CS428/528 Lecture 13: Information-Flow Security for mCertiKOS

Zhong Shao

Yale University
February 27, 2024

Based on the PLDI 2016 paper by Costanzo et al.
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)
Challenges

- 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?

No existing system solves all of these challenges!
Related Work

- Practical languages with security labels: JIF [1], FlowCaml [2]
  - Typed languages only, no C or assembly
  - No formal end-to-end guarantees


Related Work

- Dynamic label tracking and label checks (e.g., [1], [2])
  - Runtime exceptions can leak information
  - Declassifications are particularly problematic
  - Necessarily incomplete
    - dynamic label checks may disallow safe “internal leaks”
  - Execution overhead


Related Work

• seL4 (NICTA) end-to-end security proof [1]
  • no assembly code verification
  • everything verified w.r.t. a C-level machine model
    • ignores many intricacies of virtual memory address translation, page fault handling, and context switching
  • no guarantee that the C compiler maintains security

Contribution 1

New methodology to solve all of these challenges!

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

- specify – expressive generalization of classical noninterference that cleanly handles all kinds of declassifications
- prove – general proof method that subsumes both security label proofs and information hiding proofs
- propagate – security-preserving simulations and compilation
Contribution 2

Application to a real OS kernel (our group’s CertiKOS [1])

- 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)

Program Logic Basics

Program $C$

```plaintext
i := 0;
while (i < 64) do
  x := [A+i];
  if (x = 0)
    then
      output i;
    else
      skip;
  i := i+1;
```

Hoare Triple

```
{P} C {Q}
```

derive

soundness

1. $C$ doesn’t crash when $P$ holds
2. $C$ always takes $P$ states to $Q$ states
3. $C$ satisfies the security policy specified by $P$
Language

\[ E ::= x \mid n \mid E + E \mid \ldots \]

\[ B ::= E = E \mid \text{true} \mid \text{false} \mid B \land B \mid \ldots \]

\[ C ::= x := E \mid x := [E] \mid [E] := E \mid \text{output } E \mid \text{skip} \]

\[ \text{if } B \text{ then } C \text{ else } C \mid \text{while } B \text{ do } C \]
Example Program

\[ i := 0; \]

\[ \textbf{while} \ (i < 64) \ \textbf{do} \]

\[ x := [A+i]; \]

\[ \textbf{if} \ (x = 0) \]
\[ \ \textbf{then} \]
\[ \quad \text{output } i; \]

\[ \textbf{else} \]
\[ \quad \text{skip;} \]

\[ i := i+1; \]
Example Program Verification

\[
P = \bigcirc_{i \in [0, 63]} A+i \mapsto (n_i, l_i) \land (n_i = 0 \land l_i = Lo) \lor (n_i \neq 0 \land l_i = Hi)
\]

\[
\text{while } (i < 64) \text{ do}
\]

\[
\text{if } (x = 0) \text{ then}
\]

\[
\text{output } i;
\]

\[
\text{else}
\]

\[
\text{skip;}
\]

\[
i := i+1;
\]

\[
\text{Lo } \vdash \{P \land (\text{lbl}(i) = Lo)\}
\]

\[
\text{Lo } \vdash \{(\text{lo} \land (\text{lbl}(i) = Lo) \land x = 0 \land \text{lbl}(x) = Lo)\}
\]

\[
\text{Lo } \vdash \{P \land (\text{lbl}(i) = Lo)\}
\]

\[
\text{Hi } \vdash \{P \land (\text{lbl}(i) = Lo)\}
\]

\[
\text{Hi } \vdash \{P \land (\text{lbl}(i) = Lo)\}
\]

\[
\text{Lo } \vdash \{P \land (\text{lbl}(i) = Lo)\}
\]

\[
\text{Lo } \vdash \{P \land (\text{lbl}(i) = Lo)\}
\]

\[
\text{Lo } \vdash \{P\}
\]
Problems with this Approach

- **Language-specific**
  - bound to C-level reasoning and control flow constructs

- **Depends on specific code details**
  - any change in the system’s code would require reverification

- **Overlaps functional correctness with security concerns**
  - which aspects of $P$ are important for safety, and which for security?

- **Incomplete**
  - some programs are secure but cannot be verified in the logic
  - informal observation: all such programs can be rewritten to become verifiable
Ideal Solution

High Level Specification

Program $C$

\[
\begin{align*}
  i & := 0; \\
  \textbf{while } (i < 64) \textbf{ do } \\
    & x := [A+i]; \\
    & \textbf{if } (x = 0) \\
    & \quad \textbf{then } \\
    & \quad \quad \text{output } i; \\
    & \quad \textbf{else } \\
    & \quad \quad \text{skip;} \\
    & i := i+1;
\end{align*}
\]

Proof: Spec is Secure

Proof: \textbf{All} Code Implementing Spec is Secure

Conclusion: $C$ is secure
Ideal Solution – Achievable!

