Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems

Formal Methods (FM), 7 March 2023

Robert Sison\textsuperscript{1,2}, Scott Buckley\textsuperscript{2}, Toby Murray\textsuperscript{1}, Gerwin Klein\textsuperscript{3,2}, and Gernot Heiser\textsuperscript{2}

\textsuperscript{1} The University of Melbourne, Australia
\textsuperscript{2} UNSW Sydney, Australia
\textsuperscript{3} Proofcraft, Sydney, Australia
Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems

Robert Sison\textsuperscript{1,2}, Scott Buckley\textsuperscript{2}, Toby Murray\textsuperscript{1}, Gerwin Klein\textsuperscript{3,2}, and Gernot Heiser\textsuperscript{2}

\textsuperscript{1} The University of Melbourne, Australia
\textsuperscript{2} UNSW Sydney, Australia
\textsuperscript{3} Proofcraft, Sydney, Australia

Formal Methods (FM), 7 March 2023

Paper: https://doi.org/jzwj
Artifact: https://doi.org/jzwk
Threat scenario: *Trojan* and *spy*
Threat scenario:
*Trojan* and *spy*
Threat scenario: Victim/Trojan and spy?

Covert channels + Side channels

Memory
Threat scenario:
Victim/Trojan and spy?

Covert channels + Side channels

Memory

A'

B

Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems | R. Sison, S. Buckley, T. Murray, G. Klein, G. Heiser
Threat scenario: 
*Trojan* and *spy*
Threat scenario: *Trojan and spy*

- OSes typically implement *memory protection.*
Threat scenario: \textit{Trojan} and \textit{spy}

- OSes typically implement memory protection.

- But: Mere memory access can change the microarch. state — this affects timing.
Threat scenario: *Trojan and spy*

- OSes typically implement *memory protection*.
- But: Mere memory access can change the microarch. state — this affects timing.
Threat scenario: *Trojan and spy*

- OSes typically implement *memory protection*.

- But: Mere memory access can change the microarch. state — this affects timing.
Threat scenario: *Trojan* and spy

- OSes typically implement *memory protection*.

- But: Mere memory access can change the microarch. state — this affects timing.
Threat scenario: 
**Trojan and spy**

- OSes typically implement *memory protection*.

- But: Mere memory access can change the microarch. state — this affects timing.
Threat scenario: *Trojan and spy*

- OSes typically implement *memory protection*.
- But: Mere memory access can change the microarch. state — this affects timing.
Threat scenario: Trojan and spy

- OSes typically implement memory protection.

- But: Mere memory access can change the microarchitectural state — this affects timing.
Threat scenario: *Trojan* and *spy*

- OSes typically implement *memory protection*.

- But: Mere memory access can change the microarch. state — this affects timing.

- To prevent these *timing channels*, OSes can implement *time protection*: e.g. [Ge et al. 2019] for seL4 microkernel OS
  - Partition what we can
  - Flush what we can’t
Threat scenario: **Trojan and spy**

- OSes typically implement *memory protection*.

- But: Mere memory access can change the microarch. state — this affects timing.

- To prevent these *timing channels*, OSes can implement *time protection*:
  - e.g. [Ge et al. 2019] for seL4 microkernel OS
    - Partition what we can
    - Flush what we can’t
Threat scenario: **Trojan and spy**

- OSes typically implement *memory protection*.

- But: Mere memory access can change the microarch. state — this affects timing.

- To prevent these *timing channels*, OSes can implement *time protection*:
  e.g. [Ge et al. 2019] for seL4 microkernel OS

  - Partition what we can

  - Flush what we can’t

“Flush”: Write fixed content; wait up to fixed time.
Threat scenario:
*Trojan* and *spy*

- OSes typically implement *memory protection*.
- But: *Mere memory access* can change the microarch. state — this affects timing.
- To prevent these *timing channels*, OSes can implement *time protection*: e.g. [Ge et al. 2019] for seL4 microkernel OS
  - Partition what we can
  - Flush what we can’t

“Flush”: Write fixed content; wait up to fixed time.
Threat scenario: *Trojan* and *spy*

- OSes typically implement *memory protection*.

- But: *Mere memory access* can change the microarchitectural state — this affects timing.

- To prevent these *timing channels*, OSes can implement *time protection*: e.g., [Ge et al. 2019] for seL4 microkernel OS

  - Partition what we can
  - Flush what we can’t

“Flush”: Write fixed content; wait up to fixed time.
Threat scenario: *Trojan and spy*

- OSes typically implement *memory protection*.

- But: *Mere memory access* can change the microarch. state — this affects timing.

