Concurrent, Compositional Declassification without Reinventing the Wheel

Toby Murray
University of Melbourne,
Data61 (formerly NICTA), CSIRO

IFIP WG2.3, May 2018

Models, Assumptions & World-Machine (OS)

domain hypotheses:
- h/w performs to its spec
- Bootloader correct
- …

adversary tries to break these

formal spec

adversary expectations:
- cannot change hardware
- which interfaces accessible
- compute power
- cannot observe timing

system model

adversary spec

deployment assumptions*
- physically secured hardware
- device drivers
- initialized data structures

full model

Right-Hand Side Problem

Not necessarily u

machine

deployment domain

adversary

Definitely not under our control

world = domain + adversary

= machine + world

Paul van Oorschot

deployment domain = environment without adversary

*supported by: deployment and config. constraints

Confidentiality (Information Flow) Proofs

CoCon

CoSMed

SEQUENTIAL

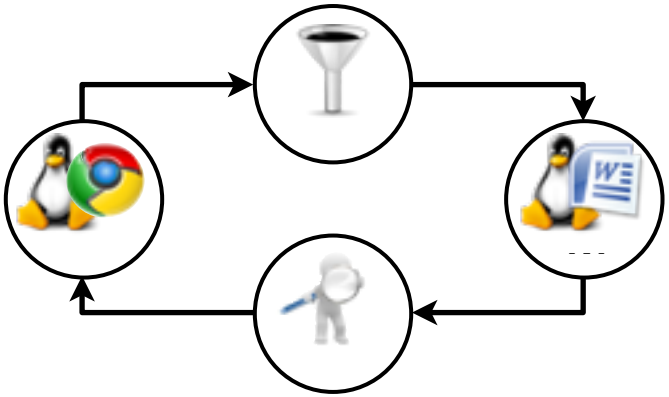
SAFE

Quark

Sequential vs Whole-System

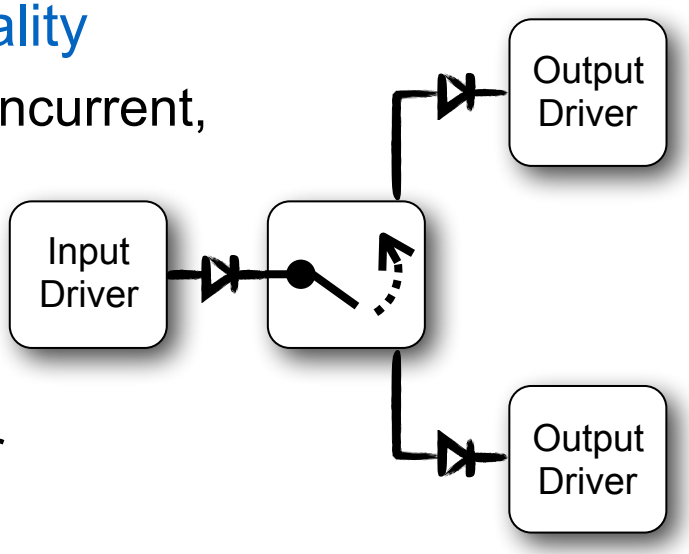
Sequential Confidentiality

- assumes application components are maximally hostile
- proved security only by reasoning about mandatory, enforced abstractions
- simple static policies only



Concurrent, Whole-System Confidentiality

- reasons about interactions between concurrent, application components
- proves that their composition yields a secure system
- runtime state-dependent and data-dependent policies and behaviour