Security Policy

Proof: spec secure wrt policy

Security-Preserving Simulation and Whole-Execution Behaviors

End-to-End Guarantee

Observation Function

Simulation

x86 Machine Model

OS Syscall Spec

CMods.s

Asm.s

CompCert

AsmSpec

CModes
Rest of Talk

1. Specifying and proving security
2. Propagating security across simulations
3. CertiKOS security proof
4. Limitations and extensions
Program $C$

\begin{verbatim}
i := 0;
while (i < 64) do
    x := [A+i];
    if (x = 0)
        then
            output i;
        else
            skip;
    i := i+1;
\end{verbatim}

Proof: All Code Implementing Spec is Secure

Conclusion: $C$ is secure
Pure Noninterference

“Alice’s behavior is influenced only by her own data.”

Common end-to-end security property for systems using security-label reasoning.
More Complex Policies

```java
void printAvg() {
    int sum = 0;
    for int i = 0 to db.size-1
        sum += db[i];

    double avg = double(sum) / (db.size-1);
    print(avg);
}
```
More Complex Policies

Bob’s detailed event calendar

schedule meeting with Bob

Bob says: Alice can see only whether a day is free or not free
Bob’s detailed event calendar

<table>
<thead>
<tr>
<th>M</th>
<th>T</th>
<th>W</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bob says: Alice can see only whether a day is free or not free

void sched(event e) {
    for int i = 0 to cal.size-1 {
        int day = -1;
        if cal[i] == None {
            day = i;
            break;
        }
    }
    if day != -1
        cal[day] = Some e;
}
Generalized Noninterference

“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”

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

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

\[\Theta_A\]

\[\{A\}, \{B\}\]

\[\Theta_A\]

\[\{A, B\}\]
Average Salary

employee salaries $\Theta_A \rightarrow$ avg salary
### Average Salary

**Employee Salaries**

<table>
<thead>
<tr>
<th>Employee</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
</tr>
</tbody>
</table>

**Average Salary**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>35</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
</tbody>
</table>

**Result:**

- Employee 0: 5
- Employee 1: 17
- Employee 2: 42
- Employee 3: 13

**Average Salary:** 19.25
Average Salary

\[ \text{avg}(\sigma) = \frac{\sigma(0) + \sigma(1) + \ldots + \sigma(\text{size}-1)}{\text{size}-1} \]

\[ \text{printAvgSpec}(\sigma) = \sigma\{\text{out} \rightarrow \text{out}(\sigma) ++ [\text{avg}(\sigma)]\} \]

\[ \Theta_A(\sigma) = (\text{avg}(\sigma), \text{out}(\sigma)) \]

\textbf{Proof: Generalized Noninterference}

```c
void printAvg() {
    int sum = 0;
    for int i = 0 to db.size-1
        sum += db[i];
    double avg = double(sum) / (db.size-1);
    print(avg);
}
```
Bob says: Alice can see only whether a day is free or not free
Event Calendar

**Abstract**

\[
\text{first}(\sigma) = \begin{cases} 
\text{Some} & \text{if an empty slot exists} \\
\text{None} & \text{otherwise}
\end{cases}
\]

\[
schedSpec(e, \sigma) = \begin{cases} 
\sigma \{f \mapsto \text{Some } e\} & \text{if } \text{first}(\sigma) = \text{Some } f \\
\sigma & \text{if } \text{first}(\sigma) = \text{None}
\end{cases}
\]

\[
\Theta_A(\sigma) = \lambda i. \text{true} & \text{if } \sigma(i) = \text{None} \\
\text{false} & \text{otherwise}
\]

**Proof: Generalized Noninterference**

```java
void sched(event e) {
    for int i = 0 to cal.size-1 {
        int day = -1;
        if cal[i] == None {
            day = i;
            break;
        }
    }
    if day != -1
        cal[day] = Some e;
}
```
Definition \( \text{va\_load} \) \( \sigma \) rs rd :=

match ZMap.get (PDX va) (ptpool \( \sigma \)) with
  PDEValid _ pte =>
    match ZMap.get (PTX va) pte with
      | PTEValid pg _ =>
        Next (rs # rd <-
          FlatMem.load (HP \( \sigma \)) (pg*PGSIZE + va%PGSIZE))
      | PTEUnPresent => exec_pagefault \( \sigma \) va rs
    end
  end
end.
Rest of Talk