- To prevent these *timing channels*, OSes can implement *time protection*:
  - e.g. [Ge et al. 2019] for seL4 microkernel OS

  - *Partition* what we can
  - *Flush* what we can’t

  “Flush”: Write fixed content; wait up to fixed time.
Threat scenario: *Trojan and spy*

- OSes typically implement *memory protection*.

- But: *Mere memory access* can change the microarch. state — this affects timing.

- To prevent these *timing channels*, OSes can implement *time protection*:
  - Partition what we can
  - Flush what we can’t
  e.g. [Ge et al. 2019] for seL4 microkernel OS

“Flush”: Write fixed content; wait up to fixed time.
How to formalise an OS enforces *time protection*?

Versus threat scenario:
trojan and spy
How to formalise an OS enforces *time protection*?

Abstract *covert state* + *time* to reflect strategies enabled by HW:
- Partition or flush state; pad time.

Versus threat scenario: trojan and spy

OS 👮‍♂️ HW

No channels!
How to formalise an OS enforces *time protection*?

Versus threat scenario: trojan and spy

Abstract *covert state* + *time* to reflect strategies enabled by HW:
- Partition or flush state; pad time.

Make security property *precise* enough to exclude flows from covert state.

No channels!
How to formalise an OS enforces *time protection*?

Versus threat scenario: trojan and spy

Abstract *covert state* + *time* to reflect strategies enabled by HW:
- Partition or *flush* state; *pad* time.

Make security property *precise* enough to exclude flows from covert state.

Demonstrating these principles, we formalised in Isabelle/HOL:
How to formalise an OS enforces *time protection*?

Versus threat scenario: trojan and spy

---

**Abstract covert state + time to reflect strategies enabled by HW:**
*Partition or flush state; pad time.*

---

**Make security property precise enough to exclude flows from covert state.**

---

Demonstrating these principles, we formalised in Isabelle/HOL:

1. OS security model imposing requirements on relevant parts of OS.
   (Intended for seL4, but *generic*)
How to formalise an OS enforces *time protection*?

Versus threat scenario: trojan and spy

Abstract covert state + time to reflect strategies enabled by HW:
Partition or flush state; pad time.

Make security property precise enough to exclude flows from covert state.

Demonstrating these principles, we formalised in Isabelle/HOL:

1. OS security model imposing requirements on relevant parts of OS.
   (Intended for seL4, but *generic*)

2. OS security property that is *dynamic*; this makes it observer relative.
   (Improving on seL4’s of [Murray et al. 2012])
How to formalise an OS enforces *time protection*?

Versus threat scenario: trojan and spy

Abstract *covert state* + *time* to reflect strategies enabled by HW:
- Partition or flush state; pad time.

Make security property *precise* enough to exclude flows from covert state.

Demonstrating these principles, we formalised in Isabelle/HOL:

1. OS security model imposing requirements on relevant parts of OS. (Intended for seL4, but *generic*)

2. OS security property that is *dynamic*; this makes it *observer relative*. (Improving on seL4’s of [Murray et al. 2012])

3. Proof our security property holds if OS model’s requirements hold.
How to formalise an OS enforces *time protection*?

Versus threat scenario: trojan and spy

Abstract *covert state* + *time* to reflect strategies enabled by HW:
- Partition or flush state; pad time.

Make security property precise enough to exclude flows from covert state.

Demonstrating these principles, we formalised in Isabelle/HOL:

1. OS security model imposing requirements on relevant parts of OS.
   (Intended for seL4, but *generic*).

2. OS security property that is *dynamic*; this makes it *observer relative*.
   (Improving on seL4’s of [Murray et al. 2012]).

3. Proof our security property holds if OS model’s requirements hold.

4. Basic instantiation of OS model exercising dynamic policy.
How to formalise an OS enforces *time protection*?

**Abstract covert state + time to reflect strategies enabled by HW:**
- Partition or flush state; pad time.

Make security property precise enough to exclude flows from covert state.

**Isabelle**

Demonstrating these principles, we formalised in Isabelle/HOL:

1. OS security model imposing requirements on relevant parts of OS. (Intended for seL4, but generic)
2. OS security property that is dynamic; this makes it observer relative. (Improving on seL4’s of [Murray et al. 2012])
3. Proof our security property holds if OS model’s requirements hold.
4. Basic instantiation of OS model exercising dynamic policy.

