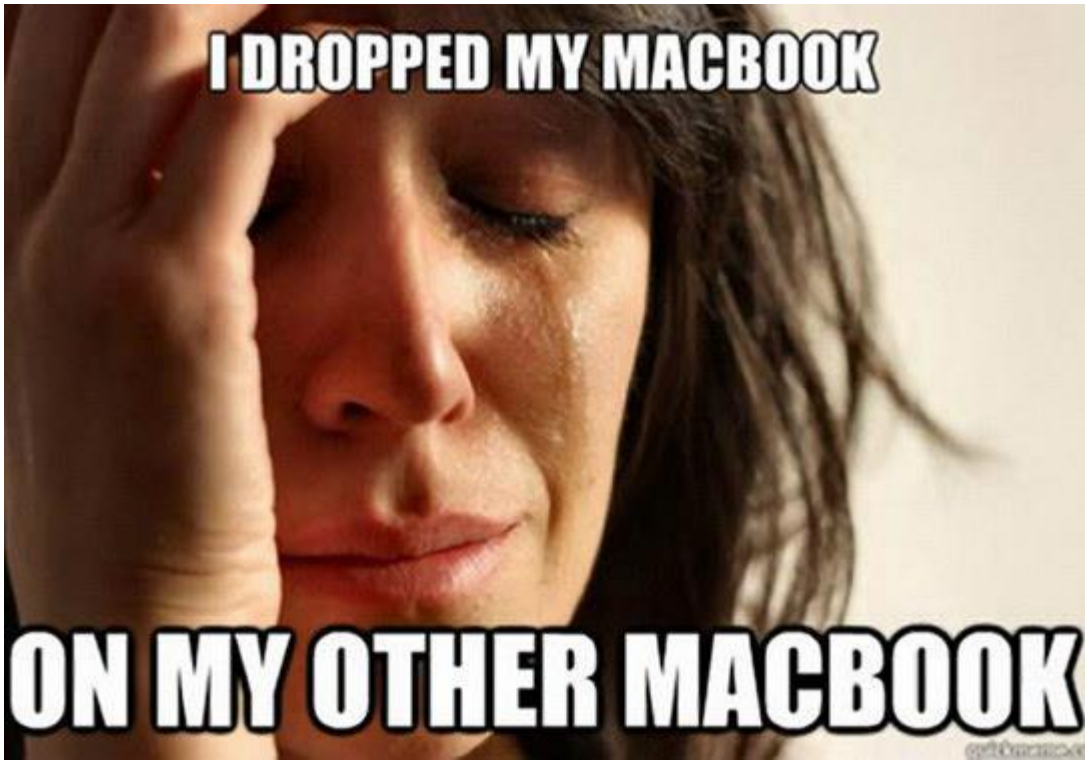




National Science Foundation
WHERE DISCOVERIES BEGIN

DARPA

The Problem



Rewriting Programs for a Capability System

[Harris et. al., Oakland 2013]

- *Basic problem:* take an insecure program and a policy, instrument program to invoke OS primitives to satisfy the policy

- *Key technique:* reduce to safety game between program and instrumentation

Capsicum

What is Capsicum?

- Capsicum is a lightweight OS capability and sandbox framework developed at the University of Cambridge Computer Laboratory
  - supported by grants from Google, the FreeBSD Foundation, and DARPA.
  - Capsicum extends the POSIX API, providing several new OS primitives to support object-capability security on UNIX-like operating systems:

Capsicum

- https://www.cl.cam.ac.uk/research/security/capsicum/
- The FreeBSD implementation of Capsicum, developed by Robert Watson and Jonathan Anderson, ships out of the box in FreeBSD 10.0 (and as an optionally compiled feature in FreeBSD 9.0, 9.1, and 9.2)
- Also available on Linux

Running example: gzip

```
compr(in, out) {
  body;
}
```

```
gzip() {
  files = parse_cl;
  for (f in files)
    (in, out) = open;
    compr(in, out);
}
```

public_leak.com

An Informal Policy for gzip

When gzip executes body,
it should only be able to read from in
and write to out.

Capsicum: An OS with Capabilities

- Two levels of capability:
  - High Capability (can open files)
  - Low Capability (cannot open files)
- Rules describing capability:
  1. Process initially executes with capability of its parent
  2. Process can invoke the drop system call to take Low Capability

Securing gzip on Capsicum

```
compr(in, out) {
  drop();
  body;
}
```

Low Cap.

public_leak.com

```
gzip() {
  files = parse_cl;
  for (f in files)
    (in, out) = open;
    compr(in, out);
}
```

High Cap.

Securing gzip on Capsicum

```
compr(in, out) {
  drop();
  body;        Low Cap.
}
```

```
gzip() {
  files = parse_cl;   High Cap.
  for (f in files)        High Cap.
    (in, out) = open;   High Cap.
    compr(in, out);   High Cap.
}
```

Securing gzip on Capsicum

```
compr(in, out) {
  drop();
  body;
}
```

```
gzip() {
  files = parse_cl;
  for (f in files)
    (in, out) = open;
    compr(in, out);
}
```

Low Cap.