1. Specifying and proving security

2. Propagating security across simulations

3. Experience with CertiKOS security proof

4. Limitations and extensions
Program $C$

```plaintext
i := 0;
while (i < 64) do
  x := [A+i];
  if (x = 0) then
    output i;
  else
    skip;
  i := i+1;
```

**abstract**

**High Level Specification**

**Proof: Spec is Secure**

**Proof:** All Code Implementing Spec is Secure

**Conclusion:** $C$ is secure
Insecure Simulation

- OS and compiler refinement proofs use simulations
- Simulations may not preserve security!

\[ R(\sigma_M, \sigma_N) := (\sigma_M(x) = \sigma_N(x) \land \sigma_M(y) = \sigma_N(y)) \]
Propagating Security

- Define an observation function for each machine, $\Theta^M$ and $\Theta^N$
- Require that the simulation is security-preserving

**Security-Preserving Simulation** (for principal $p$)

\[
\forall \sigma_1, \sigma_2, s_1, s_2. \\
\Theta^M_p(\sigma_1) = \Theta^M_p(\sigma_2) \land R(\sigma_1, s_1) \land R(\sigma_2, s_2) \implies \Theta^N_p(s_1) = \Theta^N_p(s_2)
\]
Whole-Execution Behaviors

Can define $B_A(\sigma)$ if $\Theta_A$ is “monotonic” (behaves like an output buffer)

- only required for low-level implementation
- see PLDI2016 paper for technical details
End-to-End Security

If $R$ is a security-preserving simulation and $\Theta_p^I$ is monotonic, then:

**Generalized Noninterference:** \[ \forall \sigma_1, \sigma_2, \sigma_1', \sigma_2'. \]
\[ \Theta_p^S(\sigma_1) = \Theta_p^S(\sigma_2) \land \sigma_1 \rightarrow \sigma_1' \land \sigma_2 \rightarrow \sigma_2' \]
\[ \Rightarrow \Theta_p^S(\sigma_1') = \Theta_p^S(\sigma_2') \]

**End-to-End Security:** \[ \forall \sigma_1, \sigma_2, s_1, s_2. \]
\[ \Theta_p^S(\sigma_1) = \Theta_p^S(\sigma_2) \land (\sigma_1, s_1) \in R \land (\sigma_2, s_2) \in R \]
\[ \Rightarrow B_p^I(s_1) = B_p^I(s_2) \]
Rest of Talk

1. Specifying and proving security

2. Propagating security across simulations

3. Experience with CertiKOS security proof

4. Limitations and extensions
CertikOS Overview

- 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 \textbf{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 layer** \[ \Theta^S \text{ defined as described} \]
- **MBoot machine model** \[ \Theta^I_p = p's \text{ output buffer} \]

\[ R \text{ is a security-preserving simulation} \]
\[ \Theta^I_p \text{ is monotonic} \]

**Generalized Noninterference:**
\[ \forall \sigma_1, \sigma_2, \sigma_1', \sigma_2'. \]
\[ \Theta^S_p(\sigma_1) = \Theta^S_p(\sigma_2) \land \sigma_1 \to \sigma_1' \land \sigma_2 \to \sigma_2' \]
\[ \Rightarrow \Theta^S_p(\sigma_1') = \Theta^S_p(\sigma_2') \]

**End-to-End Security:**
\[ \forall \sigma_1, \sigma_2, s_1, s_2. \]
\[ \Theta^S_p(\sigma_1) = \Theta^S_p(\sigma_2) \land (\sigma_1, s_1) \in R \land (\sigma_2, s_2) \in R \]
\[ \Rightarrow B^I_p(s_1) = B^I_p(s_2) \]
## Evaluation

### Security of Primitives (LOC)

<table>
<thead>
<tr>
<th>Operation</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>147</td>
</tr>
<tr>
<td>Store</td>
<td>258</td>
</tr>
<tr>
<td>Page Fault</td>
<td>188</td>
</tr>
<tr>
<td>Get Quota</td>
<td>10</td>
</tr>
<tr>
<td>Spawn</td>
<td>30</td>
</tr>
<tr>
<td>Yield</td>
<td>960</td>
</tr>
<tr>
<td>Start User</td>
<td>11</td>
</tr>
<tr>
<td>Print</td>
<td>17</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1621</td>
</tr>
</tbody>
</table>

### Security Proof (LOC)

<table>
<thead>
<tr>
<th>Component</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitives</td>
<td>1621</td>
</tr>
<tr>
<td>Glue</td>
<td>853</td>
</tr>
<tr>
<td>Framework</td>
<td>2192</td>
</tr>
<tr>
<td>Invariants</td>
<td>1619</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>6285</td>
</tr>
</tbody>
</table>