Versus threat scenario: trojan and spy

- OS interacts with HW
  - No channels!
Overt vs covert state

From prior seL4 infoflow proofs [Murray et al. 2012, 2013]:

“all or nothing” policies
Overt vs covert state

From prior seL4 infoflow proofs [Murray et al. 2012, 2013]: “all or nothing” policies
From prior seL4 infoflow proofs [Murray et al. 2012, 2013]: “all or nothing” policies

Principle: Need policies to allow some (overt) flows while excluding other (covert) ones
Covert state: Partitionable vs flushable

**Principle:**
Model channels as *state elements* by their *elimination strategy* as per *HW-SW contract*

\[ A \rightarrow B \]

- A's memory
- B's memory
- A's microarch. state
Covert state: Partitionable vs flushable

**Principle:**
Model channels as state elements by their elimination strategy as per HW-SW contract

- Strategy for OS:
  *Partition or flush state; pad time.*
Covert state: Partitionable vs flushable

**Principle:**
Model channels as *state elements* by their *elimination strategy* as per *HW-SW contract*

- Strategy for OS: *Partition or flush* state; *pad* time.
- Relies on HW-SW contract:
Covert state:
Partitionable vs flushable

**Principle:**
Model channels as *state elements* by their *elimination strategy* as per *HW-SW contract*

- Strategy for OS:
  *Partition or flush state; pad time.*

- Relies on HW-SW contract:
  - State: Everything must be *partitionable* or *flushable.*
Covert state: Partitionable vs flushable

**Principle:**
Model channels as *state elements* by their *elimination strategy* as per *HW-SW contract*

- Strategy for OS:
  *Partition or flush state; pad time.*

- Relies on HW-SW contract:
  - **State:** Everything must be *partitionable* or *flushable*.
    - e.g. Off-core vs on-core caches.
Covert state:
Partitionable vs flushable

**Principle:**
Model channels as *state elements* by their *elimination strategy* as per *HW-SW contract*

- Strategy for OS:
  *Partition or flush state; pad time.*

- Relies on HW-SW contract:
  - **State:** Everything must be *partitionable* or *flushable*.
    - e.g. Off-core vs on-core caches.
    - Interrupt-generating devices (partitionable; not pictured).
Covert state: Partitionable vs flushable

**Principle:**
Model channels as *state elements* by their *elimination strategy* as per *HW-SW contract*

- **Strategy for OS:**
  *Partition* or *flush* state; *pad* time.

- **Relies on HW-SW contract:**
  - **State:** Everything must be *partitionable* or *flushable*.
    - e.g. Off-core vs on-core caches.
    - Interrupt-generating devices (partitionable; not pictured).
  - **Time:** HW must give reliable
    - WCETs (worst-case execution times)
Covert state: Partitionable vs flushable

Principle:
Model channels as state elements by their elimination strategy as per HW-SW contract

- Strategy for OS:
  * Partition or flush state; pad time.

- Relies on HW-SW contract:
  - **State**: Everything must be *partitionable* or *flushable*.
    - e.g. Off-core vs on-core caches.
    - Interrupt-generating devices (partitionable; not pictured).
  - **Time**: HW must give reliable
    - WCETs (worst-case execution times)
    - method of *padding*.
How to formalise an OS enforces \textit{time protection}? 

Versus threat scenario: trojan and spy

\begin{center}
\begin{tikzpicture}
\fill [gray!30] (0,0) rectangle (6,6);
\fill [blue!30] (6,0) rectangle (12,6);
\fill [green!30] (12,0) rectangle (18,6);
\fill [red!30] (18,0) rectangle (24,6);
\end{tikzpicture}
\end{center}

Abstract \textit{covert state} + \textit{time} to reflect strategies enabled by HW: Partition or \textit{flush} state; \textit{pad} time.

Make security property \textit{precise} enough to exclude flows from \textit{covert state}.

Demonstrating these principles, we formalised in Isabelle/HOL:

1. OS security model imposing requirements on relevant parts of OS. 
   (Intended for seL4, but \textit{generic})

2. OS security property that is \textit{dynamic}; this makes it \textit{observer relative}. 
   (Improving on seL4’s of [Murray et al. 2012])

3. Proof our security property holds if OS model’s requirements hold.

4. Basic instantiation of OS model exercising dynamic policy.
How to formalise an OS enforces time protection?

Versus threat scenario: trojan and spy

OS 🤝 HW

Abstract covert state + time to reflect strategies enabled by HW:
Partition or flush state; pad time.

Make security property precise enough to exclude flows from covert state.