Mark Beaumont

CDC

The Cross Domain Desktop Compositor

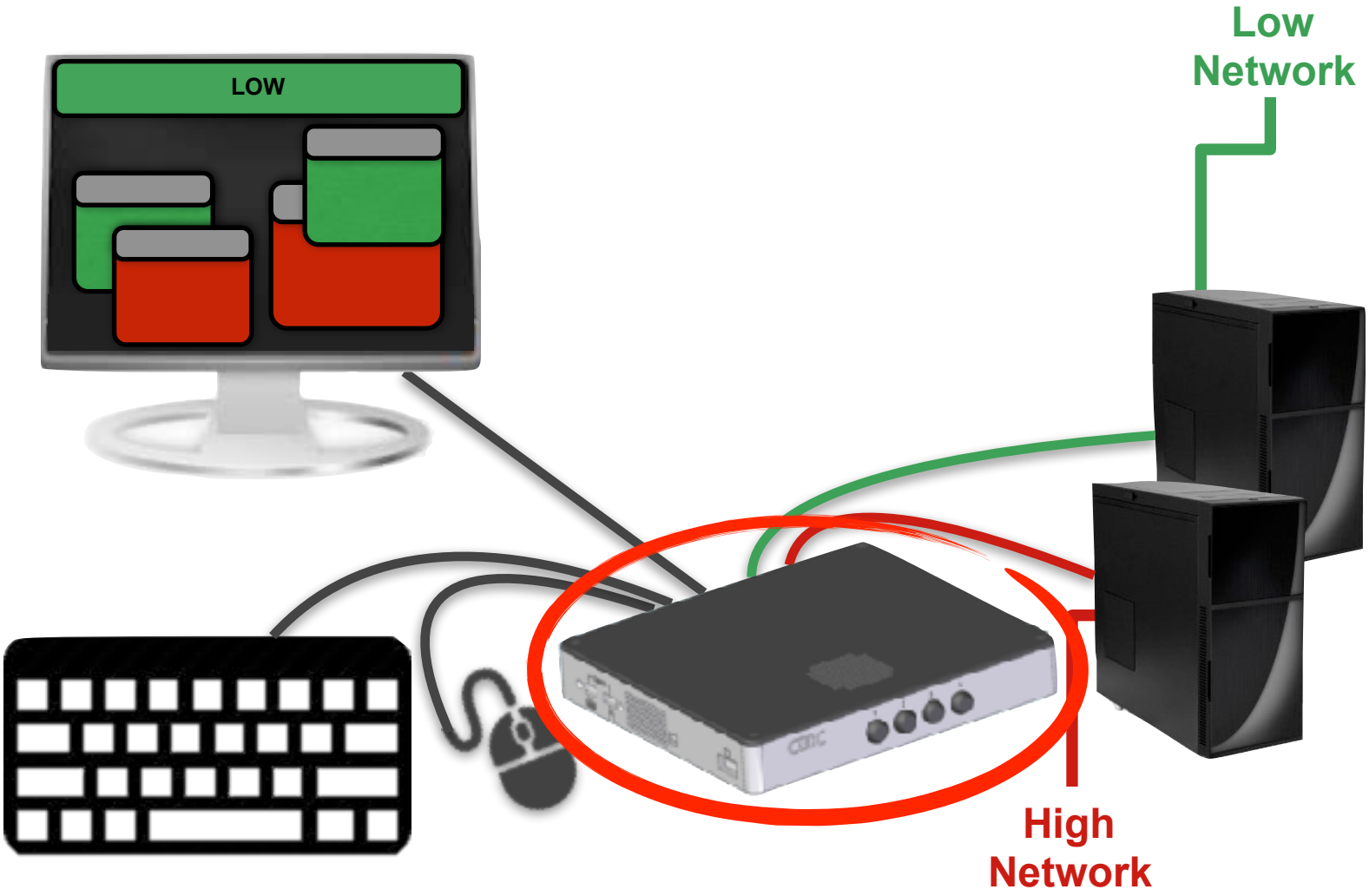
Kevin Elphinstone

Australian Government
Department of Defence
Defence Science and Technology Group

DATA 61

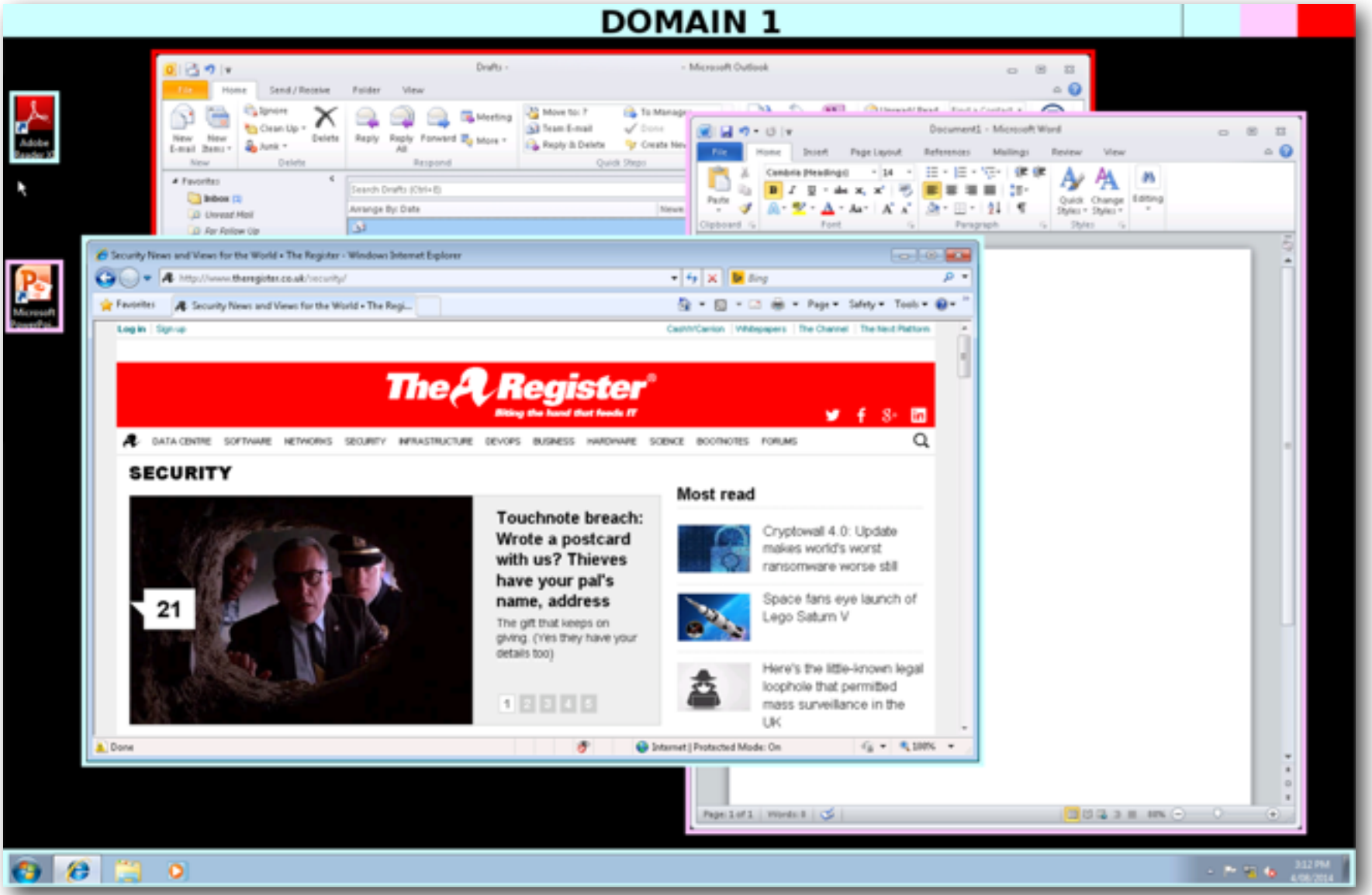
CSIRO

Cross Domain Desktop Compositor

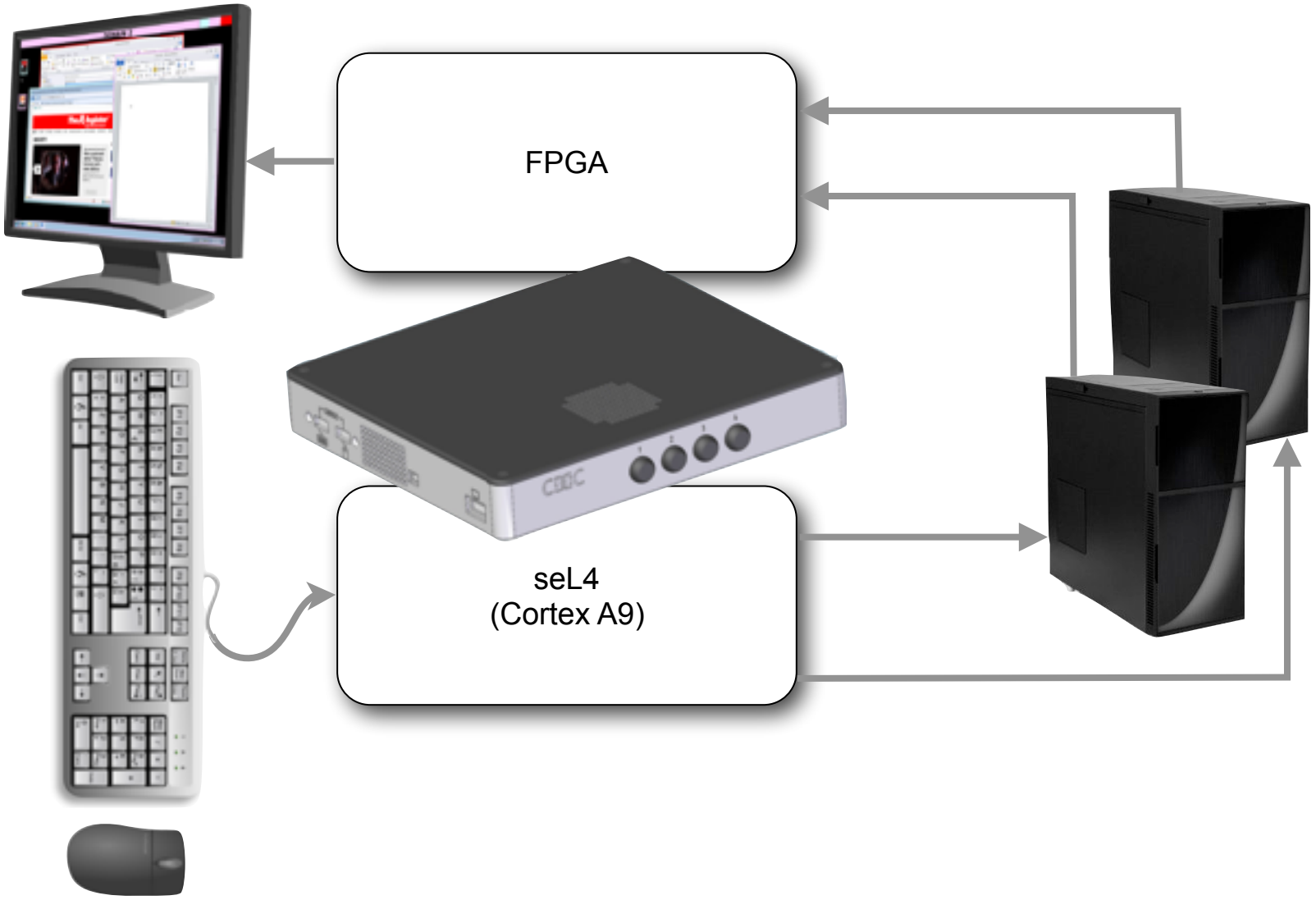


CDDC

