Secure Programs via Game-based Synthesis

Somesh Jha, Tom Reps, and Bill Harris
One-slide summary

- Secure programming on a conventional OS is intractable
- Privilege-aware OS’s take secure programming from intractable to challenging
- Our program rewriter takes secure programming from challenging to simple
Outline

1. Motivation, problem statement
2. Previous work: Capsicum [CAV ’12, Oakland ’13]
3. Ongoing work: HiStar
4. Open challenges
Outline

1. Motivation, problem statement
Secure Programming is *Intractable*

- 81 exploits in CVE since Sept. 2013
- Many exploit a software bug to carry out undesirable system operations
  - 2013-5751: exploit SAP NetWeaver to traverse a directory
  - 2013-5979: exploit bad filename handling in Xibo to read arbitrary files
  - 2013-5725: exploit ByWord to overwrite files
How to Carry Out an Exploit

software vulnerability + OS privilege = security exploit
The Conventional-OS Solution

software vulnerability
+ OS privilege
= security exploit
software vulnerability + OS privilege = security exploit
The Conventional-OS Solution

software vulnerability
+

security exploit
The Program-Verification Solution

software vulnerability + OS privilege = security exploit
The Program-Verification Solution

software vulnerability + OS privilege = security exploit
The Program-Verification Solution

software vulnerability + OS privilege ≠ security exploit
Priv.-aware OS

- Introduce **explicit privileges** over all system objects, **primitives** that update **privileges**
- Programs call **primitives** to manage **privilege**
The **Priv.**-aware OS Solution

software vulnerability + OS privilege = security exploit
The Priv.-aware OS Solution

\[ \text{software vulnerability} + \text{primitives} + \text{OS privilege monitor} = \text{security exploit} \]
The Priv.-aware OS Solution

(software vulnerability + primitives + OS privilege monitor ≠ security exploit)
The Capsicum Priv.-aware OS [Watson ’10]

- Privilege: ambient authority (Amb) to open descriptors to system objects
- Primitives: program calls `cap_enter()` to manage Amb
Rules of Capsicum’s Amb
Rules of Capsicum’s Amb

1. When a process is created, it has the Amb value of its parent.
Rules of Capsicum’s $\texttt{Amb}$

1. When a process is created, it has the $\texttt{Amb}$ value of its parent

2. After a process calls $\texttt{cap\_enter()}$, it does not have $\texttt{Amb}$
Rules of Capsicum’s Amb

1. When a process is created, it has the Amb value of its parent.

2. After a process calls cap_enter(), it does not have Amb.

3. If a process does not have Amb, then it can never obtain Amb.
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        L1: compress(in, out);
}
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        L1: compress(in, out);
}
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        L1: compress(in, out);
}
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        L1: compress(in, out);
    }

gzip

http://evil.com

/usr/local
A simple gzip policy

• When `gzip` calls `open2()` at L0, it should be able to open descriptors

• When `gzip` calls `compress()` at L1, it should not be able to open descriptors
A simple gzip policy with AMB

- When gzip calls open2() at L0, it should have AMB

- When gzip calls compress() at L1, it should not have AMB
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        L1: compress(in, out);
}
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        L1: compress(in, out);
    }

gzip with AMB
**gzip with AMB**

```python
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        L1: compress(in, out);
}
```

L0: AMB
L1: no AMB

Tuesday, October 22, 13
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        L1: compress(in, out);
        cap_enter()
}
gzip with AMB

```c
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        L1: compress(in, out);
}
```

L0: AMB
L1: no AMB
1. Amb policies are not explicit

2. cap_enter primitive has subtle temporal effects
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        in, out) = open2(f);
        L0: (in, out) = open2(f);

        L1: compress(in, out);
    }

L0: AMB
L1: no AMB
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        cap_enter();
        L1: compress(in, out);
}
Programming Challenges

```c
main() {
    file_nms = parse_cl(); AMB
    for (f in file_nms):
        (in, out) = open2(f);
        cap_enter();
        L1: compress(in, out);
}
```

L0: AMB
L1: no AMB
main() {
    file_nms = parse_cl();
    for (f in file_nms): AMB
        (in, out) = open2(f);
        cap_enter();
    L1: compress(in, out);
}