Demonstrating these principles, we formalised in Isabelle/HOL:

1. OS security model imposing requirements on relevant parts of OS.
   (Intended for seL4, but generic)

2. OS security property that is dynamic; this makes it observer relative.
   (Improving on seL4’s of [Murray et al. 2012])

3. Proof our security property holds if OS model’s requirements hold.

4. Basic instantiation of OS model exercising dynamic policy.
OS security model

Transition system

- OS entry
- OS exit
- User step
- OS step

Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems | R. Sison, S. Buckley, T. Murray, G. Klein, G. Heiser
OS security model

Transition system

User step

OS entry

OS exit

State fields

mem :: \( addr \rightarrow \text{int} \)
flst :: \( addr \rightarrow \text{bool} \) /* Flushable microarch. */
pst :: \( addr \rightarrow \text{bool} \) /* Partitionable microarch. */
tm :: \( \text{nat} \) /* Time */
dom :: \( \text{domain} \) /* Current domain */
devs :: \( \text{device set} \) /* Interrupt-generating devices */
event :: \{ \text{Syscall, UserInterrupt, TimerInterrupt} \}
args :: \( \text{args} \) /* System call arguments */
prot :: \( \text{prot} \) /* Protection state */
OS security model

Transition system

OS entry → User step → OS step → OS exit

State fields

- `mem :: addr ⇒ int`
- `flst :: addr ⇒ bool` /* Flushable microarch. */
- `pst :: addr ⇒ bool` /* Partitionable microarch. */
- `tm :: nat` /* Time */
- `dom :: domain` /* Current domain */
- `devs :: device set` /* Interrupt-generating devices */
- `event :: {Syscall, UserInterrupt, TimerInterrupt}`
- `args :: args` /* System call arguments */
- `prot :: prot` /* Protection state */
OS security model

Transition system

State fields

- `mem :: addr \rightarrow int`
- `flst :: addr \rightarrow bool /* Flushable microarch. */`
- `pst :: addr \rightarrow bool /* Partitionable microarch. */`
- `tm :: nat /* Time */`
- `dom :: domain /* Current domain */`
- `devs :: device set /* Interrupt-generating devices */`
- `event :: \{Syscall, UserInterrupt, TimerInterrupt\}
- `args :: args /* System call arguments */`
- `prot :: prot /* Protection state */`
Modelled to affect all flst + user’s pst, devs;

OS security model

Transition system

OS entry

User step

OS step

OS exit

State fields

mem :: addr $\rightarrow$ int
flst :: addr $\rightarrow$ bool /* Flushable microarch. */
pst :: addr $\rightarrow$ bool /* Partitionable microarch. */
tm :: nat /* Time */
dom :: domain /* Current domain */
devs :: device set /* Interrupt-generating devices */
event :: {Syscall, UserInterrupt, TimerInterrupt}
args :: args /* System call arguments */
prot :: prot /* Protection state */
OS security model

Transition system

State fields

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem</td>
<td>addr → int</td>
<td></td>
</tr>
<tr>
<td>flst</td>
<td>addr → bool</td>
<td>/* Flushable microarch. */</td>
</tr>
<tr>
<td>pst</td>
<td>addr → bool</td>
<td>/* Partitionable microarch. */</td>
</tr>
<tr>
<td>tm</td>
<td>nat</td>
<td>/* Time */</td>
</tr>
<tr>
<td>dom</td>
<td>domain</td>
<td>/* Current domain */</td>
</tr>
<tr>
<td>devs</td>
<td>device set</td>
<td>/* Interrupt-generating devices */</td>
</tr>
<tr>
<td>event</td>
<td>{Syscall, UserInterrupt, TimerInterrupt}</td>
<td></td>
</tr>
<tr>
<td>args</td>
<td>args</td>
<td>/* System call arguments */</td>
</tr>
<tr>
<td>prot</td>
<td>prot</td>
<td>/* Protection state */</td>
</tr>
</tbody>
</table>

Modelled to affect all flst and user's pst, devs; choose args; time advances OS security model.

No channels!

Microarchitecture, Devices, Policy-determining state

Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems | R. Sison, S. Buckley, T. Murray, G. Klein, G. Heiser
OS security model

Transition system

State fields

- mem :: addr ⇒ int
- flst :: addr ⇒ bool /* Flushable microarch. */
- pst :: addr ⇒ bool /* Partitionable microarch. */
- tm :: nat /* Time */
- dom :: domain /* Current domain */
- devs :: device set /* Interrupt-generating devices */
- event :: {Syscall, UserInterrupt, TimerInterrupt}
- args :: args /* System call arguments */
- prot :: prot /* Protection state */

Modelled to affect all flst + user's pst, devs; choose args; time advances

OS entry

User step

OS step

OS exit

Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems | R. Sison, S. Buckley, T. Murray, G. Klein, G. Heiser
OS security model

Transition system

OS entry

OS exit

User step

Modelled to affect all list + user’s pst, devs; choose args; time advances

OS step
OS security model

Transition system

Modelled to affect all list + user's pst, dev; choose args; time advances.

No channels!