Low Cap. ≠ **High Cap.**

Securing gzip on Capsicum

```
compr(in, out) {
  drop();
  body;  Low Cap.
}

gzip() {
  files = parse_cl;
  for (f in files)                High Cap.
    (in, out) = open;             High Cap.
    fork_proc(compr, in, out);    High Cap.
    wait_proc(in);
}
```

Securing gzip on Capsicum

```
compr(in, out) {
  drop();
  body;
}
```

Low Cap.

```
gzip() {
  files = parse_cl;
  for (f in files)
    (in, out) = open;
    fork_compr(in, out);
    close(in, out);
}
```

High Cap.

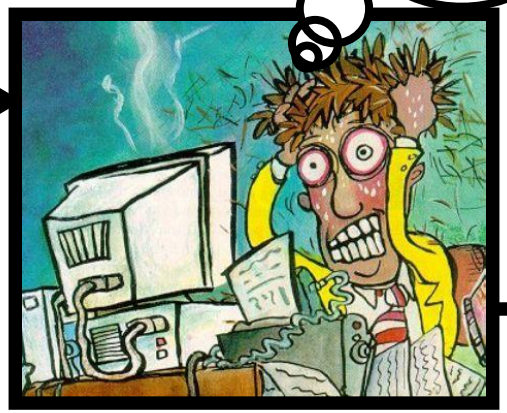
State of the Art in Rewriting

gzip should always execute comp() with low cap, but always open files in main with high cap

Insecure Program

```
gzip() {
  …
  compr();
  …
}

compr(…) { … }
```



Secure Program

```
gzip() {
  …
  fork_compr();
  …
}

compr(…) {
  drop();
  …
}
```

Insights

First Key Insight

Policies are not *instrumented programs*, and they should be explicit.

First Key Insight

gzip should always execute compr() with low cap, but always open files in main with high cap

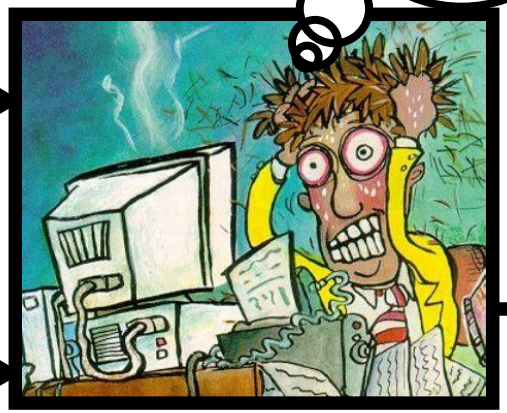
Insecure Program

```
gzip() {
  ...
  compr();
  ...
}

compr(...) { ... }
```

Disallowed Executions

.* [compr() with high cap]
| .* [open() with low cap]



Secure Program

```
gzip() {
  ...
  fork_compr();
  ...
}

compr(...) {
  drop();
  ...
}
```

Second Key Insight

From an insecure program and policy,
we can automatically write a *secure* program
by a solving a two-player safety game.

[Harris et. al., CAV 2012]

Second Key Insight

<u>Insecure Program</u>

```
gzip() {
  …
  compr();
  …
}

compr(…) { … }
```

<u>Disallowed Executions</u>

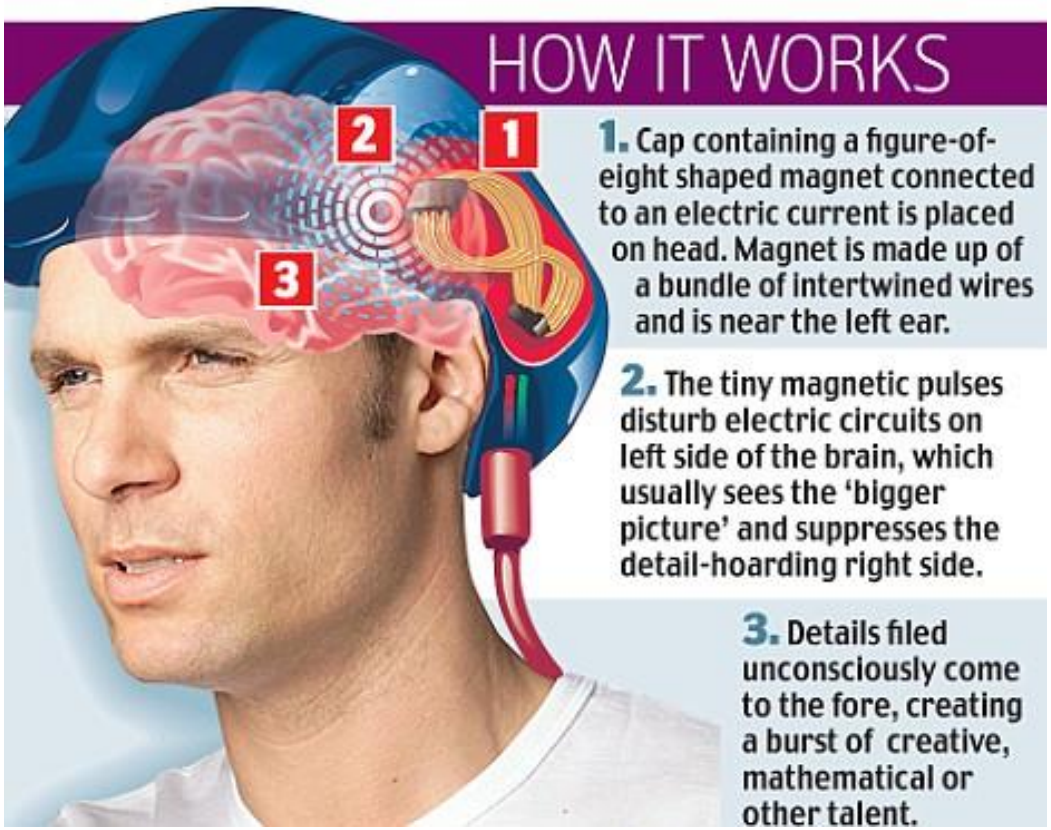
.* [compr() with high cap]
| .* [open() with low cap]