L0: AMB
L1: no AMB
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f); AMB
        cap_enter();
        L1: compress(in, out);
}
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        (in, out) = open2(f);
        cap_enter();
        L1: compress(in, out);
    L0: (in, out) = open2(f);
}

L0: AMB
L1: no AMB


```c
main() {
    file_nms = parse_cl();
    for (f in file_nms):  
        L0: (in, out) = open2(f);
        cap_enter();
        L1: compress(in, out);
}
```

`L0: AMB`

`L1: no AMB`
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f); no AMB
        cap_enter();
        L1: compress(in, out);
}

L0: AMB
L1: no AMB
Rules of Capsicum’s Amb

1. When a process is created, it has the AMB value of its parent
2. After a process calls cap_enter(), it never has AMB
3. If a process does not have Amb, then it can never obtain Amb
Rules of Capsicum’s Amb

1. When a process is created, it has the AMB value of its parent
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        cap_enter();
        L1: compress(in, out);
    }

L0: AMB
L1: no AMB
main() {
    file_nms = parse_cl();
    for (f in file_nms):
      L0: (in, out) = open2(f);
      sync_fork();
      cap_enter();
      L1: compress(in, out);
      sync_join();
  }

L0: AMB
L1: no AMB
main() {
    file_nms = parse_cl(); AMB
    for (f in file_nms):
        L0: (in, out) = open2(f);
            sync_fork();
            cap_enter();
        L1: compress(in, out);
            sync_join();
}
main() {
    file_nms = parse_cl();
    for (f in file_nms):  AMB
L0:  (in, out) = open2(f);
    sync_fork();  L1:  compress(in, out);
    cap_enter();
L1:  compress(in, out);
    sync_join();
}

L0: AMB
L1: no AMB
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f); \textbf{AMB}
            sync_fork();
            cap_enter();
        L1: compress(in, out);
            sync_join();
}

L0: \textbf{AMB}
L1: \textbf{no AMB}
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        sync_fork();
        cap_enter();
        L1: compress(in, out);  no AMB
        sync_join();
}
Instrumenting gzip

```c
main() {
  file_nms = parse_cl();
  for (f in file_nms):  AMB
    L0: (in, out) = open2(f);
    sync_fork();  AMB
    cap_enter();  AMB
    L1: compress(in, out);
    sync_join();  AMB
}
```

L0: AMB
L1: no AMB
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f); AMB
        sync_fork();
        cap_enter();
        L1: compress(in, out);
        sync_join();
}
Capsicum Challenges

Not Appearing in This Talk

- Program can construct **capability** from each UNIX descriptor
- Capability has a vector of 63 **access rights** (~1 for every system call on a descriptor)
- Programs can assume new capabilities via a Remote Procedure Call (RPC)
Instrumenting Programs with CapWeave

1. Programmer writes an explicit Amb policy

2. CapWeave instruments program to invoke primitives so that it satisfies the policy
gzip with CapWeave

```c
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        L1: compress(in, out);
}
```

L0: AMB
L1: no AMB
gzip with CapWeave

```c
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        L1: compress(in, out);
}
```

Policy

\[
\text{Cur}(p) \Rightarrow (\text{pc}[L0](p) \Rightarrow \text{AMB}(p) \\
& (\text{pc}[L1](p) \Rightarrow !\text{AMB}(p)))
\]
main() {
    file_nms = parse_cl();
    for (f in file_nms):
L0: (in, out) = open2(f);
L1: compress(in, out);
}

Policy
Cur(p) => (pc[L0](p) => AMB(p))
& (pc[L1](p) => !AMB(p))
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        L1: compress(in, out);
}
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        L1: compress(in, out);
    }

Policy
Cur(p) => (pc[L0](p) => AMB(p) & (pc[L1](p) => !AMB(p)))

CapWeave
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        L1: compress(in, out);
}
The Next 700 Policy Weavers

Analogous challenges with **Decentralized Information Flow Control (DIFC)**

- Asbestos [Efstathopoulos ‘05]
- HiStar [Zeldovich ’06]
- Flume [Krohn ‘07]
gzip() {
    file_nms = parse_cl();
    ...
}

Policy
Cur(p) =>
(pc[L0](p) => AMB(p))
& (pc[L1](p) => !AMB(p))

CapWeave

gzip() {
    file_nms = parse_cl();
    sync_fork();
    cap_enter();
    ...
}
gzip() {
    file_nms = parse_cl();
    ...  
}

Policy
Cur(p) =>
(pc[L0](p) => AMB(p))
& (pc[L1](p) => !AMB(p))
gzip() {
    file_nms = parse_cl();
    ...  
}

gzip() {
    file_nms = parse_cl();
    sync_fork();  
cap_enter();
    ...  
}
HiStar Designer

create_cat(&c):
Flows'(p, q) := Flows(p, q) || ...