Microarchitecture
Devices
Policy-determining state
Time
Modelled to affect all list + user's list, devs; choose args; time advances

Organisation System (OS) security model

Transition system

OS entry

User step

OS step

OS exit

Case 1:
Device interrupt

Where \( w_i \leq w_0 \)

Handle interrupt (WCET \( w_i \))

Architecture-specific

Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems | R. Sison, S. Buckley, T. Murray, G. Klein, G. Heiser

OS HW

Microarchitecture
Devices
Policy-determining state
Time

No channels!
OS security model

Transition system

Case 1: Device interrupt

- OS entry
- User step
- Modelled to affect all list + user’s pst, devs; choose args; time advances
- OS step
- Modelled as for user step
- OS exit
- Architecture-specific
- Where $w_i \leq w_0$

Case 2: System call

- OS entry
- User step
- Modelled to affect all list + user’s pst, devs; choose args; time advances
- OS step
- Modelled as for user step
- OS exit
- OS-specific (incl. infoflow policies)
- Where $w_d + w_c \leq w_0$

OS security model

- Decode (WCET $w_d$) (as for user step)
- Commit (WCET $w_c$) (as for user step)

No channels!
OS security model

Transition system

Case 1: Device interrupt
- OS entry
- User step
- OS step
- OS exit

Where \( w_i \leq w_0 \)

Case 2: System call
- OS entry
- User step
- OS step
- OS exit

Where \( w_d + w_c \leq w_0 \)

Decoded \((WCET w_d)\)

Case 3: Domain switch
- OS entry
- User step
- OS step
- OS exit

Timer interrupt delivered at (worst-case) \( T_0 + w_0 \)

- Partially flush \( pst \) \((WCET w_1)\)
- Flush \( flst \) \((WCET w_2)\)
- Change domain \((WCET w_3)\)
- Pad time until \( T_0 + w_0 + w_1 + w_2 + w_3 \)

Architecture-specific

OS-specific (incl. infoflow policies)

- Handle interrupt \((WCET w_i)\)
- Commit \((WCET w_c)\)

OS-specific

- Modelled as for user step
- Modelled to affect all \( flst + user’s \( pst, \) dev; choose args; time advances\)
OS security model

Transition system

Case 1: Device interrupt
- OS entry
- User step
- Modelled to affect all flst + user’s pst, devs, choose args; time advances
- OS step
- OS entry
- ... (as for user step)
- Modelled as for user step
- Handle interrupt (WCET $w_i$)
- Time advances
- Choose args
- OS exit
- Architecture-specific (incl. infoflow policies)
- Where $w_i \leq w_0$

Case 2: System call
- OS entry
- User step
- Modelled to affect all flst + user’s pst, devs, choose args; time advances
- OS step
- OS entry
- ... (as for user step)
- Modelled as for user step
- Handle interrupt (WCET $w_i$)
- Time advances
- Choose args
- OS exit
- Architecture-specific (incl. infoflow policies)
- Where $w_d + w_c \leq w_0$

Case 3: Domain switch
- OS entry
- User step
- Modelled to affect all flst + user’s pst, devs, choose args; time advances
- OS step
- OS entry
- ... (as for user step)
- Modelled as for user step
- Handle interrupt (WCET $w_i$)
- Time advances
- Choose args
- OS exit
- Architecture-specific (incl. infoflow policies)
- Where $w_i \leq w_0$
- Timer interrupt delivered at (worst-case) $T_0 + w_0$
- Partially flush pst (WCET $w_1$)
- Flush flst (WCET $w_2$)
- Change domain (WCET $w_3$)
- Pad time until $T_0 + w_0 + w_1 + w_2 + w_3$

Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems  |  R. Sison, S. Buckley, T. Murray, G. Klein, G. Heiser
OS security model

Transition system

**Case 1: Device interrupt**
- OS entry
- User step
- OS step
- OS exit
- Where $w_i \leq w_0$
- Modelled as for user step
- Handle interrupt (WCET $w_i$)

**Case 2: System call**
- OS entry
- User step
- OS step
- OS exit
- Where $w_d + w_c \leq w_0$
- Decode (WCET $w_d$
- Commit (WCET $w_c$

**Case 3: Domain switch**
- OS entry
- User step
- OS step
- OS exit
- Timer interrupt delivered at (worst-case) $T_0 + w_0$
- Partially flush $pst$ (WCET $w_1$
- Flush $fst$ (WCET $w_2$
- Change domain (WCET $w_3$
- Pad time until $T_0 + w_0 + w_1 + w_2 + w_3$

Architecture-specific

OS-specific (incl. infoflow policies)

Architecture-specific

Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems | R. Sison, S. Buckley, T. Murray, G. Klein, G. Heiser
OS security model

Transition system

Case 1: Device interrupt
- OS entry
- Modelled to affect all flst + user’s pst, devs; choose args; time advances
- User step
- OS entry
- OS step
- OS exit
- Architecture-specific
- Where \( w_i \leq w_0 \)