<u>Secure Program</u>

```
gzip() {
  …
  fork_compr();
  …
}

compr(…) {
  drop();
  …
}
```

The Technique



Weaving as a Game

Two steps:

1. Model uninstrumented program, policy, and Capsicum as languages/automata

2. From automata, translate weaving problem to a two-player safety game

Step 1: Model

- Program is a language over program instructions (Instrs)
- Policy is a language of instructions paired with capability (Instrs x Caps)
- Capsicum is a *transducer* from instructions and primitives to capabilities (Instrs U Prims → Caps)

Step 2: Construct a Game

- From models, construct a “game” between insecure program and instrumentation
- Program plays instructions (Instrs), instrumentation plays primitives (Prims)
- Program wins if it generates an execution that violates the policy

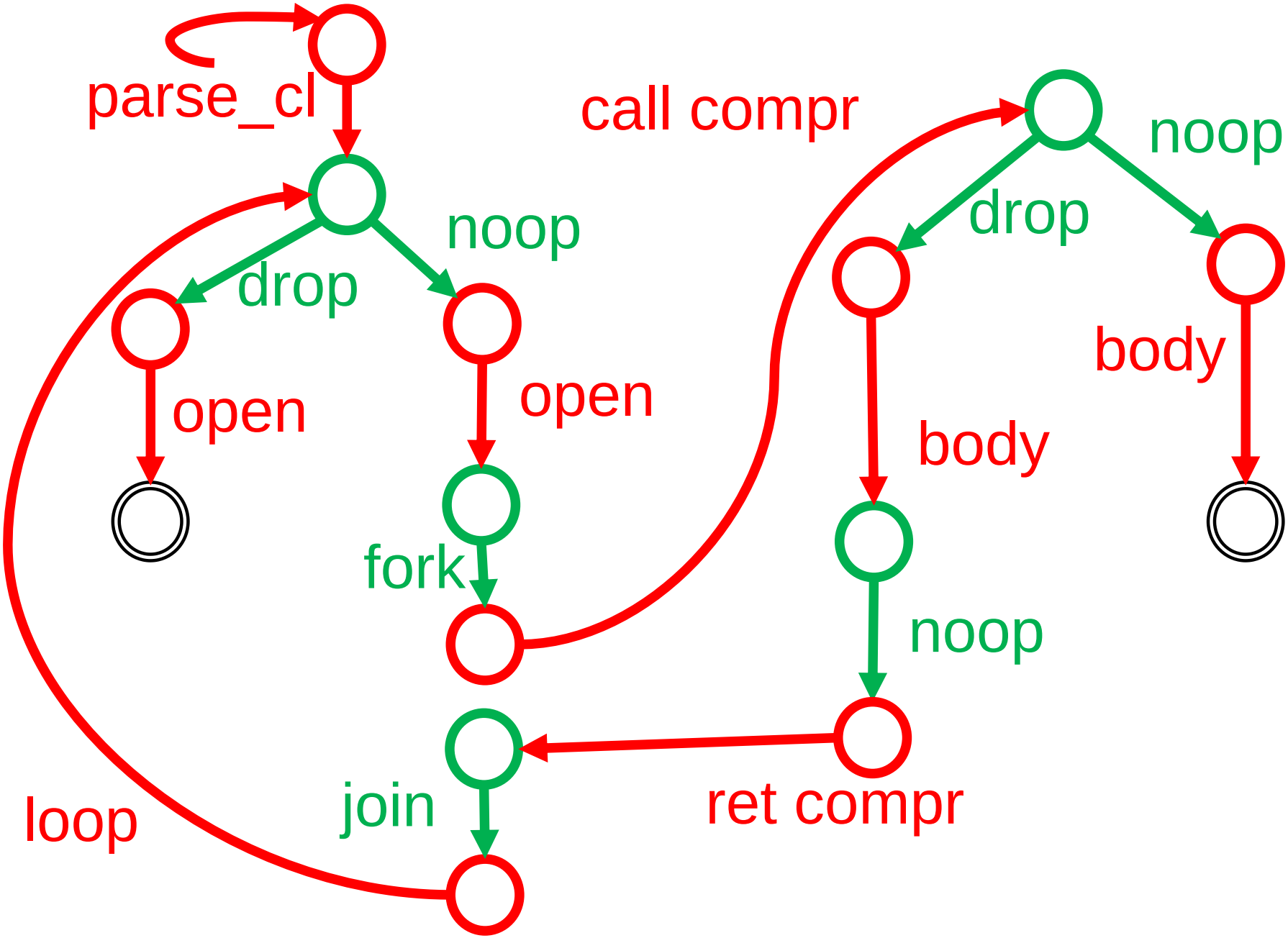
Safety Games: A Primer

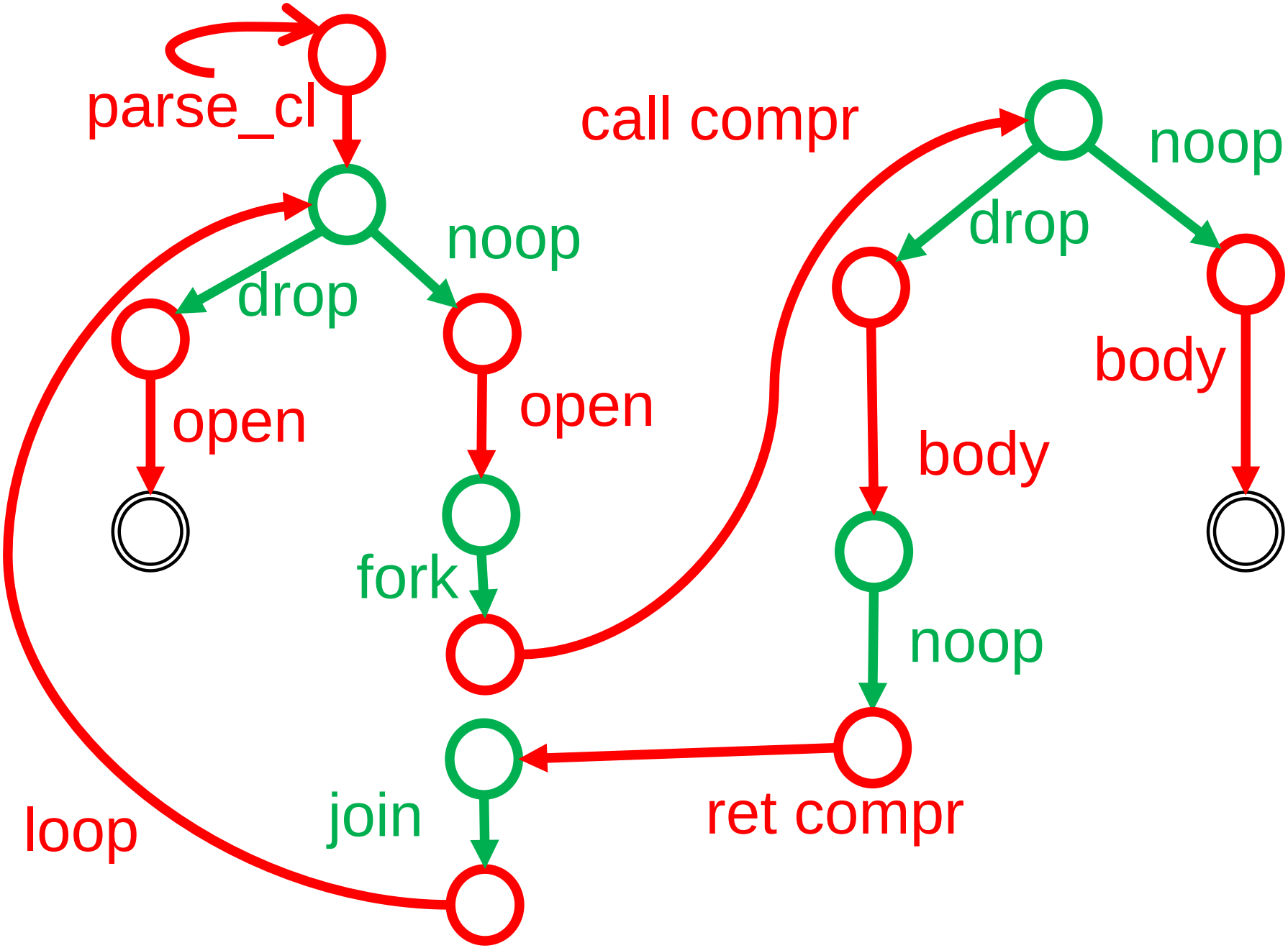
Two players: Attacker and Defender

Play: Attacker and Defender choose actions in alternation

Player goals:

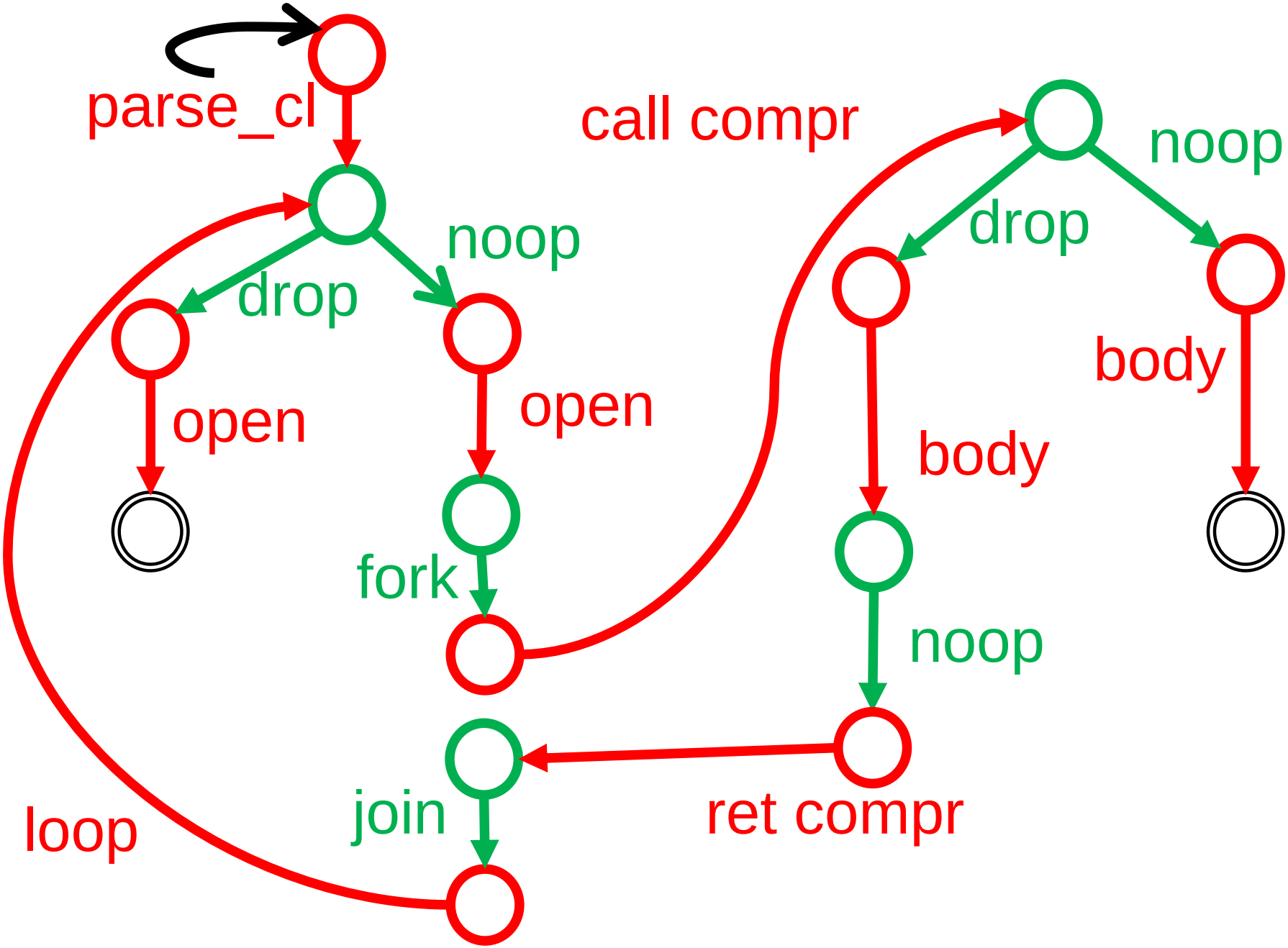
- Attacker: generate a play accepted by the game
- Defender: thwart the Attacker