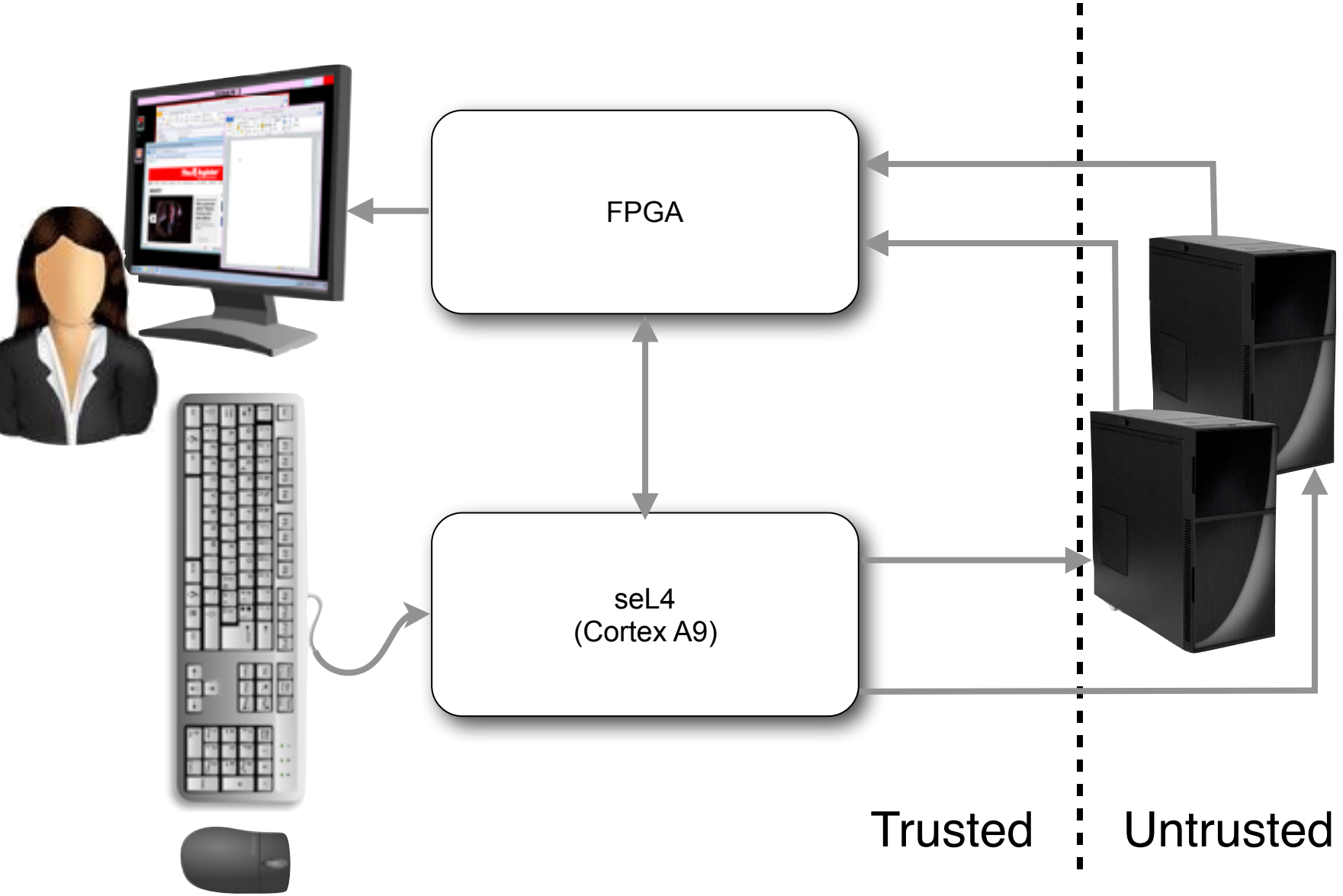
CDDC User Experience



CDDC Architecture

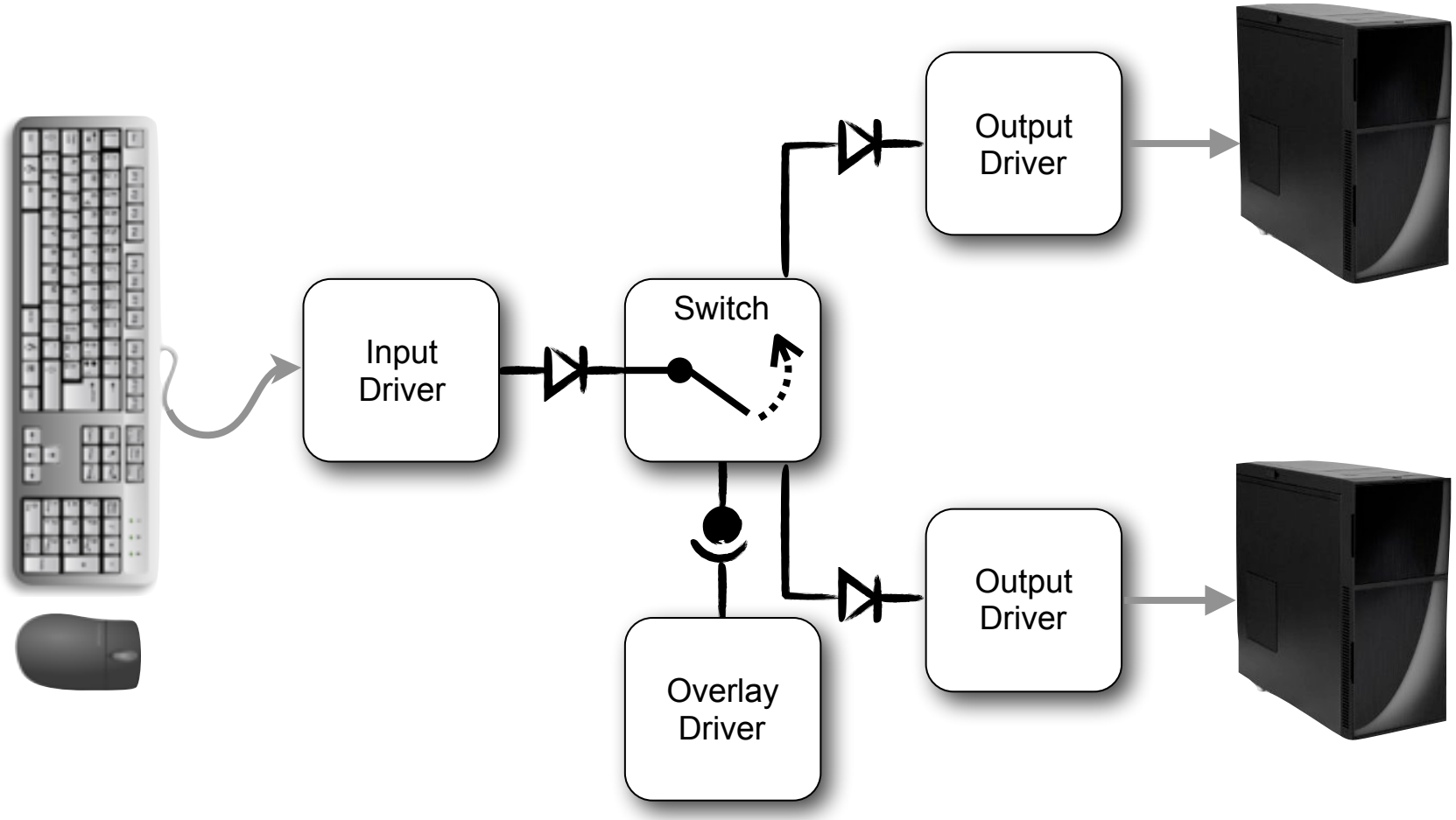


Threat Model

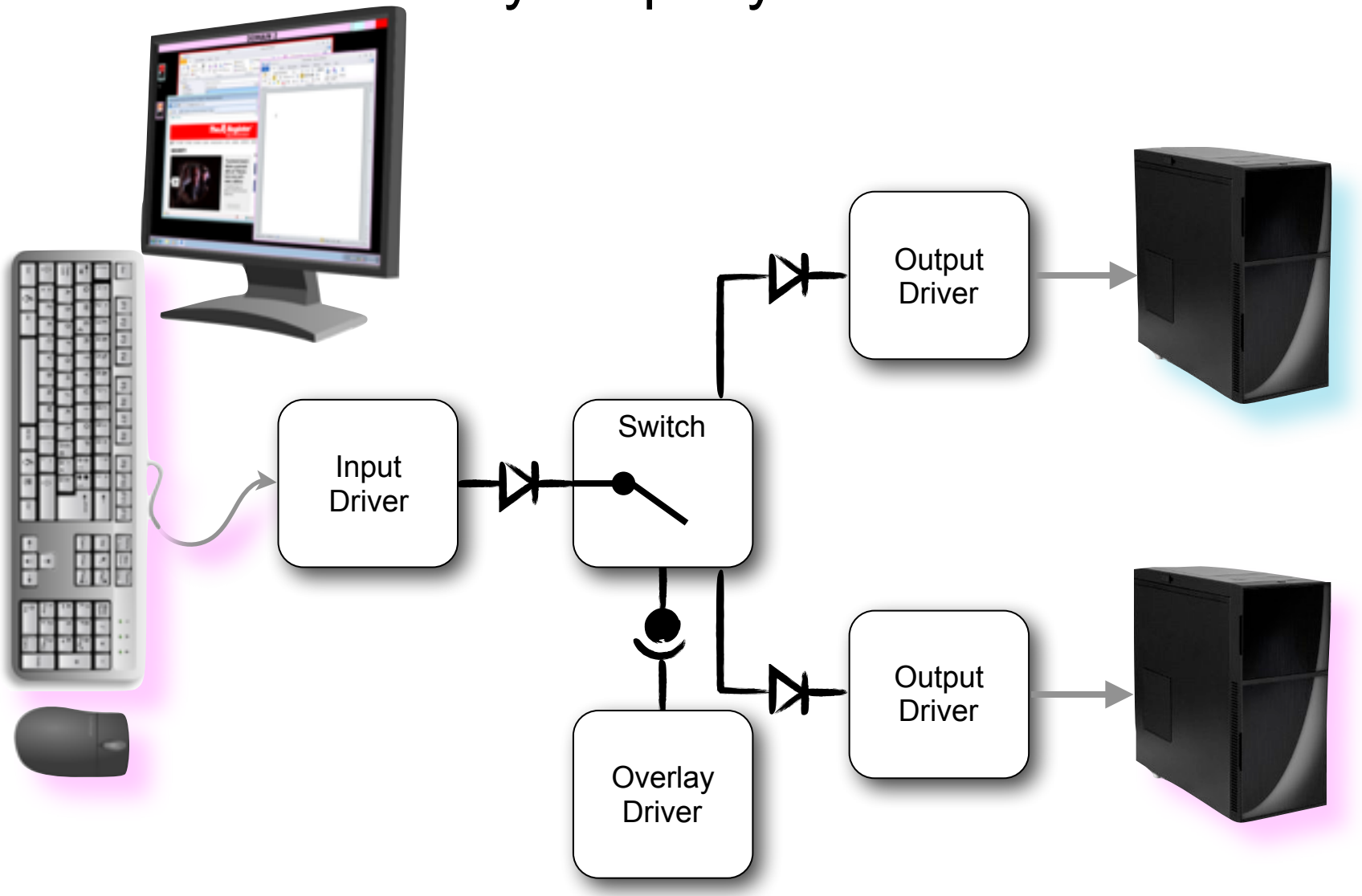


seL4-Based Software Architecture

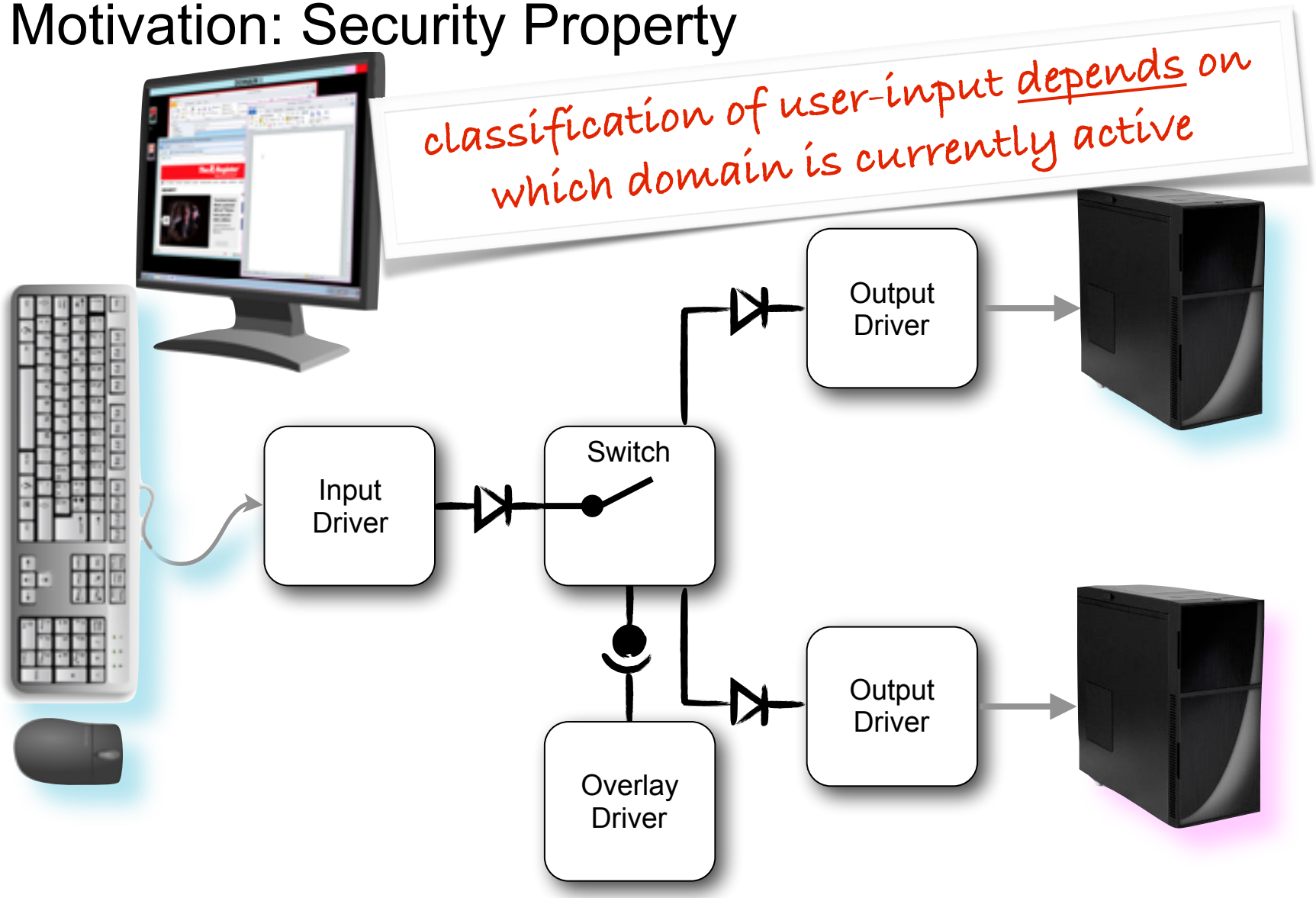
(ignoring device administration and configuration, plus keyboard LED control)