Weaver
Generator
HiWeave

Weaver Generator

create_cat(&c):
Flows’(p, q) := Flows(p, q) || ...

HiStar Designer
wrapper() {
    exec(...);
    ...
}

Policy
forall w, s.
    Flows(w, s) => ...

create_cat(&c):
    Flows'(p, q) := Flows(p, q) || ...

HiWeave

Weaver Generator

25 Tuesday, October 22, 13
HiWeave

Programmer

wrapper() {
    exec(...);
    ...
}

HiStar Designer

create_cat(&c):
    Flows'(p, q) := Flows(p, q) || ...

Weaver Generator

scanner() {
    create_cat(&c);
    exec(...);
    ...
}
Outline

1. Motivation, problem statement
2. Previous work: Capsicum
3. Ongoing work: HiStar
4. Open challenges
2. Previous work: Capsicum
CapWeave Algorithm
CapWeave Algorithm

Inputs: Program $P$, Amb Policy $Q$
CapWeave Algorithm

Inputs: Program $P$, Amb Policy $Q$
Output: Instrumentation of $P$ that always satisfies $Q$
CapWeave Algorithm

Inputs: Program $P$, Amb Policy $Q$
Output: Instrumentation of $P$ that always satisfies $Q$

1. Build finite $\text{IP#} \supseteq$ instrumented runs that violate $Q$
I. Building IP#: Inputs

Program

```c
main() {
    file_nms = parse_cl();
    for (f in file_nms):
        L0: (in, out) = open2(f);
        L1: compress(in, out);
}
```

Amb Policy

L0: Amb
L1: no Amb
1. Building IP#: Output
1. Building IP#: Output

L1:compress()}

noop

cap_enter()
1. Building IP#: Output

parse_cl

cap_enter

noop

L0:open2()

L0:open2()

noop

noop

cap_enter()

L1:compress()

noop

sync_fork()

noop

sync_join()

noop

noop

cap_enter()

L0:open2()

L1:compress()

noop

L1:compress()

L1:compress()

L1:compress()
Building IP#

Basic idea: construct IP# as a forward exploration of an abstract state space
I (a). IP#: Define
Abstract State-space
I (a). IP#: Define Abstract State-space
1(a). IP\#: Define Abstract State-space
I (a). IP#: Define Abstract State-space

\[ Q \]
\[ Q' \]
\[ Q' \]
\[ Q' \]
I (b). IP#: Define Abstract Transformers
I (b). IP#: Define Abstract Transformers
I (b). IP#: Define Abstract Transformers

\( Q \# \quad \alpha \quad Q \)

\[ \tau[\text{cap\_enter}]\# \quad \tau[\text{cap\_enter}] \]

Tuesday, October 22, 13
I (c). Explore Abstract State Space
I (c). Explore Abstract State Space

\[ Q# \]

\[ \alpha \]

\[ Q \]

init

L0
I (c). Explore Abstract State Space

\[ \begin{align*}
Q \# & \quad \text{\textcopyright{\textcolor{green}{cap\_enter}}\#} \\
\alpha & \quad \text{\textcopyright{\textcolor{green}{noop}}} \\
\text{init} & \quad \text{\textcopyright{\textcolor{green}{cap\_enter}}} \\
L_0 & \quad \text{\textcopyright{\textcolor{green}{noop}}} \\
L_0' & \quad \text{\textcopyright{\textcolor{green}{noop}}} \\
\end{align*} \]
State-Structure Exploration

If a concrete state is a logical structure, ...
State-Structure Exploration

If a concrete state is a logical structure, ...