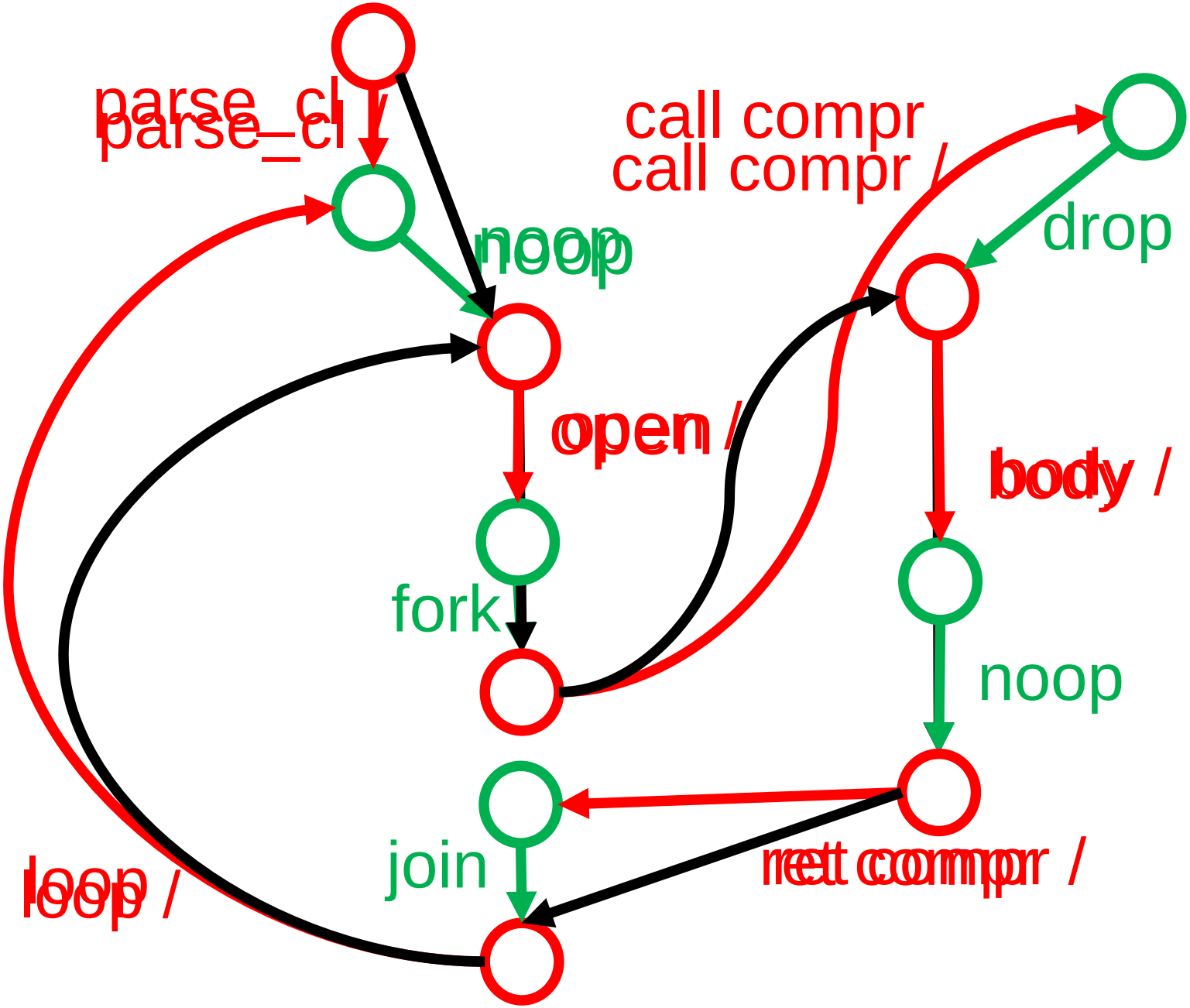

parse_cl
call compr
noop
drop
noop
drop
open
open
body
body
fork
noop
loop
join
ret compr


parse_cl
call compr
noop
drop
noop
drop
open
open
body
body
fork
noop
loop
join
ret compr

Winning Strategy

Winning strategy: choices that a player can make to always win a game


parse_cl
call compr
noop
drop
noop
drop
open
open
body
body
fork
noop
join
ret compr
loop


parse_cl /
noop
call compr /
drop
open /
body /
fork
noop
join
loop /
ret compr /

Some Details

Experimental Highlights

- capweave policies are small compared to program size (10's of lines vs. thousands)
- capweave instruments most programs fast enough to be in an edit-compile-run cycle
- capweave-rewritten programs have reasonable overhead vs. hand-rewritten

capweave Performance

Name	kLoC	Pol. Lines	Time
bzip2	8	70	4m57s
gzip	9	68	3m26s
php-cgi	852	114	46m36s
tar	108	49	0m08s
tcpdump	87	52	0m09s
wget	64	35	0m10s

Weaved-program Performance

Name	Tests	Passed	Overhead: capweave / hand
bzip2	6	6	20.90%
gzip	2	2	15.03%
php-cgi	11	2	65.64%
tar	1	1	64.78%
tcpdump	29	27	24.77%
wget	4	4	0.91%

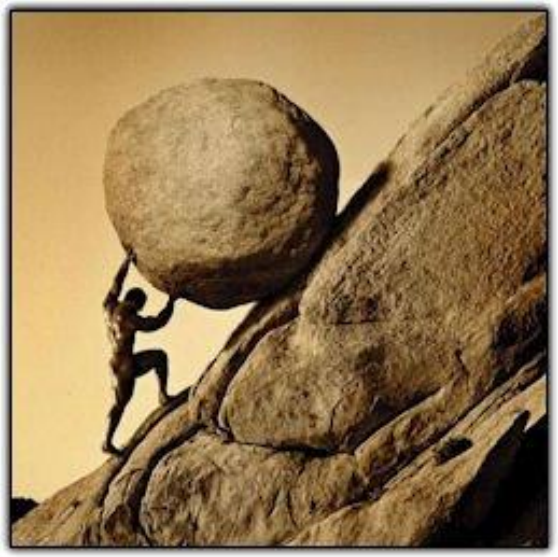
Additional Challenges

- User Study
  - Patterson: “How do you know you are doing better?”

- Optimizations
  - Incorporate quantitative measures into games (e.g., mean-payoff games)
- User-friendliness
  - Better policy language

The Future

OK… but when does it end?