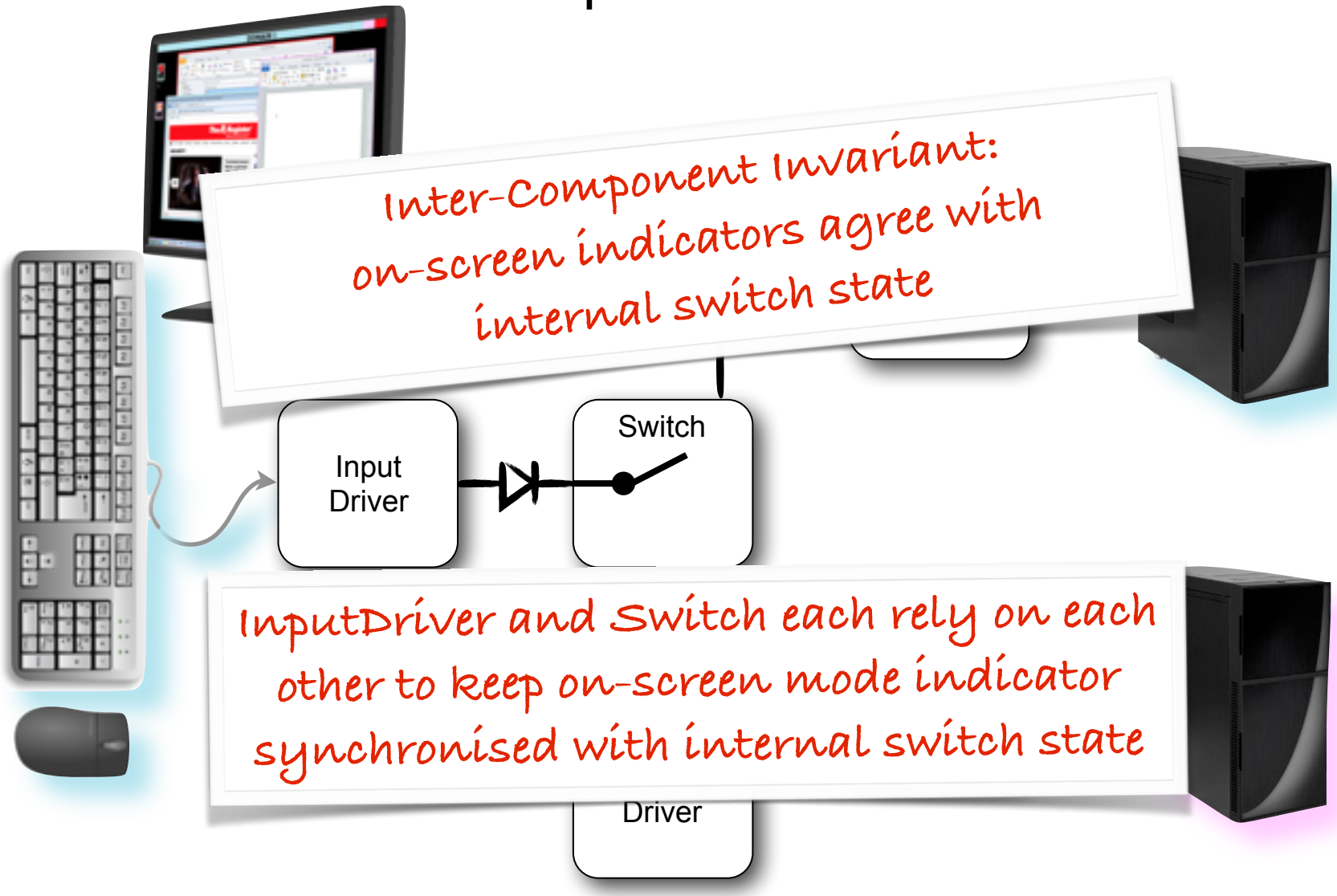
Motivation: Security Property



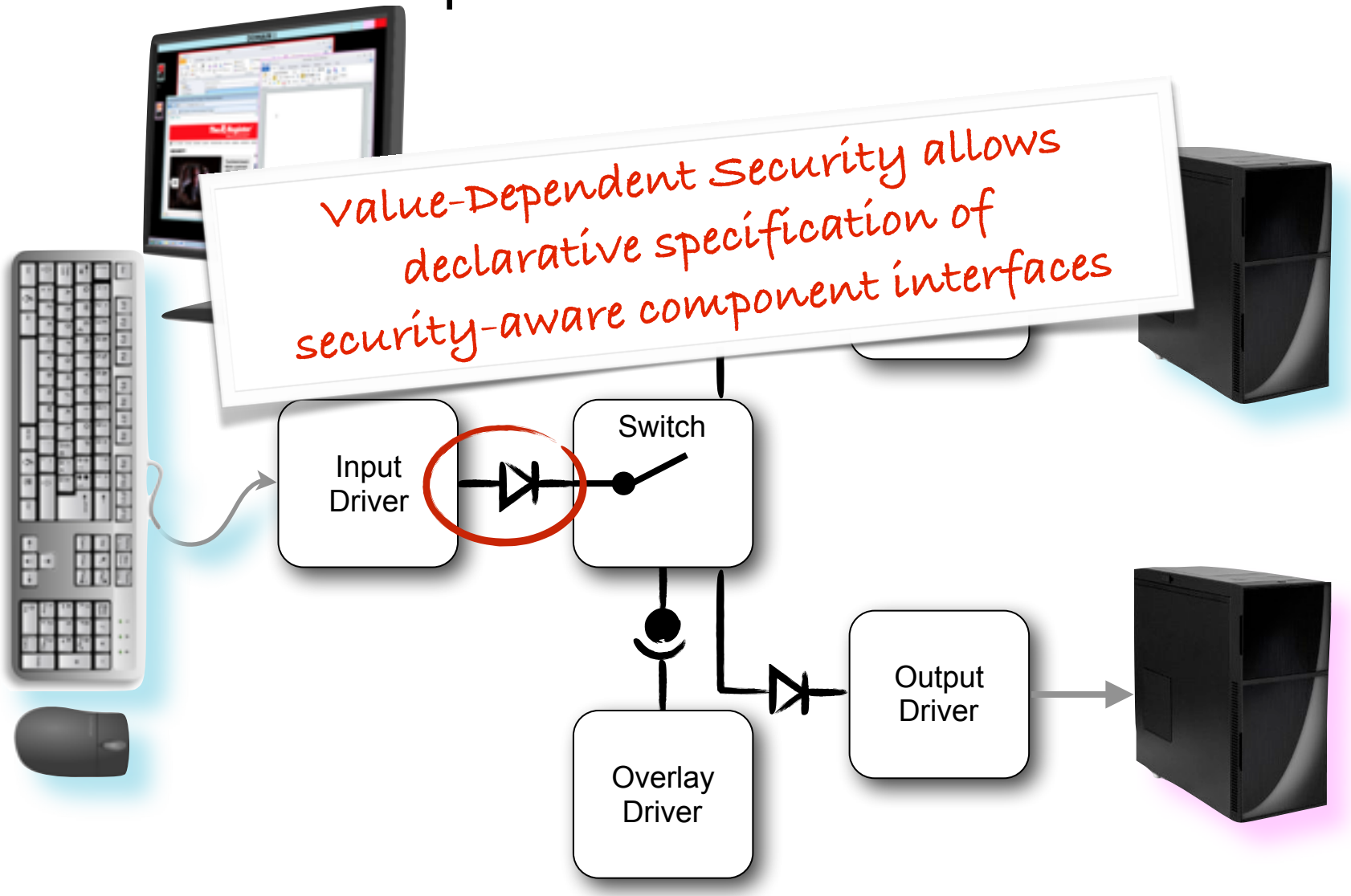
Motivation: Security Property



Motivation: Inter-Component Invariants



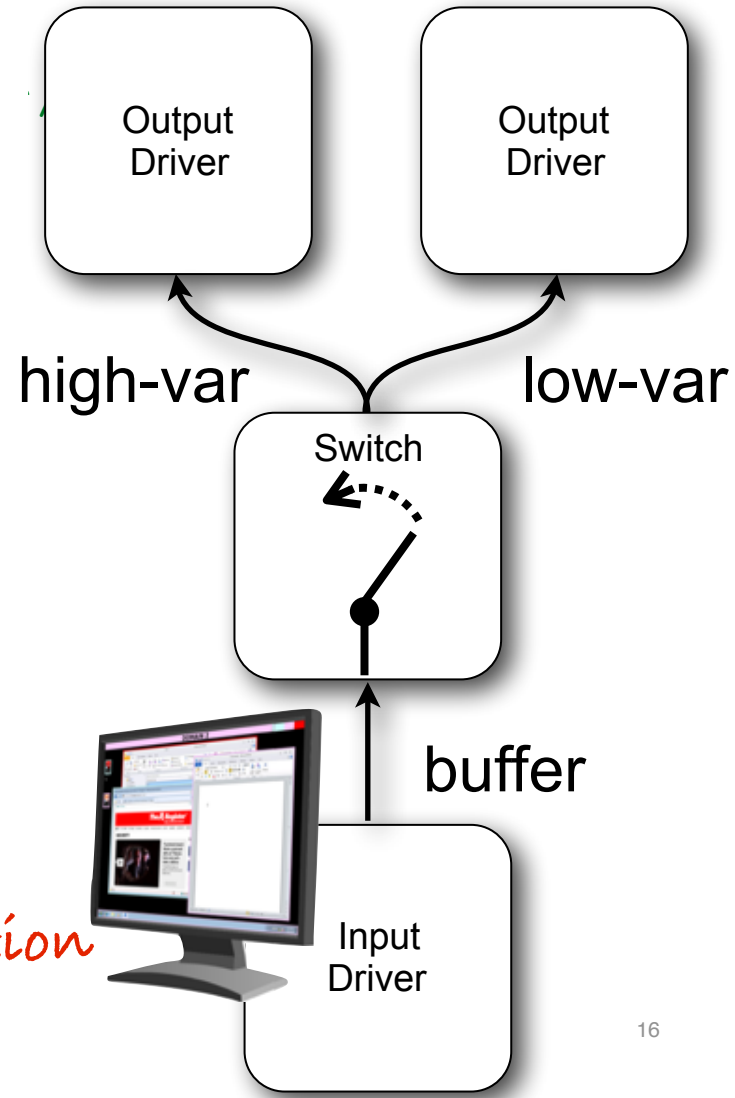
Motivation: Component Interfaces



Motivation: Shared Memory Concurrency

Switch (idealised, obviously):

```
temp := buffer;
if sMode == 0 then
   low-var := temp
else
   high-var := temp
endif;
temp := 0
```



sMode is switch's mode

dMode defines buffer's classification

Compositional, Value-Dependent Security

control variable (must be statically Low)

```
var dMode                 // clas: Low
var high-var              // clas: High
var low-var, sMode        // clas: Low
var buffer                // clas: dMode == 0 ? Low : High
lock l                    // NRW: {dMode,sMode,temp}
                          // inv: (dMode == sMode)

    lock(l);
    /* {dMode,sMode,temp} += AsmNoRW
         acquire (dMode == sMode) */
    temp := buffer;
    if sMode == 0 then
      low-var := temp
    else
      high-var := temp
    endif;
    temp := 0
```

value-dependent classification

lock footprint and invariant

(simplified, of course)

dynamic, local assumptions about environment, for compositional reasoning (acquired and released via locks)

