End-to-End Verification of Information-Flow Security for C and Assembly Programs

David Costanzo, Zhong Shao, Ronghui Gu
Yale University

PLDI 2016
June 17, 2016
Information–Flow Security

**Goal**: formally prove an end-to-end information-flow policy that applies to the low-level code of these systems.
Challenges

- How to **specify** the information flow policy?
  - ideally, specify at high level of abstraction
  - allow for some well-specified flows (e.g., declassification)
Most systems are written in both C and assembly
• must deal with low-level assembly code
• must deal with compilation
  • even verified compilation may not preserve security
Challenges

- **How to prove security on low-level code?**
  - Security type systems (e.g., JIF) don’t work well for weakly-typed languages like C and assembly
  - How do we deal with declassification?
  - Systems may have “internal leaks” hidden from clients

- How to prove security for all components in a **unified** way that allows us to **link** everything together into a system-wide guarantee?
Contribution 1

New methodology to specify, prove, and propagate IFC policies with a single unifying mechanism: the observation function

- **specify** – expressive generalization of classical noninterference
- **prove** – general proof method that subsumes both security label proofs and information hiding proofs
- **propagate** – security-preserving simulations
Contribution 2

Application to a real OS kernel (CertiKOS [POPL15])

- First fully-verified secure kernel involving C and assembly, including compilation
- Verification done entirely within Coq
- Fixed multiple bugs (security leaks)
- Policy: user processes running over CertiKOS cannot influence each other in any way (IPC disabled)
Our Solution

Security Policy

Observation Function

Proof: spec secure wrt policy

Security-Preserving Simulation

End-to-End Guarantee

Verified

OS Syscall Spec

CMODULES

CompCert

CMods.s

AsmSpec

Asm.s

x86 Machine Model
1. Specifying security

2. Proving security (example)

3. Propagating security across simulations

4. Experience with CertiKOS security proof
“Alice’s behavior is influenced only by her own data.”
“Alice’s behavior is influenced only by her own observation.”
Observation Function

\[ \Theta : \text{principal} \rightarrow \text{program state} \rightarrow \text{observation} \]
(can be any type)

\[ S : \text{program state} \rightarrow \text{program state} \rightarrow \text{prop} \]

“spec S is secure for principal p”

\[ \forall \sigma_1, \sigma_2, \sigma'_1, \sigma'_2 . \]
\[ \Theta_p(\sigma_1) = \Theta_p(\sigma_2) \land S(\sigma_1, \sigma'_1) \land S(\sigma_2, \sigma'_2) \]
\[ \Rightarrow \]
\[ \Theta_p(\sigma'_1) = \Theta_p(\sigma'_2) \]
Example Observation Functions

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>(5, {A})</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>(17, {A,B})</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>(42, {B})</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>(13, {})</td>
<td></td>
</tr>
</tbody>
</table>

\[ \Theta_A \]

\[ \{A,B\} \]

\[ \{A\} \]

\[ \{B\} \]
Example Observation Functions

employee salaries

avg salary

Bob’s detailed event calendar

Bob’s available / unavailable time slots
Rest of Talk

1. Specifying security

2. Proving security (example)

3. Propagating security across simulations

4. Experience with CertiKOS security proof
Definition \( \text{va\_load}\) \(\sigma\) :=

\[
\text{match } \text{ZMap.get} \ (\text{PDX } \text{va}) \ (\text{ptpool } \sigma) \ \text{with}
\]

\[
PDEValid \_ \ pte =>
\]

\[
\text{match } \text{ZMap.get} \ (\text{PTX } \text{va}) \ pte \ \text{with}
\]

\[
| \text{PTEValid} \_ \ pg \_ =>
\]

\[
\text{Next} \ (\text{rs} \# \ \text{rd} < - \ \\
\text{FlatMem.load} \ (\text{HP } \sigma) \ (\text{pg}\times\text{PGSIZE} + \text{va}\%\text{PGSIZE}))
\]

\[
| \text{PTEUnPresent} => \text{exec\_pagefault } \sigma \ \text{va} \ \text{is}
\]

end

end.
Rest of Talk

1. Specifying security

2. Proving security (examples)

3. Propagating security across simulations

4. Experience with CertiKOS security proof
OS and compiler refinement proofs use simulations
Simulations may not preserve security!