$Q \equiv \{ \text{Diagram} \} \equiv \{ \text{State} \}$
State-Structure Exploration

properties are FOL formulas, ...

∀p. A(p) ⇒ ((B(p) ⇒ C(p)) ∧ (D(p) ⇒ ¬C(p))))
State-Structure Exploration

...and semantics is given as predicate updates, ...

\[ \tau[\text{action}] \equiv \]

\[ A'(x) = A(x) \lor \exists y. C(y) \land B(q, p) \]

\[ B'(x, y) = B(x, y) \lor (C(x) \land D(y)) \]

\[ C'(x) = ... \]

\[ D'(x) = ... \]
State-Structure Exploration

...then abstract space and transformers can be generated automatically [Sagiv ’99]
State-Structure Exploration

...then abstract space and transformers can be generated automatically [Sagiv ’99]
Capsicum Semantics

1. \( Q \equiv \)

2. \( \tau[\text{action}] \equiv \)

\[ \begin{align*}
A'(x) &= A(x) \lor \exists y. C(y) \land B(q, p) \\
B'(x, y) &= B(x, y) \lor (C(x) \land D(y)) \\
C'(x) &= \ldots \\
D'(x) &= \ldots
\end{align*} \]
Capsicum State as Structure
Capsicum State as Structure
Capsicum State as Structure

\[ \text{Cur} \]

\[ O \quad O \quad O \]
Capsicum State as Structure
Capsicum State as Structure

Amb \hspace{2cm} \bigcirc \hspace{2cm} \bigcirc \hspace{2cm} \text{Cur}
L1
Amb
Capsicum State as Structure

Amb → Parent → Cur

L1
Amb
∀ p. Cur(p) ∧ LI(p) ⇒ ¬ Amb(p)
Capsicum State as Structure

∀ p. \textbf{Cur}(p) \land \textbf{LI}(p) \Rightarrow \lnot \textbf{Amb}(p)
∀ p. Cur(p) ∧ LI(p) ⇒ ¬ Amb(p)
Capsicum State as Structure

∀ p. Cur(p) \land LI(p) \Rightarrow \neg Amb(p)
∀ p. \text{Cur}(p) \land \text{Ll}(p) \Rightarrow \neg \text{Amb}(p)
## Capsicum Structure Transformers

<table>
<thead>
<tr>
<th>Action</th>
<th>sync_fork()</th>
</tr>
</thead>
</table>

Tuesday, October 22, 13
### Capsicum Structure Transformers

<table>
<thead>
<tr>
<th>Action</th>
<th>sync_fork()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cur</td>
<td></td>
</tr>
<tr>
<td>Amb</td>
<td>O</td>
</tr>
</tbody>
</table>

Tuesday, October 22, 13
Capsicum Structure Transformers

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync_fork()</td>
</tr>
</tbody>
</table>
# Capsicum Structure Transformers

<table>
<thead>
<tr>
<th>Action</th>
<th>sync_fork()</th>
</tr>
</thead>
</table>

*Figure: Diagram illustrating the structure of Capsicum Structure Transformers with nodes labeled Cur Amb and Fresh Amb.*
Capsicum Structure Transformers

Amb → Parent → Fresh → Cur Amb

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync_fork()</td>
</tr>
</tbody>
</table>
Capsicum Structure Transformers

Action | Structure Transformer
--- | ---
 sync_fork() |  

Tuesday, October 22, 13
Capsicum Structure Transformers

\[ \text{Amb}'(p) := \text{Amb}(p) \lor (\text{Fresh}(p) \land \exists q. \text{Cur}(q) \land \text{Amb}(q)) \]

<table>
<thead>
<tr>
<th>Action</th>
<th>Structure Transformer</th>
</tr>
</thead>
</table>
| sync_fork()  | Intro Fresh

\[ \text{Amb}'(p) := \text{Amb}(p) \lor (\text{Fresh}(p) \land \exists q. \text{Cur}(q) \land \text{Amb}(q)) \]
Capsicum Structure Transformers

<table>
<thead>
<tr>
<th>Action</th>
<th>Structure Transformer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Capsicum Structure Transformers

Action | Structure Transformer
---|---
cap_enter() |
# Capsicum Structure Transformers

![Diagram showing relationships between Amb, Parent, Fresh, and Cur.]