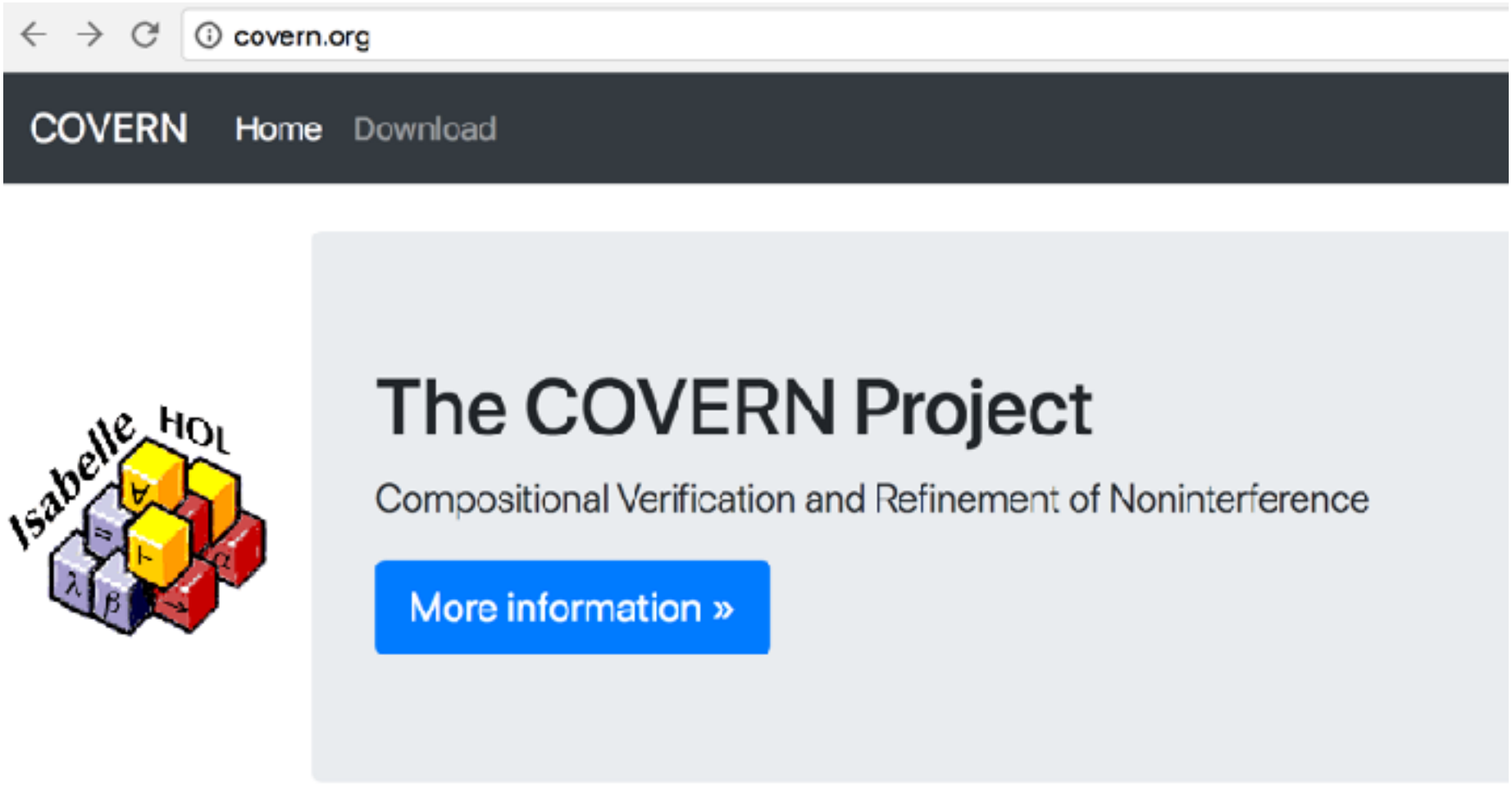
THE COVERN LOGIC

Robert Sison

$$\Gamma, L, P \ \{c\} \ \Gamma', L', P'$$

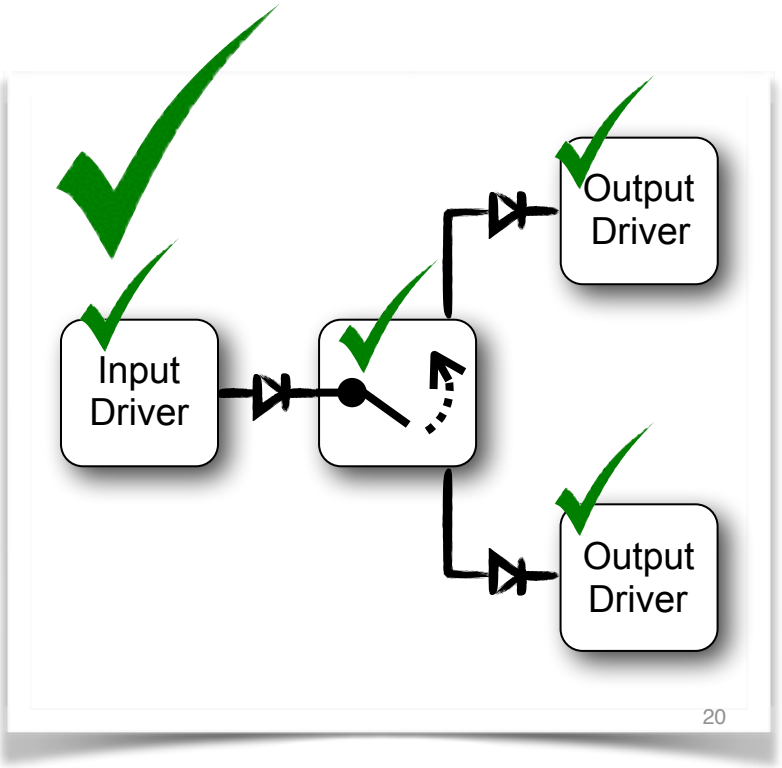
Kai Engelhardt

http://covern.org

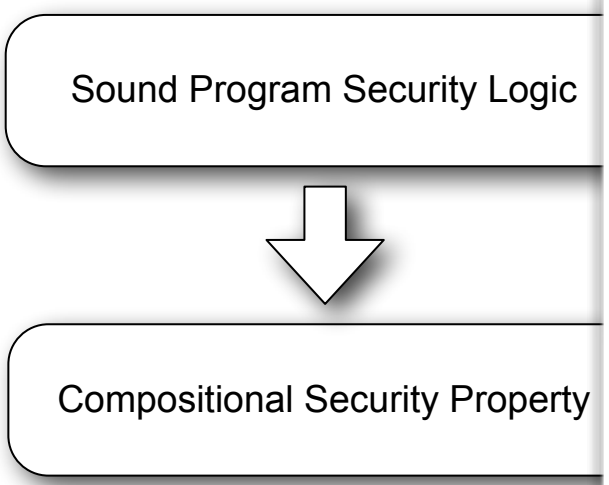


Verification Framework: Requirements

Compositional Security Property

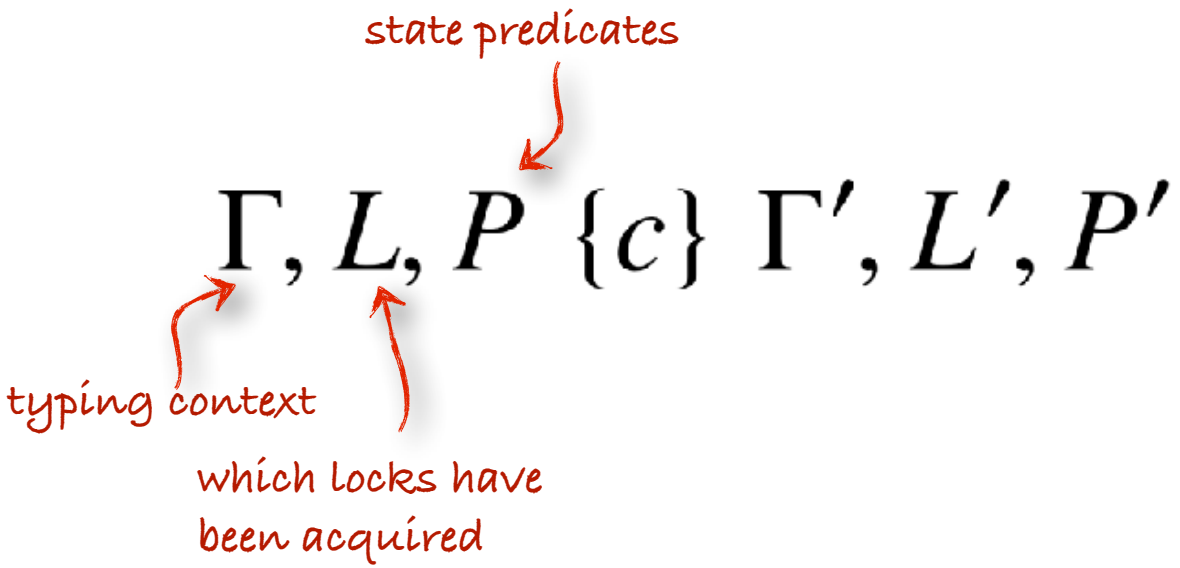


Verification Framework: Requirements




```
lock(l);
/* {dMode,sMode,temp} += AsmNoRW
     acquire (dMode == sMode) */
temp := buffer;
if sMode == 0
  low-var := t
else
  high-var := temp
endif;
temp := 0
```

Types and Typing Judgements



types depend on memory

we encode them as state predicates

type interpretation: $[\![t]\!]_{mem} \equiv \mathit{if}\ \lfloor t \rfloor_{mem}\ \mathit{then}\ \mathsf{Low}\ \mathit{else}\ \mathsf{High}$

subtype: $t \leq:_P t' \equiv (P \wedge t') \vdash t$

Type System: Example in Action

[]	{}	**true**	`lock(l);`
[t : **true**]	{l}	(sM = dM)	`temp := buffer;`
[t : (dM = 0)]	{l}	(sM = dM)	`if sMode == 0 then`
[t : (dM=0)]	{l}	(sM=dM ∧ sM=0)	`low-var := temp`
Γ	$\mathcal{S}$	P	

`else`

`  high-var := temp`

`endif;`

`temp := 0`

Is (dM=0) a subtype of **true**, under (sM=dM ∧ sM=0) ?