Machine M

<table>
<thead>
<tr>
<th>x</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>42</td>
</tr>
</tbody>
</table>

swap(x,y)

<table>
<thead>
<tr>
<th>x</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>17</td>
</tr>
</tbody>
</table>

Machine N

<table>
<thead>
<tr>
<th>x</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>42</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
</tr>
</tbody>
</table>

z = x; x = y; y = z

<table>
<thead>
<tr>
<th>x</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>17</td>
</tr>
<tr>
<td>z</td>
<td>17</td>
</tr>
</tbody>
</table>

\[
R(\sigma_M, \sigma_N) := (\sigma_M(x) = \sigma_N(x) \land \sigma_M(y) = \sigma_N(y))
\]
• Define an observation function for each machine, $\Theta_M$ and $\Theta_N$
• Require that the simulation is security-preserving

Security-Preserving Simulation (for principal $p$)

$$\forall \sigma_1, \sigma_2, s_1, s_2 .$$

$$\Theta^M_p(\sigma_1) = \Theta^M_p(\sigma_2) \land R(\sigma_1, s_1) \land R(\sigma_2, s_2)$$

$$\Rightarrow$$

$$\Theta^N_p(s_1) = \Theta^N_p(s_2)$$

• No significant changes to CompCert were needed
Rest of Talk

1. Specifying security
2. Proving security (examples)
3. Propagating security across simulations
4. Experience with CertiKOS security proof
Certified functionally correct OS kernel with 32 layers

354 lines of assembly code, ~3000 lines of C code
- CompCert compiles C to assembly

Each layer has primitives that can be called atomically

Bottom layer MBoot is the x86 machine model

Top layer TSysCall contains 9 system calls as primitives
- init, vmem load/store, page fault, memory quota, spawn child, yield, print
CertiKOS Observation Function

- For a process p, the observation function is:
  - registers, if p is currently executing
  - the output buffer of p
  - the function from p’s virtual addresses to values
  - p’s available memory remaining (quota)
  - the number of children p has spawned
  - the saved register context of p
  - the spawned status and currently-executing status of p
CertiKOS Security Property

**TSysCall** → \( \Theta^S_p = \text{(as described)} \)

**MBoot** → \( \Theta^I_p = \text{p’s current output buffer} \)

\( B^I_p = \text{p’s “final” output buffer} \)

(whole–execution behavior)

**Generalized Noninterference:**

\[
\forall \sigma_1, \sigma_2, \sigma_1', \sigma_2'. \quad 
\Theta^S_p(\sigma_1) = \Theta^S_p(\sigma_2) \land (\sigma_1, \sigma_1') \in S \land (\sigma_2, \sigma_2') \in S \\
\Rightarrow \Theta^S_p(\sigma_1') = \Theta^S_p(\sigma_2')
\]

**End–to–End Security:**

\[
\forall \sigma_1, \sigma_2, s_1, s_2. \quad 
\Theta^S_p(\sigma_1) = \Theta^S_p(\sigma_2) \land (\sigma_1, s_1) \in R \land (\sigma_2, s_2) \in R \\
\Rightarrow B^I_p(s_1) = B^I_p(s_2)
\]
function **alice** {  
    int pid1 = proc_spawn();  
    yield();  
    int pid2 = proc_spawn();  
    print(pid2 - pid1 + 1);  
}  

function **bob** {  
    int secret = 42;  
    for i = 0 to secret {  
        proc_spawn();  
    }  
    yield();  
}
Solution to Leak

max children = 3
Conclusion

- New methodology using observation function to specify, prove, and propagate IFC policies
  - applicable to all kinds of real-world systems!

- Verification of secure kernel done fully within Coq
  - machine-checked proofs!

- **Future Work**: virtualized time (already done), more realistic x86 model, preemption, concurrency
Thank You!

CertiKOS info – [http://flint.cs.yale.edu/certikos/](http://flint.cs.yale.edu/certikos/)
PLDI certified artifact – ask me for link