<table>
<thead>
<tr>
<th>Action</th>
<th>Structure Transformer</th>
</tr>
</thead>
<tbody>
<tr>
<td>cap_enter()</td>
<td></td>
</tr>
</tbody>
</table>
### Capsicum Structure Transformers

**Diagram:**
- **Amb** → **Parent** → **Fresh** → **Cur**

**Formal Definition:**
\[
\text{Amb}'(p) := \text{Amb}(p) \land \neg \text{Cur}(p)
\]

<table>
<thead>
<tr>
<th>Action</th>
<th>Structure Transformer</th>
</tr>
</thead>
<tbody>
<tr>
<td>cap_enter()</td>
<td>\text{Amb}'(p) := \text{Amb}(p) \land \neg \text{Cur}(p)</td>
</tr>
</tbody>
</table>
Building IP#: Summary

• If semantics is given as transforms of logical structures, we can generate an approximation of runs that cause a violation

• Capsicum semantics can be modeled as structure transforms
CapWeave Algorithm

Inputs: Program $P$, Amb Policy $Q$
Output: Instrumentation of $P$ that always satisfies $Q$

1. Build finite $IP# \supseteq$ instrumented runs that violate $Q$
CapWeave Algorithm

Inputs: Program P, Amb Policy Q
Output: Instrumentation of P that always satisfies Q

1. Build finite IP# ⊇ instrumented runs that violate Q
2. From IP#, build safety game G
   won by violations of Q
Two-Player Safety Games

- In an Attacker state, the Attacker chooses the next input
- In a Defender state, the Defender chooses the next input
- Attacker wants to reach an accepting state
## Instrumentation as a Game

<table>
<thead>
<tr>
<th>Capsicum Instrumentation</th>
<th>Two-player Games</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program instructions</td>
<td>Attacker actions</td>
</tr>
<tr>
<td>Capsicum primitives</td>
<td>Defender actions</td>
</tr>
<tr>
<td>Policy violations</td>
<td>Attacker wins</td>
</tr>
<tr>
<td>Satisfying instrumentation</td>
<td>Winning Defender strategy</td>
</tr>
</tbody>
</table>
parse_cl

L0:open2()

cap_enter

noop

L0:open2()

noop

L1:compress()

noop

cap_enter()

L1:compress()

noop

cap_enter()

noop

cap_enter()

L1:compress()

noop

cap_enter()

L1:compress()

noop

cap_enter()

L1:compress()
CapWeave Algorithm

Inputs: Program $P$, Amb Policy $Q$
Output: Instrumentation of $P$ that always satisfies $Q$

1. Build finite $IP# \supseteq$ instrumented runs that violate $Q$
2. From $IP#$, build safety game $G$
   won by violations of $Q$
CapWeave Algorithm

Inputs: Program P, Amb Policy Q
Output: Instrumentation of P that always satisfies Q

1. Build finite $\mathbf{IP#} \supseteq$ instrumented runs that violate Q
2. From IP#, build safety game $\mathbf{G}$ won by violations of Q
3. From winning strategy for G, generate primitive controller for P
## CapWeave Performance

<table>
<thead>
<tr>
<th>Name</th>
<th>Program kLoC</th>
<th>Policy LoC</th>
<th>Weaving Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2-1.0.6</td>
<td>8</td>
<td>70</td>
<td>4m57s</td>
</tr>
<tr>
<td>gzip-1.2.4</td>
<td>9</td>
<td>68</td>
<td>3m26s</td>
</tr>
<tr>
<td>php-cgi-5.3.2</td>
<td>852</td>
<td>114</td>
<td>46m36s</td>
</tr>
<tr>
<td>tar-1.25</td>
<td>108</td>
<td>49</td>
<td>0m08s</td>
</tr>
<tr>
<td>tcpdump-4.1.1</td>
<td>87</td>
<td>52</td>
<td>0m09s</td>
</tr>
<tr>
<td>wget-1.12</td>
<td>64</td>
<td>35</td>
<td>0m10s</td>
</tr>
</tbody>
</table>
## Performance on Included Tests