(sM=dM ∧ sM=0 ∧ **true**) ⊢ (dM=0) ?

lock l protects `sMode`, `dMode` and `temp`, with invariant `sMode = dMode`

CONCLUSION

General Logic for Verifying IFC for Concurrent Programs

WRONG APPROACH?

- concurrency → compositionality ~~→~~ timing-sensitivity, assume-guarantee reasoning
- ~~inter-component contracts on shared memory:~~
  - ~~data invariants + value-dependent classifications~~
- concurrent IFC reasoning borrowing ideas from CSLs
- modelled and verified security-critical s/w functionality of CDDC

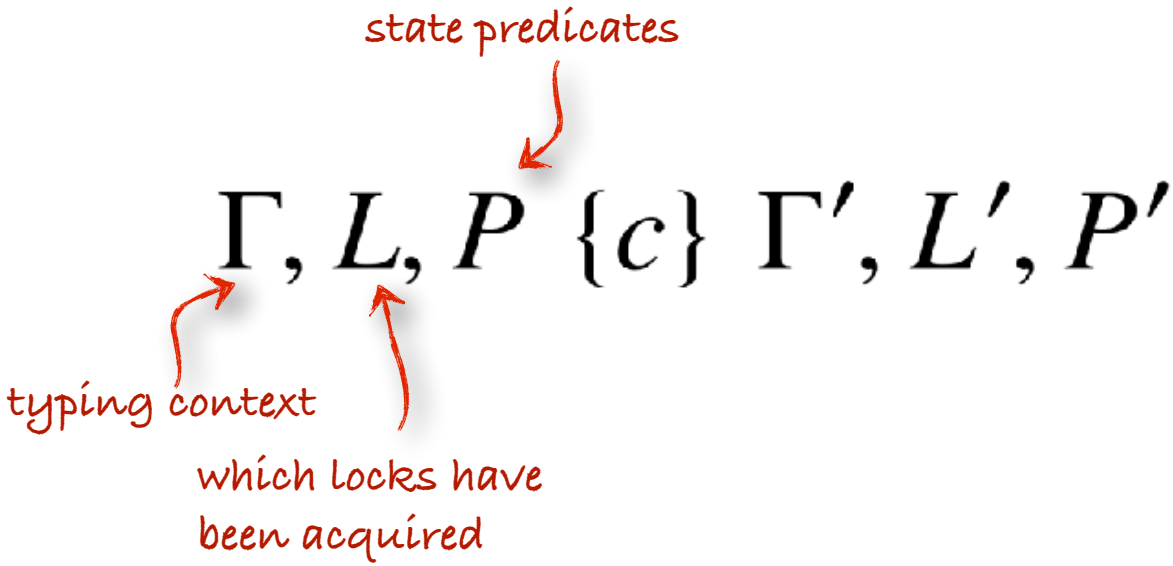
Still lots more to be done.

Daniel Schoepe

Andrei Sabelfeld

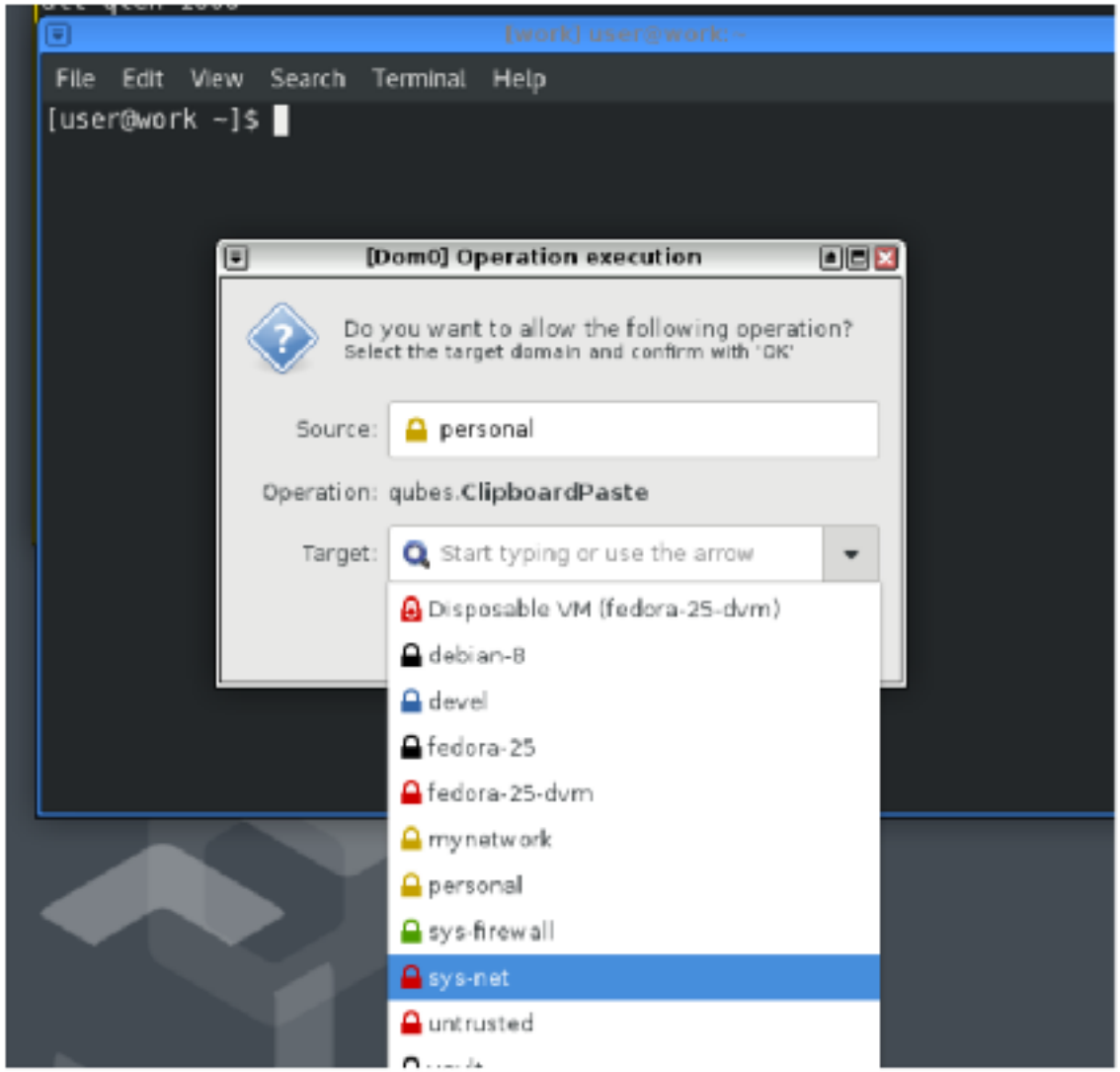
Drawbacks

- No support for declassification
  - Compositional declassification in general not well understood



- Our logic does double duty:
  - Functional correctness (P and L)
  - Security (Γ)

Declassification (in Qubes OS)



Declassification

```
secret := …  // get secret from somewhere
output(H, "Declassify: " + secret);
answer := input(L);
if (answer = 1)
then declassify(secret);
```

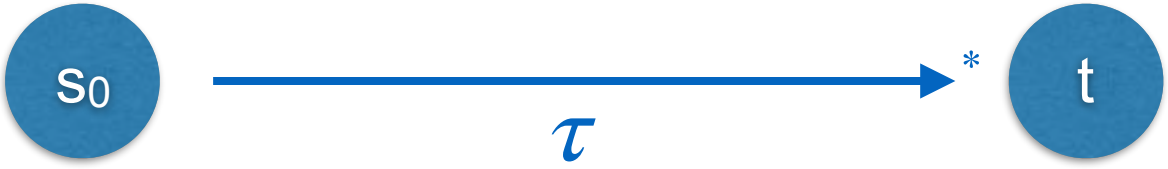
We model allowed declassifications as *declassification predicates* on the current state *s* and the declassification event *a*

- e.g. $D(\alpha,s) \equiv last\_output(s,H) = \alpha \wedge last\_input(s,L) = 1$

- States *s* include full trace *trace(s)* of input/output events so far

Defining Secure Declassification

There are so many declassification definitions.
We use a modern, **knowledge-flavoured** definition.



Attacker can observe:

- initial Low (public) memory
- Low parts of the trace τ (I/O on Low channels)

Attacker's **uncertainty**:

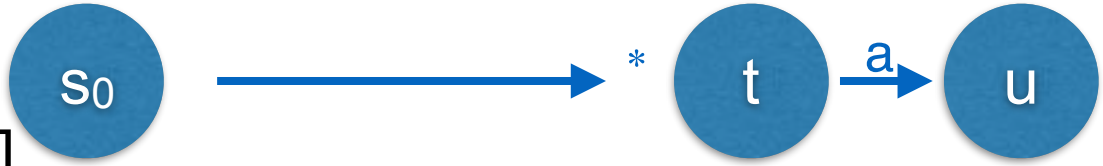
- we could have started in any state s that has the same initial Low memory and produced the same Low trace
- $uncertainty(s_0,t) \equiv$

  $\{s_0' \mid s_0' =_L s_0 \wedge \exists t'.\ s_0' \rightarrow^* t' \wedge trace(t') =_L trace(t)\}$

Secure Declassification

Security holds iff, whenever:

- $s_0 \rightarrow^* t$
- $t \rightarrow u$
- trace(u) = trace(t).[*a*]



Then:

- If *a* is declassification then *D(a,t)* must hold
- Otherwise: uncertainty(s_0,t) ⊆ uncertainty(s_0,u) must hold

Aside: This is a bit weak for programs that branch on secrets:

```
if (birthYear > 2000)
    Declassify(birthDay);
else
    Declassify(birthMonth);
```

DISALLOWED

How to prove security compositionally?