Case 2: System call
- OS entry
- Modelled as for user step
- Handle interrupt (WCET \( w_i \))
- Architecture-specific
- Where \( w_d + w_c \leq w_0 \)
- Decode (WCET \( w_d \))
- Commit (WCET \( w_c \))
- OS-specific (incl. infoflow policies)

Case 3: Domain switch
- OS entry
- Modelled as for user step
- Partially flush pst (WCET \( w_1 \))
- Flush flst (WCET \( w_2 \))
- Change domain (WCET \( w_3 \))
- Pad time until \( T_0 + w_0 + w_1 + w_2 + w_3 \)
- Architecture-specific
- Timer interrupt delivered at (worst-case) \( T_0 + w_0 \)
OS security model

Transition system

Case 1: Device interrupt
- OS entry
- Where $w_i \leq w_0$
- Handle interrupt (WCET $w_i$)
- Architecture-specific

Case 2: System call
- OS entry
- Where $w_d + w_c \leq w_0$
- Decode (WCET $w_d$)
- Commit (WCET $w_c$)
- OS-specific (incl. infoflow policies)

Case 3: Domain switch
- Timer interrupt delivered at (worst-case) $T_0 + w_0$
- Partially flush $pst$ (WCET $w_1$)
- Flush $fist$ (WCET $w_2$)
- Change domain (WCET $w_3$)
- Pad time until $T_0 + w_0 + w_1 + w_2 + w_3$
- Architecture-specific

Modelled to affect all $flst + user’s pst$, devs; choose args; time advances

OS entry

OS exit

User step

OS step

Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems
R. Sison, S. Buckley, T. Murray, G. Klein, G. Heiser
Security proof approach

Transition system

Case 1: Device interrupt
- OS entry
- Case 1: Device interrupt
  - Handle interrupt
    - (WCET $w_i$)
- OS exit
- User step
- OS step
- Modelled as for user step
- Where $w_i \leq w_0$
- Architecture-specific

Case 2: System call
- System call
  - Decode
    - (WCET $w_d$)
  - Commit
    - (WCET $w_c$)
- Where $w_d + w_c \leq w_0$
- OS-specific (incl. infoflow policies)

Case 3: Domain switch
- Domain switch
  - Timer interrupt delivered at (worst-case) $T_0 + w_0$
  - Partially flush $pst$ (WCET $w_1$)
  - Flush $flst$ (WCET $w_2$)
  - Change domain (WCET $w_3$)
  - Pad time until $T_0 + w_0 + w_1 + w_2 + w_3$
- Architecture-specific

Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems | R. Sison, S. Buckley, T. Murray, G. Klein, G. Heiser

No channels!
Security proof approach

Requirements
(In addition to WCETs)

**Transition system**

**Case 1:**
Device interrupt

- Where \( w_i \leq w_0 \)
- Handle interrupt (WCET \( w_i \))
- Architecture-specific

**Case 2:**
System call

- Where \( w_d + w_c \leq w_0 \)
- Decode (WCET \( w_d \))
- Commit (WCET \( w_c \))
- OS-specific (incl. infoflow policies)

**Case 3:**
Domain switch

- Timer interrupt delivered at (worst-case) \( T_0 + w_0 \)
- Partially flush \( pst \) (WCET \( w_1 \))
- Flush \( flst \) (WCET \( w_2 \))
- Change domain (WCET \( w_3 \))
- Pad time until \( T_0 + w_0 + w_1 + w_2 + w_3 \)
- Architecture-specific
Security proof approach

Transition system

OS entry

Case 1: Device interrupt

Where $w_i \leq w_0$

Handle interrupt (WCET $w_i$)

Confidentiality

Modelled to affect all flst + user’s pst, devs; choose args; time advances

OS step

Case 2: System call

Where $w_d + w_c \leq w_0$

Decode (WCET $w_d$)

Commit (WCET $w_c$)

Architecture-specific

OS-specific (incl. infoflow policies)

Case 3: Domain switch

Timer interrupt delivered at (worst-case) $T_0 + w_0$

Partially flush pst (WCET $w_1$)

Flush flst (WCET $w_2$)

Change domain (WCET $w_3$)

Pad time until $T_0 + w_0 + w_1 + w_2 + w_3$

Architecture-specific

Requirements
(In addition to WCETs)

Microarchitecture
Devices
Policy-determining state
Time

No channels!

Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems | R. Sison, S. Buckley, T. Murray, G. Klein, G. Heiser
Security proof approach

Transition system

**Case 1: Device interrupt**
- OS entry
- User step
- OS step
- OS exit

Where $w_i \leq w_0$

**Case 2: System call**
- Handle interrupt (WCET $w_i$)
- Confidentiality (relative to policy)
- Commit (WCET $w_c$)

Where $w_d + w_c \leq w_0$

**Case 3: Domain switch**
- Timer interrupt delivered at (worst-case) $T_0 + w_0$
- Partially flush $pst$ (WCET $w_1$)
- Flush $flst$ (WCET $w_2$)
- Change domain (WCET $w_3$)
- Pad time until $T_0 + w_0 + w_1 + w_2 + w_3$

Requirements (In addition to WCETs)

- Integrity
- Confidentiality
- Architecture-specific
- OS-specific (incl. infoflow policies)

Architecturespecific

Case 1: Device interrupt

OS entry

User step

OS step

OS exit

Case 2: System call

Case 3: Domain switch

Microarchitecture

Devices

Policy-determining state

Time

No channels!
Security proof approach

OS entry

User step

OS step

OS exit

Transition system

Case 1: Device interrupt

Where $w_i \leq w_0$

Handle interrupt (WCET $w_i$)

Confidentiality

Modelled to affect all flst + user’s pst, devs; choose args; time advances

Confidentiality

Modelled as for user step

Architecturespecific

Case 2: System call

Where $w_d + w_c \leq w_0$

Decide

(WCET $w_d$)

Integrity

Confidentiality

Relative to policy

OS-specific

(incl. infoflow policies)

Case 3: Domain switch

Timer interrupt delivered at (worst-case) $T_0 + w_0$

Flush flst (WCET $w_2$)

Correctness

Flush

(correct only)

Correctness

Change domain (WCET $w_3$)

Correctness

Pad time until

$T_0 + w_0 + w_1 + w_2 + w_3$

Architecture-specific

Requirements

(In addition to WCETs)

Microarchitecture

Devices

Policy-determining state

Time
Security proof approach

Transition system

Case 1: Device interrupt
- Where \( w_i \leq w_0 \)
- \( w_i \) ≤ \( w_0 \) (modelled as for user step)
- Handle interrupt (WCET \( w_i \))
- Confidentiality
- Time advances
- Choose args

Case 2: System call
- Where \( w_d + w_c \leq w_0 \)
- Decode (WCET \( w_d \))
- Integritly
- Correctness
- Commit (WCET \( w_c \))
- Confidentiality (relative to policy)

Case 3: Domain switch
- Timer interrupt delivered at (worst-case) \( T_0 + w_0 \)
- Partially flush \( pst \) (WCET \( w_1 \))
- \( w_d \) + \( w_c \) ≤ \( w_0 \)
- Flush \( flst \) (WCET \( w_2 \))
- Change domain (WCET \( w_3 \))
- Pad time until \( T_0 + w_0 + w_1 + w_2 + w_3 \)

We prove: Confidentiality property (bisimulation) step lemmas
Security proof approach

Transition system

Case 1: Device interrupt
- OS entry
- User step
- OS step
- OS exit

Where \( w_i \leq w_0 \)

Confidentiality
Modelled to affect all flst + user’s pst, devs; choose args; time advances

Architecture-specific

We prove: Confidentiality property (bisimulation) step lemmas

Case 2: System call
- OS entry
- User step
- OS step
- OS exit

Where \( w_d + w_c \leq w_0 \)

Confidentiality
Modelled as for user step

OS-specific (incl. infoflow policies)

Case 3: Domain switch
- OS entry
- User step
- OS step
- OS exit

Timer interrupt delivered at (worst-case) \( T_0 + w_0 \)

Confidentiality
(relative to policy)

Correctness

End

Pad time until \( T_0 + w_0 + w_1 + w_2 + w_3 \)

Correctness

Flush flst

Correctness

Partially flush pst

Correctness

Change domain

(RWCET \( w_1 \))

(RWCET \( w_2 \))

(RWCET \( w_3 \))
We prove: **Confidentiality property (bisimulation) step lemmas**
Security proof approach

Transition system

**Case 1:** Device interrupt
- Where \( w_i \leq w_0 \)
- Confidentiality
- Handle interrupt (WCET \( w_i \))

**Case 2:** System call
- Where \( w_d + w_c \leq w_0 \)
- Confidentiality (relative to policy)
- Commit (WCET \( w_c \))

**Case 3:** Domain switch
- Timer interrupt delivered at (worst-case) \( T_0 + w_0 \)
- Pad time until \( T_0 + w_0 + w_1 + w_2 + w_3 \)
- Partially flush \( pst \)
- Flush \( flst \)
- Change domain

**Requirements**
(In addition to WCETs)

- Integrity
- Correctness
- Correctness
- Correctness

**We prove:** Confidentiality property (bisimulation) step lemmas

Microarchitecture
- Devices
- Policy-determining state
- Time

Architecture-specific
OS-specific (incl. infoflow policies)

Architecture-specific

No channels!
How to formalise an OS enforces *time protection*?