<table>
<thead>
<tr>
<th>Name</th>
<th>Base Time</th>
<th>Hand Overhd</th>
<th>capweave Overhd</th>
<th>Diff. Overhd (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2-1.0.6</td>
<td>0.593s</td>
<td>0.909</td>
<td>1.099</td>
<td>20.90</td>
</tr>
<tr>
<td>gzip-1.2.4</td>
<td>0.036s</td>
<td>1.111</td>
<td>1.278</td>
<td>15.03</td>
</tr>
<tr>
<td>php-cgi-5.3.2</td>
<td>0.289s</td>
<td>1.170</td>
<td>1.938</td>
<td>65.64</td>
</tr>
<tr>
<td>tar-1.25</td>
<td>0.156s</td>
<td>13.301</td>
<td>21.917</td>
<td>64.78</td>
</tr>
<tr>
<td>tcpdump-4.1.1</td>
<td>1.328s</td>
<td>0.981</td>
<td>1.224</td>
<td>24.77</td>
</tr>
<tr>
<td>wget-1.12</td>
<td>4.539s</td>
<td>1.906</td>
<td>1.106</td>
<td>0.91</td>
</tr>
</tbody>
</table>

**Tuesday, October 22, 13**
Outline

1. Motivation, problem statement
2. Previous work: Capsicum
3. Ongoing work: HiStar
4. Open challenges
3. Ongoing work: HiStar
The HiStar Priv-aware OS
[Zeldovich ’06]

- **Privilege**: OS allows flow between processes
- **Primitives**: system calls update labels, which define allowed flows
- Very powerful: mutually untrusting login (?)
Sandboxing a Virus Scanner

```c
launcher() {
    exec("/bin/scanner");
}

wrapper() {
    child = sync_fork(&launcher);
    while (true) {
        read(child, buf);
        sanitize(buf);
        write(netd, buf);
    }
}
```
A Flow Policy for a Virus Scanner

- Information should never transitively flow from the scanner to the network, unless it goes through the wrapper.
- Information should always flow from the scanner to the wrapper.
- Information should always flow from the wrapper to the network.
Rules for HiStar’s Flow

• A process’s label maps each category to low or high

• If process p calls create_cat, then each process is low in c, and p can declassify c

• Each process may raise its level at each category

• Each process may relinquish declassification
Rules for HiStar’s Flow

Information can **flow** from p to q if for each category:

- The level of p is lower than the level of q at c, or
- p can declassify c
Sandboxing a Virus Scanner

launcher() {

    exec("/bin/scanner"); }

wrapper() {

    child = sync_fork(&launcher);
    while (true) {
        read(child, buf);
        sanitize(buf);
        write(netd, buf); ] } }
Sandboxing a Virus Scanner

    launcher() {
        exec("/bin/scanner");
    }

    wrapper() {
        create_cat(&x);
        raise(x);
        child = sync_fork(&launcher);
        while (true) {
            read(child, buf);
            sanitize(buf);
            write(netd, buf);
        }
    }
Sandboxing a Virus Scanner

```c
launcher() {
    drop_declass(x);
    exec("/bin/scanner");
}
wrapper() {
    create_cat(&x);
    raise(x);
    child = sync_fork(&launcher);
    while (true) {
        read(child, buf);
        sanitize(buf);
        write(netd, buf);
    }
}
```
HiStar Challenges Not Appearing in This Talk

- There are actually **four** levels
- Each process has to manage its **clearance**
- Processes can create **labeled closures** (calling a closure implicitly performs two label operations and three ordering checks)
CapWeave Algorithm

Inputs: Program P, Amb Policy Q
Output: Instrumentation of P that satisfies Q

1. Build IP# ⊇ instrumented runs that violate Q (using Capsicum semantics)
2. From IP#, build safety game G won by violations of Q
3. From winning strategy for G, generate primitive controller for P
CapWeave Algorithm

Inputs: Program P, Amb Policy Q
Output: Instrumentation of P that satisfies Q

1. Build \( IP# \supseteq \) instrumented runs that violate Q (using semantics)
2. From IP#, build safety game G won by violations of Q
3. From winning strategy for G, generate primitive controller for P
HiWeave Algorithm

Inputs: Program P, Flow Policy Q
Output: Instrumentation of P that satisfies Q

1. Build IP# ⊇ instrumented runs that violate Q (using HiStar semantics)
2. From IP#, build safety game G won by violations of Q
3. From winning strategy for G, generate primitive controller for P
Capsicum Semantics