Time needed for Coq proof effort: ~ 6 months
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
Rest of Talk

1. Specifying and proving security

2. Propagating security across simulations

3. Experience with CertiKOS security proof

4. Limitations and Extensions
   a. Model Fidelity
   b. Virtualized Time
   c. Top-Level CertiKOS Theorem
Machine Model Fidelity

- Gaps between MBoot machine model and the physical x86 hardware
  - **Completeness** – some unmodeled assembly instructions (e.g., RDTSC)
  - **Soundness** – must trust that we modeled x86 instructions faithfully
  - **Safety** – must assume that users never execute code modeled as undefined behavior

Future plans to deal with safety gap:

- Define a user-level machine model with three types of instructions
  - **Interrupt** – trap into the kernel to handle a privileged instruction or syscall
  - **Load/Store** – access global heap according to the kernel’s load/store specs
  - **Other** – other user-level instructions, which are only allowed to use local registers

- Instructions of first two types are proved to be safe
- Instructions of third type are safe due to restriction to local registers
New Feature: Virtualized Time

```
function alice {
    int t0 = gettime();
    while (true) {
        for i = 0 to 10^6 {
            // do some work...
        }
        int t = gettime();
        print(t - t0);
        yield();
    }
}
```

```
function bob {
    int t0 = gettime();
    while (true) {
        for i = 0 to 10^6 {
            // do some work...
        }
        int t = gettime();
        print(t - t0);
        yield();
    }
}
```
New Feature: Virtualized Time

void stoptime() {
    int p = get_cid();
    int t = rd_tsc();
    sum_p += t - cur;
}

int gettime() {
    int p = get_cid();
    int t = rd_tsc();
    return (sum_p + (t - cur));
}

void starttime() {
    cur = rd_tsc();
}

int gettime() {
    int p = get_cid();
    int t = rd_tsc();
    return (sum_p + (t - cur));
}

void stoptime() {
    int p = get_cid();
    int t = rd_tsc();
    sum_p += t - cur;
}

void starttime() {
    cur = rd_tsc();
}
New Feature: Virtualized Time

Hacker: The current time is 65735500.
Hacker: Ok, yielding now to let Alice execute her program. See you later.

Alice: I did something secret, the time is now 88014576.
Alice: I did something secret, the time is now 116917548.
Alice: I did something secret, the time is now 146445524.
Alice: I did something secret, the time is now 203650560.
Alice: I did something secret, the time is now 205546124.
Alice: I did something secret, the time is now 300386953.
Alice: I did something secret, the time is now 427359527.
Alice: I did something secret, the time is now 429350439.
Alice: I did something secret, the time is now 456707395.
Alice: I've finished my top secret computation!
Alice: It took me 396460583 cycles. I sure hope no one was able to learn anything
Alice: about what I did. Goodbye!

Hacker: And we're back! Let's see what we can figure out about Alice's secret computation.
Hacker: The time is now 104580368. That's only 38844868 cycles since last time.
Hacker: I guess Alice's execution had no effect on my view of time. Oh well.
dsc5@fromage:~/mycertikos-secure-tsc/certikos/kernel$
End-to-End Security in CertiKOS

End-to-End Security: \[ \forall \sigma_1, \sigma_2, s_1, s_2 . \]
\[ \Theta^S_p(\sigma_1) = \Theta^S_p(\sigma_2) \land (\sigma_1, s_1) \in R \land (\sigma_2, s_2) \in R \]
\[ \Rightarrow B^I_p(s_1) = B^I_p(s_2) \]

Requires understanding and trusting the observation function.
But CertiKOS enforces pure isolation on processes; can we do better?

Proposed solution (not yet completed):
1. Define \( Spawning(p) = \) process \( p \) was just spawned by the kernel
2. Prove: \( \forall \sigma_1, \sigma_2 \in Spawning(p) . \Theta^S_p(\sigma_1) = \Theta^S_p(\sigma_2) \)

\[ \Rightarrow \text{End-to-end security theorem is independent from choice of observation function!} \]
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
  - higher-level process isolation theorem (independent of observation function choice)
  - more realistic x86 model
  - preemption
  - concurrency

---

Observation Function

Security Policy

OS Syscall Spec

CModes
CompCert
AsmSpec
CMods.s
Asm.s

x86 Machine Model

Proof: spec secure wrt policy

Security-Preserving Simulation and Whole-Execution Behaviors

End-to-End Guarantee