A compositional knowledge-based definition?

Too hard

Observation: Type systems / program logic just defines and proves a (relational) invariant.

Why not just do that directly?

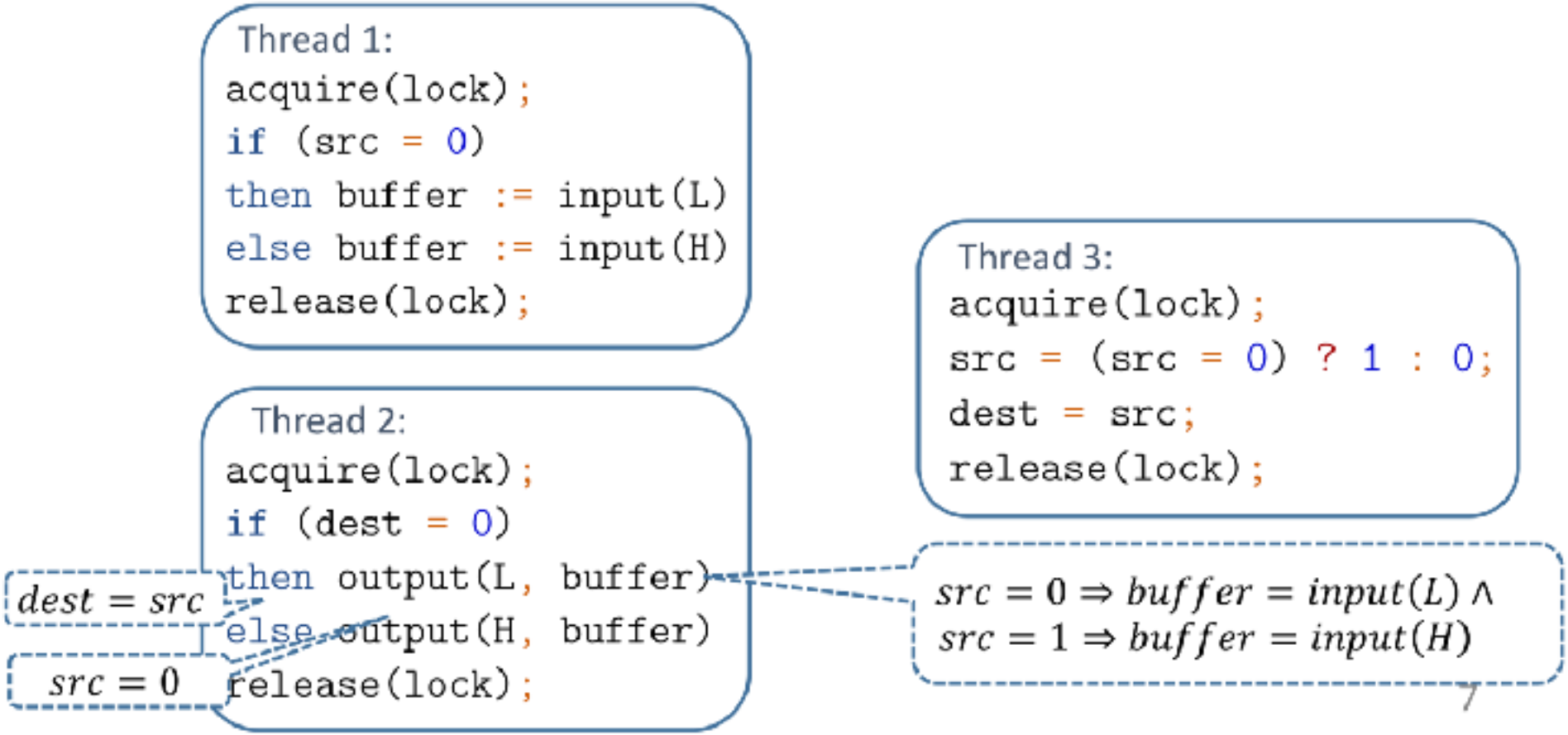
Also, let's decouple the functional correctness and security parts.

Example

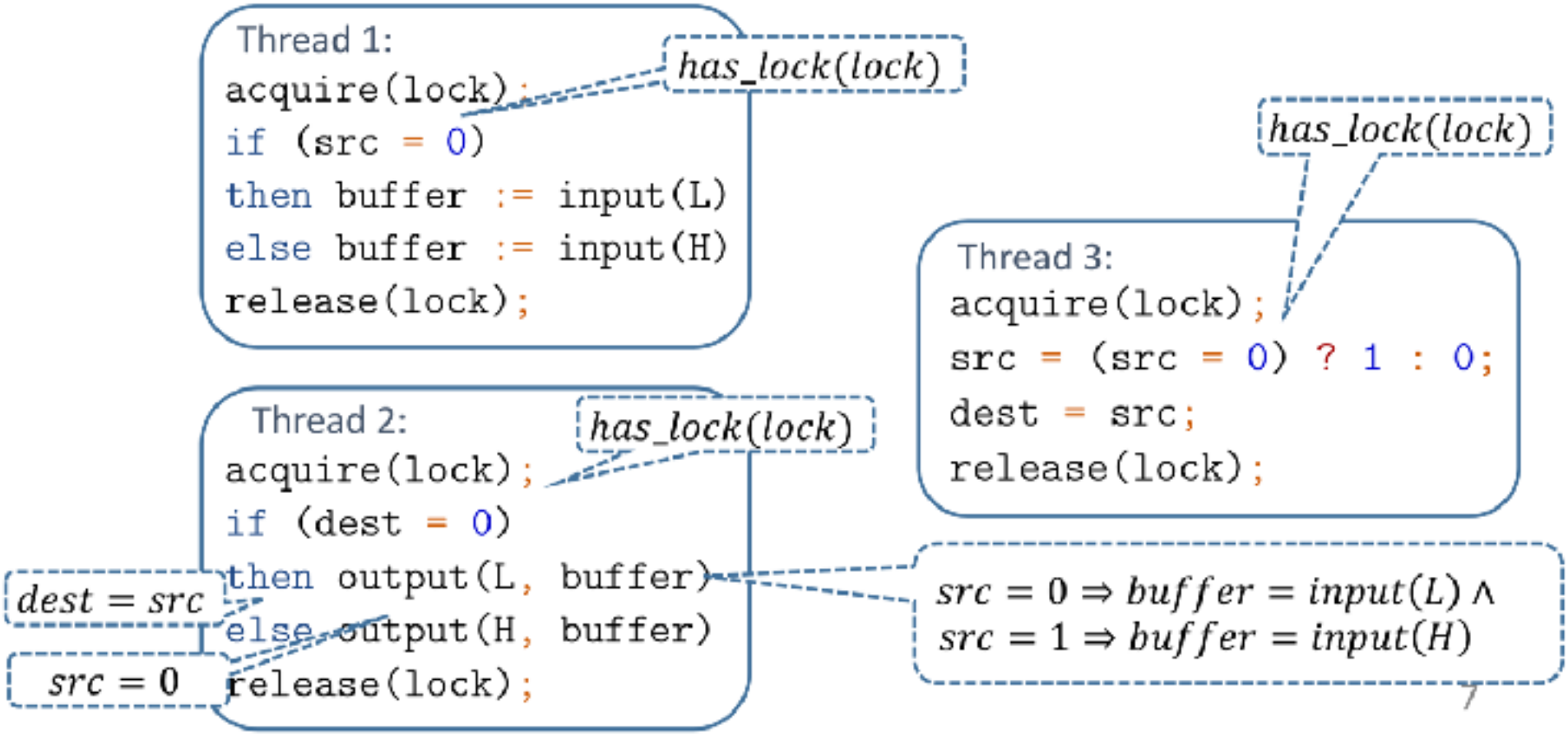
```
Thread 1:
acquire(lock);
if (src = 0)
then buffer := input(L)
else buffer := input(H)
release(lock);
```

7

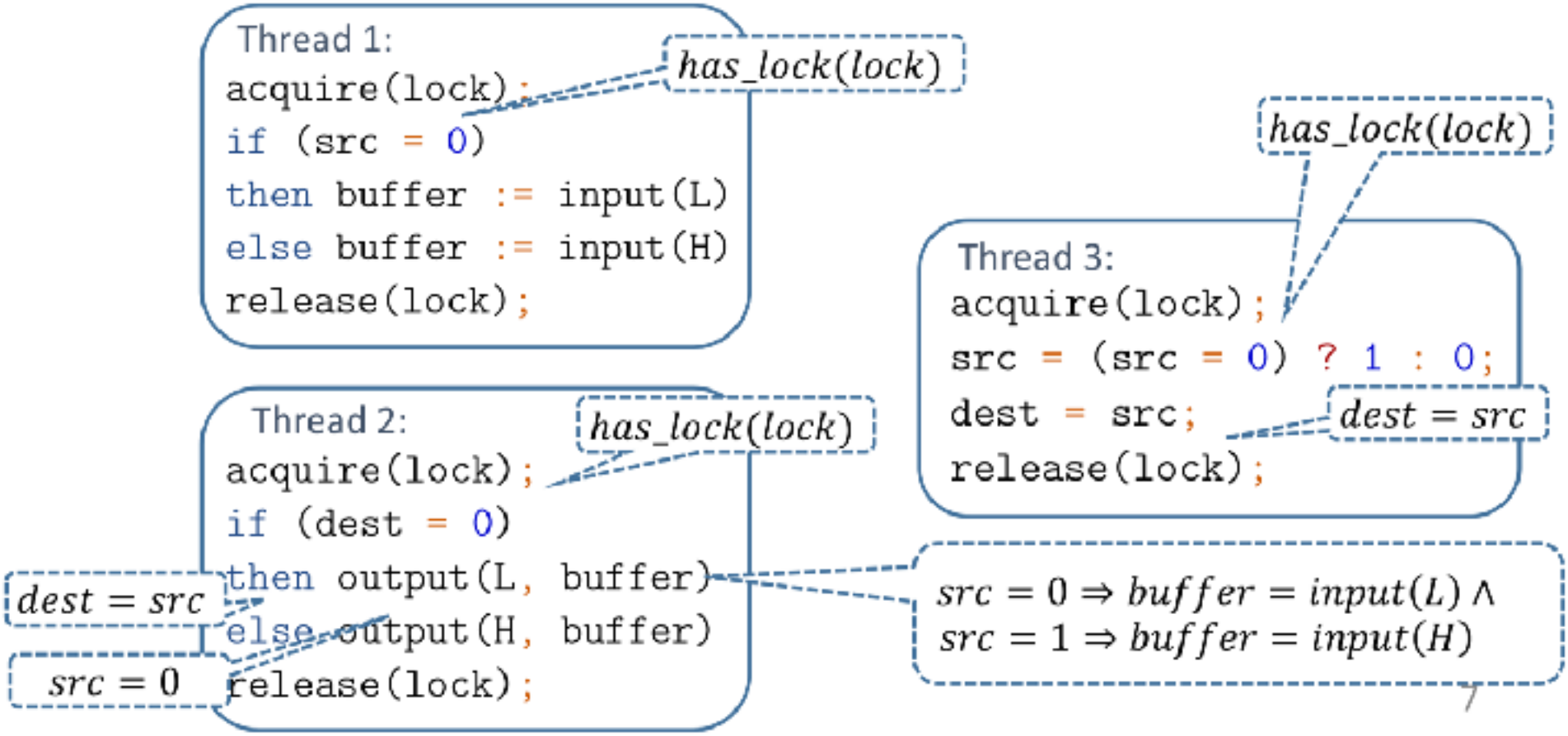
Annotations (1)