1. \[ Q \equiv \]

2. \[ \tau[action] \equiv \]
   \[
   \begin{align*}
   A'(x) &= A(x) \lor \exists y. C(y) \land B(q, p) \\
   B'(x, y) &= B(x, y) \lor (C(x) \land D(y)) \\
   C'(x) &= \ldots \\
   D'(x) &= \ldots
   \end{align*}
   \]
HiStar Semantics

1. \( Q \equiv \)

2. \( \tau[\text{action}] \equiv \)

\[
\begin{align*}
A'(x) &= A(x) \lor \exists y. C(y) \land B(q, p) \\
B'(x, y) &= B(x, y) \lor (C(x) \land D(y)) \\
C'(x) &= \ldots \\
D'(x) &= \ldots
\end{align*}
\]
HiStar State as Structure
HiStar State as Structure

Cur

O

O

O

O

Cur
HiStar State as Structure

Label

Label

Cur
HiStar State as Structure
HiStar State as Structure

HiStar State as Structure

Label

Label

Low

Low

Low

High

Cur

Tuesday, October 22, 13
HiStar State as Structure
∀ w, s, n. \( \text{Wrap}(w) \land \text{Scan}(s) \land \text{Netd}(n) \Rightarrow \text{Flows}(s, w) \land \text{Flows}(w, n) \)
∀ w, s, n. Wrap(w) ∧ Scan(s) ∧ Netd(n) ⇒ Flows(s, w) ∧ Flows(w, n)
∀ w, s, n. \texttt{Wrap}(w) \land \texttt{Scan}(s) \land \texttt{Netd}(n) \Rightarrow \texttt{Flows}(s, w) \land \texttt{Flows}(w, n)
∀ w, s, n. \textit{Wrap}(w) \land \textit{Scan}(s) \land \textit{Netd}(n) \Rightarrow \\
\textit{Flows}(s, w) \land \textit{Flows}(w, n)
∀ w, s, n. Wrap(w) ∧ Scan(s) ∧ Netd(n) ⇒ Flows(s, w) ∧ Flows(w, n)
<table>
<thead>
<tr>
<th>Action</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create_cat(&amp;x)</code></td>
<td></td>
</tr>
</tbody>
</table>
HiStar State Transformers

Cur \rightarrow \text{Label} \rightarrow \text{Flows}

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_cat(&amp;x)</td>
</tr>
</tbody>
</table>

Tuesday, October 22, 13
# HiStar State Transformers

## Diagram

```
Cur -> Label -> Flows 
```

## Table

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create_cat(&amp;x)</code></td>
</tr>
</tbody>
</table>

Tuesday, October 22, 13
HiStar State Transformers

create_cat(&x)

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_cat(&amp;x)</td>
</tr>
</tbody>
</table>
HiStar State Transformers

The diagram illustrates the flow of information between states labeled `Cur`, `Decl`, and `Fresh`. Arrows indicate the direction of flow.

The action `create_cat(&x)` is defined in the table below.

<table>
<thead>
<tr>
<th>Action</th>
<th>Structure Transform</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create_cat(&amp;x)</code></td>
<td>Intro Fresh</td>
</tr>
<tr>
<td></td>
<td>Decl’(p, c) :=</td>
</tr>
<tr>
<td></td>
<td>Decl(p, c)</td>
</tr>
<tr>
<td></td>
<td>∨ (Cur(p) ∧ Fresh(c))</td>
</tr>
</tbody>
</table>
HiStar State Transformers

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>raise(&amp;x)</td>
</tr>
</tbody>
</table>
HiStar State Transformers

Action

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>raise(&amp;x)</td>
</tr>
</tbody>
</table>
HiStar State Transformers

\[ \text{Intro Fresh} \]

\[ \exists \ p. \ \text{Cur}(p) \land \text{Label}(p, l) \land x(c) \]

<table>
<thead>
<tr>
<th>Action</th>
<th>Structure Transform</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>raise(&amp;x)</code></td>
<td>Intro Fresh</td>
</tr>
<tr>
<td></td>
<td>High'(l, c) := High(l, c)</td>
</tr>
<tr>
<td></td>
<td>( \forall \exists \ p. \ \text{Cur}(p) \land \text{Label}(p, l) \land x(c) )</td>
</tr>
</tbody>
</table>
Summary: HiStar Semantics

- We can define the HiStar semantics as FOL predicate transforms and automatically generate a weaver for HiStar

- FOL predicate transforms can describe capability and DIFC semantics
Scanner Game

noop

create_cat(&c)

noop

raise(c)

drop_decl(c)

exec
Scanner Game

create_cat(&c)
raise(c)
sync_fork
drop_decl(c)
exec
...