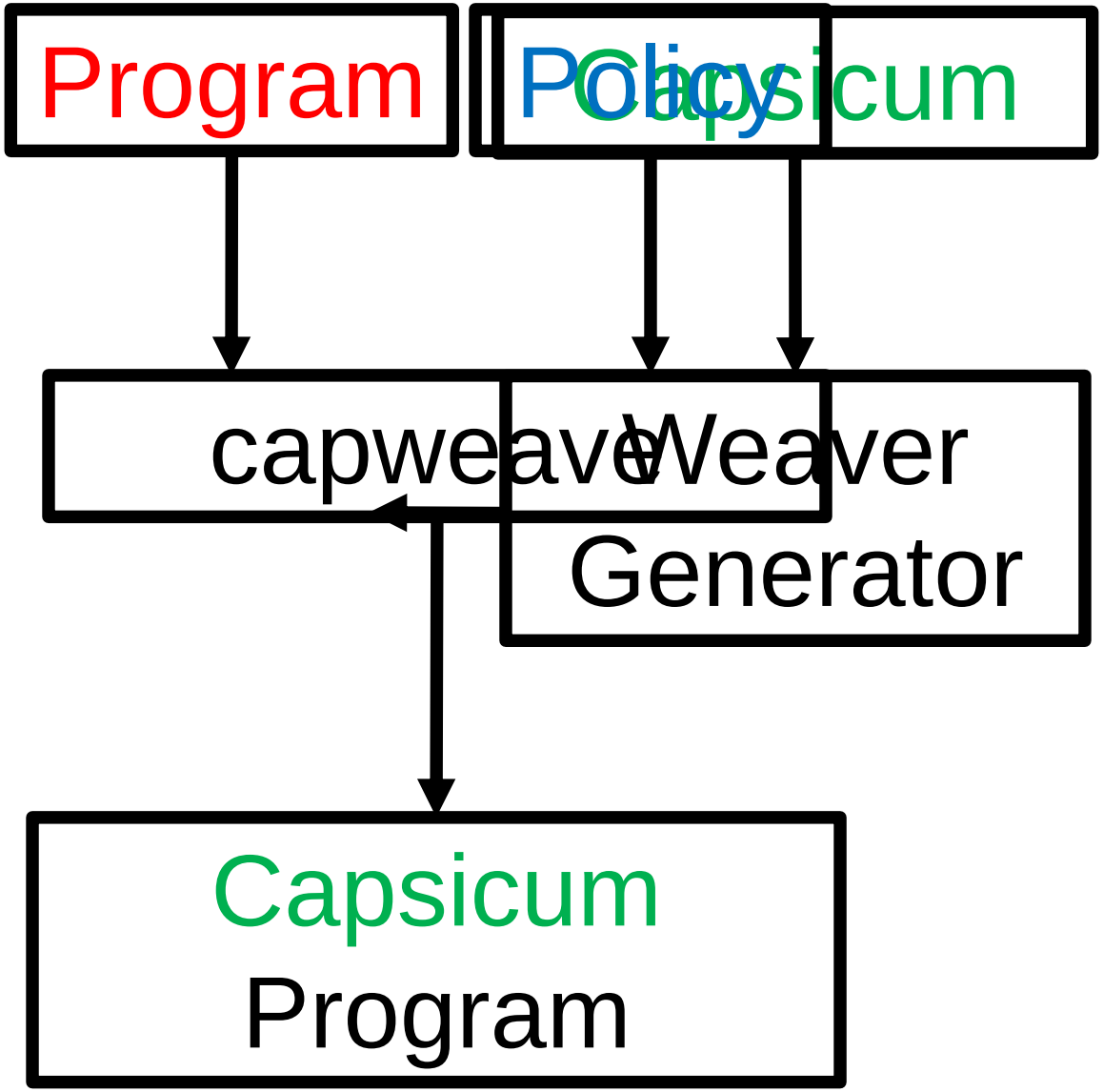
Decentrailized Information Flow

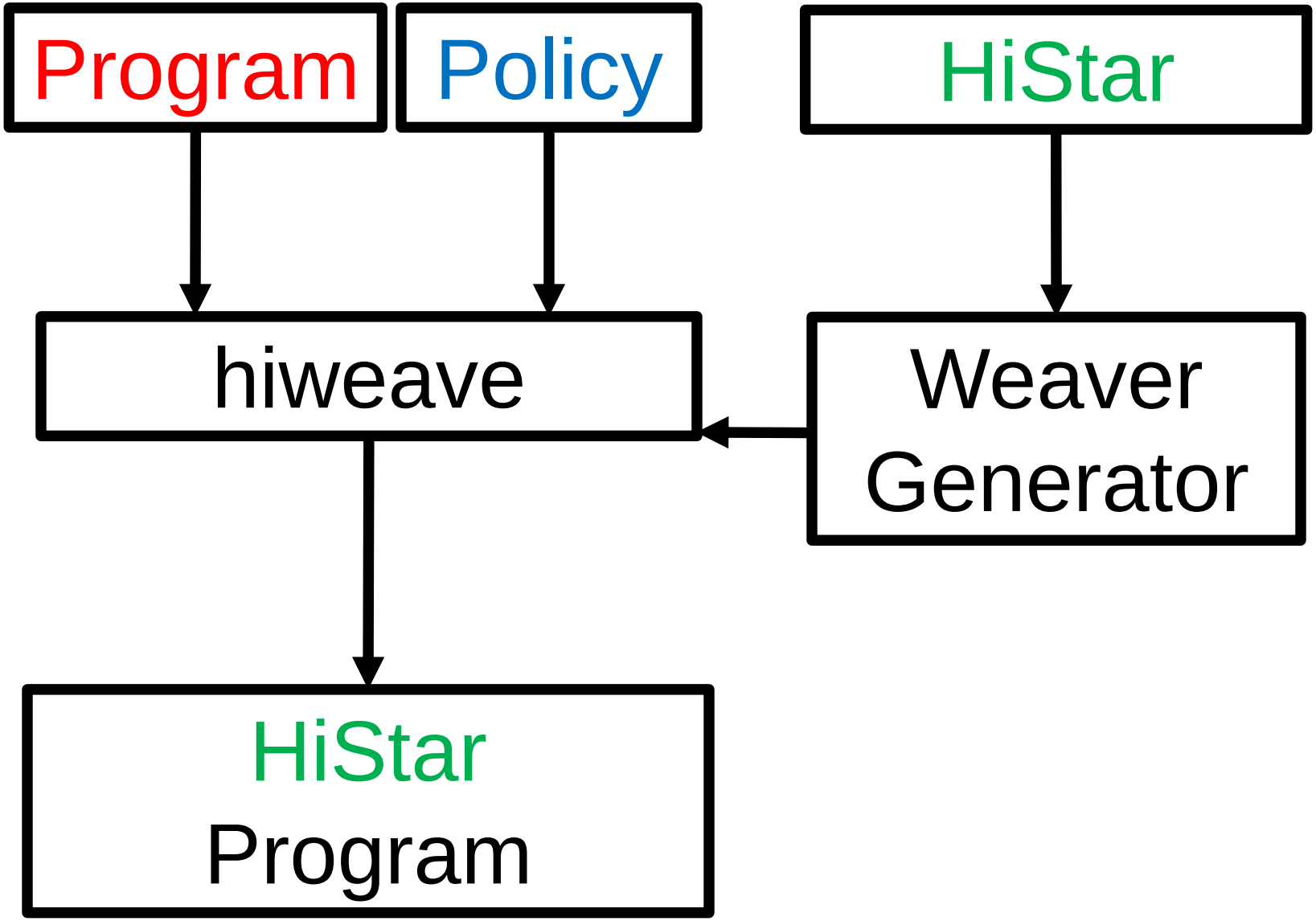
- Asbestos [SOSP 2005]
- HiStar [SOSP 2006]
- Flume [SOSP 2007]*