Annotations (2)



Annotations (3)



Annotated Programs

Commands c carry precondition annotations A

$$
\begin{aligned}
c \;::=\; & \textbf{skip} \\
\mid\; & c_1 \,;\, c_2 \\
\mid\; & \{A\}\; x := e \\
\mid\; & \{A\}\; x :=_D e \\
\mid\; & \{A\}\; \textbf{out}(l, e) \\
\mid\; & \{A\}\; x \leftarrow l \\
\mid\; & \{A\}\; \textbf{if}\ (e)\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \\
\mid\; & \{A\}\; \textbf{while}\ e\ \{I\}\ \textbf{do}\ c \\
\mid\; & \{A\}\; \textbf{acquire}(lck) \\
\mid\; & \{A\}\; \textbf{release}(lck)
\end{aligned}
$$

Type System for Annotated Programs

Intuition: Precise annotations allow dependency (data flow) information to be inferred

T-OUT

$$\frac{\forall s, s'.\ s =_\ell s' \wedge s \in A \wedge s' \in A \Rightarrow [\![e]\!](s) = [\![e]\!](s')}{\vdash \{A\}\ \mathbf{out}(\ell, e)}$$

T-DECL

$$\frac{\forall s.\ s \in A \Rightarrow \mathcal{D}([\![e]\!](s), s)}{\vdash \{A\}\ \mathbf{declassify}(e)}$$

Soundness Theorem

If every thread is well-typed,

local reasoning

and all annotations hold when the threads are run in parallel,

use your favourite concurrent program verification method

Then their parallel composition satisfies the security property

(so far we have used Owicki-Gries (Prensa Nieto '02) and Rely-Guarantee)

(generate annotations using your favourite program analysis methods, since generation doesn't need to be trusted anyway)

Type System Example

$$\vdash \textbf{if } (dest = 0) \textbf{ then out}(L, buffer) \textbf{ else out}(H, buffer)$$

Typing Rules

$$\textsc{T-Skip}\ \frac{}{\ell \vdash \textbf{skip}} \qquad \textsc{T-LAssign}\ \frac{x \in \mathrm{dom}(\mathcal{L}) \qquad \mathcal{L}(x) \sqsubseteq \mathcal{L}(e)}{\ell \vdash \{A\}\ x = e} \qquad \textsc{T-UAssign}\ \frac{x \notin \mathrm{dom}(\mathcal{L})}{\ell \vdash \{A\}\ x = e}$$

$$\textsc{T-Decl}\ \frac{\forall g.\ A(g) \Rightarrow \mathcal{D}(\mathcal{L}(e), \mathcal{L}(x), [\![e]\!](g), \{A\}\ x =_D e)}{\ell \vdash \{A\}\ x =_D e} \qquad \textsc{T-Out}\ \frac{A \models e}{\ell \vdash \{A\}\ \mathbf{out}(\ell', e)}$$

$$\textsc{T-LIn}\ \frac{x \in \mathrm{dom}(\mathcal{L}) \qquad \mathcal{L}(x) \sqsubseteq \ell'}{\ell \vdash \{A\}\ x \leftarrow \ell'} \qquad \textsc{T-UIn}\ \frac{x \notin \mathrm{dom}(\mathcal{L})}{\ell \vdash \{A\}\ x \leftarrow \ell'}$$

$$\textsc{T-If}\ \frac{A \models e \qquad \ell \vdash c_1 \qquad \ell \vdash c_2}{\ell \vdash \{A\}\ \textbf{if}\ (e)\ \textbf{then}\ c_1\ \textbf{else}\ c_2} \qquad \textsc{T-While}\ \frac{A \models e \qquad I \models e \qquad \ell \vdash c}{\ell \vdash \{A, I\}\ \textbf{while}\ e\ \textbf{do}\ c}$$

where:

$$A \models e = \forall m, m'.\, m =_\ell m' \wedge m \in A \wedge m' \in A \rightarrow [\![e]\!](m) = [\![e]\!](m')$$

c.f. Murray et al., EuroS&P 2018

$$\frac{\mathit{modifiable}_L(x) \quad \mathit{readable}_L(e) \quad \Gamma \vdash e : t \quad \mathit{stable}_L(t) \quad P' = \mathit{post}_L(x := e, P) \quad x \notin \mathit{NRW}(L) \longrightarrow t \leq_{P'} \mathcal{L}(x)}{\Gamma, L, P\ \{x := e\}\ \Gamma[x \mapsto t], L, P'}\ \textsc{Assign}_1$$

$$\frac{x \in C \quad \mathit{modifiable}_L(x) \quad \mathit{readable}_L(e) \quad \Gamma \vdash e : t \quad t \leq_P \mathsf{Low} \quad (\forall v \in \mathit{dom}(\Gamma).\ x \notin \mathit{vars}(\Gamma(v))) \quad P' = \mathit{post}_L(x := e, P) \quad \mathit{secure\_update}}{\Gamma, L, P\ \{x := e\}\ \Gamma', L, P'}\ \textsc{Assign}_C$$

$$\frac{\mathit{readable}_L(e) \quad \Gamma \vdash e : t \quad t \leq_P \mathsf{Low} \quad \Gamma, L, (P \wedge e)\ \{c_1\}\ \Gamma_1, L', P_1 \quad \Gamma, L, (P \wedge \neg e)\ \{c_2\}\ \Gamma_2, L', P_2 \quad \Gamma_1 =_{P_1} \Gamma' \quad \Gamma_2 =_{P_2} \Gamma' \quad P_1 \vdash P' \quad P_2 \vdash P' \quad (\Gamma_1, L', P_1) \leq (\Gamma', L', P') \quad (\Gamma_2, L', P_2) \leq (\Gamma', L', P')}{\Gamma, L, P\ \{\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2\}\ \Gamma', L', P'}\ \textsc{IfLow}$$

$$\frac{\mathit{readable}_L(e) \quad \Gamma \vdash e : t \quad t \leq_P \mathsf{Low} \quad \Gamma, L, (P \wedge e)\ \{c\}\ \Gamma, L, P}{\Gamma, L, P\ \{\textbf{while } e \textbf{ do } c\}\ \Gamma, L, P}\ \textsc{While}$$

$$\frac{\Gamma, L, P\ \{c_1\}\ \Gamma', L', P' \quad \Gamma', L', P'\ \{c_2\}\ \Gamma'', L'', P''}{\Gamma, L, P\ \{c_1 ; c_2\}\ \Gamma'', L'', P''}\ \textsc{Seq} \qquad \frac{}{\Gamma, L, P\ \{\textbf{skip}\}\ \Gamma, L, P}\ \textsc{Skip}$$

$$\frac{\Gamma' = \Gamma \ominus \ell}{\Gamma, L, P\ \{\textbf{lock}(\ell)\}\ \Gamma', L \cup \{\ell\}, (P \wedge \mathit{lockinv}(\ell))}\ \textsc{LockAcq}$$

$$\frac{\ell \in L \quad P \vdash \mathit{lockinv}(\ell) \quad \Gamma' = \Gamma \ominus \ell \quad P' = P \ominus \ell \quad \mathit{side\_condition}}{\Gamma, L, P\ \{\textbf{unlock}(\ell)\}\ \Gamma', L - \{\ell\}, P'}\ \textsc{LockRel}$$

Isabelle Formalisation

Theory	LoC
Language	1k
Security defs.	150
Type system & soundness	2k
Owicki-Gries	600
Rely-Guarantee	1.5k
Delimited Release	600
Misc	500
Total	**6.5k**

c.f. COVERN is 10K

More modular, extensible; integrates with existing O-G framework; easy to automate type checking via Eisbach

Future

- Extending to programs that branch on secrets
  - How far can we get with Hoare logic annotations here?
- Extending to weak memory
  - Feasible now that functional correctness has been decoupled
    - Since “morally” IFC type system doesn’t care about the order in which statements are executed
  - Use your favourite weak memory verification techniques…
- Larger case studies
  - e.g. redo the CDDC verification in the new framework

CONCLUSION

Decouple functional correctness and security for verifying IFC for Concurrent Programs

- supports compositional declassification
- inter-component contracts on shared memory:
  - expressed as Hoare logic assertions
- use your favourite concurrent program techniques
-

Still lots more to be done.

Thank You


toby.murray@unimelb.edu.au


http://people.eng.unimelb.edu.au/tobym


@tobycmurray

Connection to Delimited Release

- Delimited release [Sabelfeld & Myers 2003] ensures only information intended to be declassified can be leaked
- Can be soundly embedded into our model with side condition that *occurrence* of declassification events does not depend on secrets.
- Makes sense for programs that do not branch on secrets
- But unclear how to extend to programs that do branch on secrets