Tuesday, October 22, 13
HiWeave $\alpha$ Performance

Generates code for clamwrap in < 3 mins
Programmer

```c
scanner() {
    sync_fork();
    ...
}
```

HiWeave

```c
HiStar Designer

create_cat(&c):
Decl'(p, c) := Decl(p, c) || ...
```

Weaver

```c
Generator

forall w, s, n.
Wrap(w) && ...
```

Tuesday, October 22, 13
Outline

1. Motivation, problem statement
2. Previous work: Capsicum
3. Ongoing work: HiStar
4. Open challenges
Open Challenges

• Automating abstraction refinement
• Automating error diagnosis
• Compositional synthesis
• Optimizing generated code
• Designing a policy logic
Automating Abstraction Refinement

- Picking the right abstraction predicates requires a lot of design effort
- Can we refine the abstraction predicates via counter-strategies?
Automating Error Diagnosis

- When weaver fails, it has a counter-strategy
- How can we simplify these when presenting them to the user?
Compositional Synthesis

• Real programs are structured as a composition of processes

• Policies are expressed naturally as conjunction of local, global policies

• Can we adapt compositional verification? [Long, ’89]
HiStar Logger

Local (security) policy: only Logger should be able to modify log

Logger

log.txt
HiStar Logger

Global (functionality) policy: under certain conditions, Logger will append log on behalf of Environment

Logger → Environment

Logger → log.txt
Optimizing Generated Code

- Mean-payoff games present an appealing cost model, but have high complexity in general
- Can we apply any domain specific optimizations?
Designing a Policy Logic

- The weaver generator allows a policy writer to declare policies purely over privileges
- What logic over privileges is easiest for a policy writer to understand?
- How do we evaluate value added?
Our Collaborators

**Capsicum-dev**
- Pawel Jakub Dawidek
- Khilan Gudka
- Ben Laurie
- Peter Neumann

**MIT-LL**
- Jeffrey Seibert
- Michael Zhivich

**HiStar**
- Nickolai Zeldovich

**TVLA**
- Mooly Sagiv
Questions?

HiWeave

scannner() {
  sync_fork();
  ...
}

Policy
forall w, s, n.
  Wrap(w) && ...

create_cat(&c):
Decl'(p, c) := Decl(p, c) || ...

Weaver Generator

scannner() {
  create_cat(&c);
  sync_fork();
  ...
}
Extra Slides
Three-valued logic

• Values: true, false, and unknown
• true & unknown = unknown
• false & unknown = false
Three-valued Structures

Diagram:

- Amb
- Cur
- Parent

Connections:

- Amb → Cur
- Parent
- Amb → Cur
- Parent
- Amb → Cur
- Parent
- Amb → Cur
- Parent
- Amb → Cur
- Parent
Abstraction Function

\[ \alpha_{\text{Cur}} \]

\[ \text{Amb} \rightarrow \text{Cur} \rightarrow \text{Parent} \]

\[ \text{Amb} \rightarrow \text{Amb} \rightarrow \text{Cur} \rightarrow \text{Parent} \rightarrow \text{Parent} \]
Abstraction Function

$$\alpha_{\{\text{Cur}\}}$$

Amb? \rightarrow Cur

Parent

Amb

Parent \rightarrow Parent \rightarrow Cur
Abstract Fork (def)

Action

fork()

Semantics

Intro Fresh

Amb(p) := Amb(p) || (Fresh(p) & E q. Cur(q) & Amb(q))
Abstract Fork (definite)

Action

fork()

Intro Fresh

Amb(p) := Amb(p) || (Fresh(p) & E q. Cur(q) & Amb(q))