Analogous problem to capabilities

- Capabilities ≈ flows
- drop ≈ labels

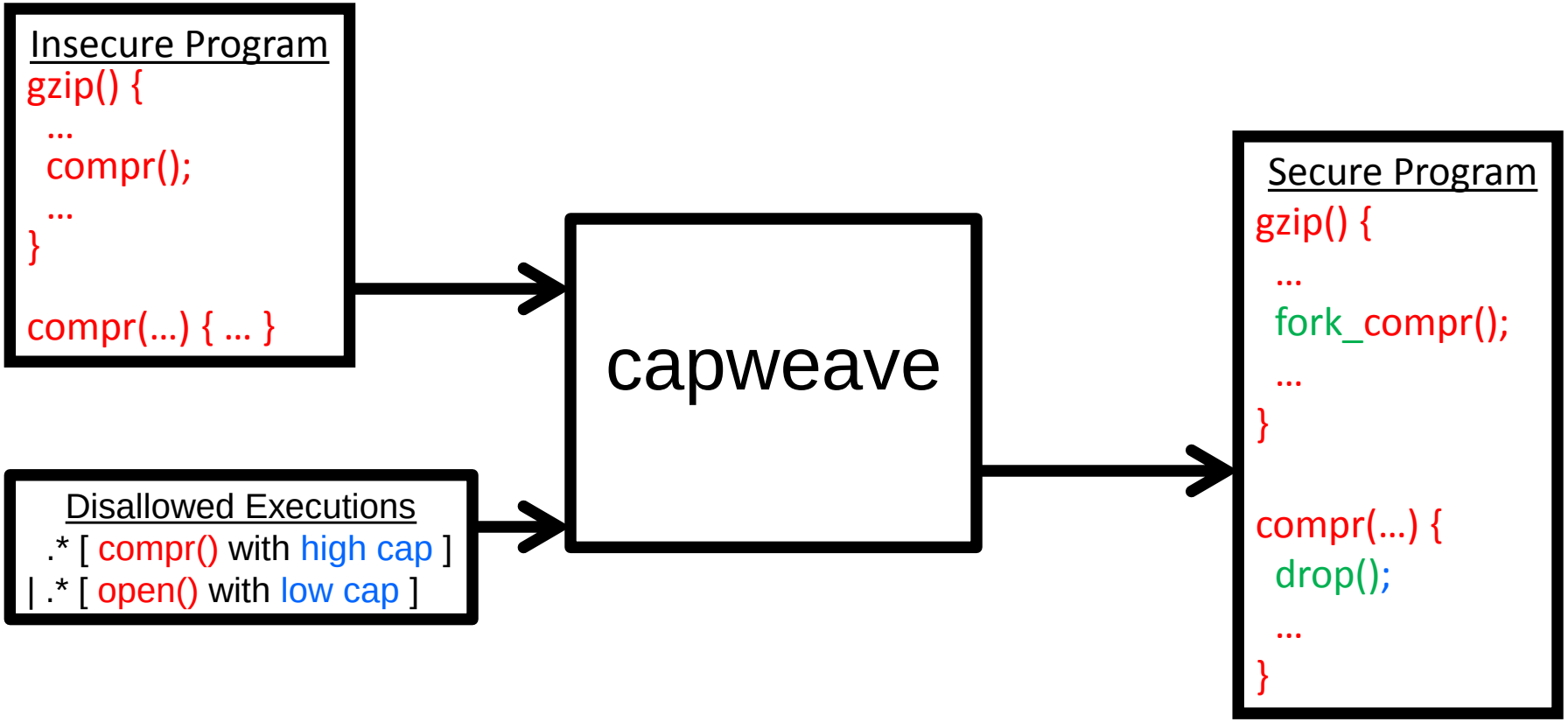
* Related work in [Harris et. al., CCS 2010]


Program
Policy
Capsicum
capweaver
Weaver
Generator
Capsicum
Program


Program
Policy
HiStar
hiweave
Weaver Generator
HiStar Program

Questions

Summary


Insecure Program
gzip() {
...
compr();
...
}
compr(...) { ... }
Disallowed Executions
.* [compr() with high cap]
| .* [open() with low cap]
capweave
Secure Program
gzip() {
...
fork_compr();
...
}
compr(...) {
drop();
...
}