Versus threat scenario: trojan and spy

Abstract *covert state* + *time* to reflect strategies enabled by HW:
- Partition or flush state; pad time.

Demonstrating these principles, we formalised in Isabelle/HOL:

1. OS security model imposing requirements on relevant parts of OS.
   (Intended for seL4, but *generic*)

2. OS security property that is *dynamic*;
   this makes it *observer relative*.
   (Improving on seL4’s of [Murray et al. 2012])

3. Proof our security property holds if OS model’s requirements hold.

4. Basic instantiation of OS model exercising dynamic policy.
How to formalise an OS enforces *time protection*?

Versus threat scenario: trojan and spy

Abstract *covert state* + *time* to reflect strategies enabled by HW:
Partition or *flush* state; *pad* time.

Demonstrating these principles, we formalised in Isabelle/HOL:

1. OS security model imposing requirements on relevant parts of OS.
   (Intended for seL4, but *generic*)

2. OS security property that is *dynamic*; this makes it *observer relative*.
   (Improving on seL4’s of [Murray et al. 2012])

3. Proof our security property holds if OS model’s requirements hold.

4. Basic instantiation of OS model exercising dynamic policy.
**OS security property**

Recall:

From prior seL4 infowflow proofs [Murray et al. 2012, 2013]:

“all or nothing” policies

For time protection, need spatial precision to allow some flows but exclude others
OS security property

Our infoflow policies:

For time protection, need *spatial precision* to allow some flows but exclude others.
Our infoflow policies:

- *Arbitrary* spatial precision

For time protection, need *spatial precision* to allow some flows but exclude others.
Our infoflow policies:

- *Arbitrary* spatial precision

- *Policy channels* specified as state relations: $s \sim_A^{t} B$

If $s \sim_A^{t} B$ equates part of A, then info flow is allowed from there to B.

For time protection, need *spatial precision* to allow some flows but exclude others.
OS security property

Our infoflow policies:

- **Arbitrary** spatial precision

- **Policy channels** specified as state relations: $\models_s^{\sim_t} A \sim B$

  If $\sim_t$ equates part of A, then info flow is allowed from there to B.

- Also arbitrary **temporal precision**

  For time protection, need **spatial precision to allow some flows but exclude others**
OS security property

Our infoflow policies:

• *Arbitrary* spatial precision

• *Policy channels* specified as *state relations*: \( s |A \sim B| t \)

  If \( s |A \sim B| t \) equates part of A, then info flow is allowed from there to B.

• Also arbitrary *temporal precision*

• The *dynamicity* gives us *observer-relative* properties

For time protection, need *spatial precision to allow some flows but exclude others*
Our infoflow policies:

- **Arbitrary** spatial precision

- **Policy channels** specified as state relations: $s |A \sim B| t$

  If $|A \sim B|$ equates part of A, then info flow is allowed from there to B.

- Also arbitrary **temporal precision**

- The **dynamicity** gives us **observer-relative** properties

For time protection, need **spatial precision to allow some flows but exclude others**
Dynamic policy, observer relativity
Dynamic policy, observer relativity

Two basic system calls: Subscribe(\(d\)), Broadcast()

\[
\begin{array}{c}
\text{A's memory} \\
\text{A's cache partition} \\
\text{Flushable caches} \\
\end{array} ~/> ~ \begin{array}{c}
\text{B's memory} \\
\text{B's cache partition} \\
\end{array}
\]
Dynamic policy, observer relativity

Two basic system calls:
\texttt{Subscribe(d)}, \texttt{Broadcast()}

1. \textbf{Dynamic policy}: \(A \sim \rightarrow B\) ?

✓ Only when \(A\) calls
  
  - \texttt{Subscribe(B)}, or
  
  - \texttt{Broadcast()} 1st time after \(B\) called \texttt{Subscribe(A)}.
Dynamic policy, observer relativity

Two basic system calls: \textbf{Subscribe}(d), \textbf{Broadcast}()

1. \textbf{Dynamic policy}: A \text{~\textleftarrow\textrightarrow~} B ?
   
   ✔️ Only when A calls
   • \textbf{Subscribe}(B), or
   • \textbf{Broadcast}() 1st time after B called \textbf{Subscribe}(A).

   ✗ Otherwise, no channel.
Dynamic policy, observer relativity

Two basic system calls: 
\texttt{Subscribe}(d), \texttt{Broadcast}()

1. Dynamic policy: A \rightarrow B ?
   - Only when A calls
     - \texttt{Subscribe}(B), or
     - \texttt{Broadcast}() 1st time after B called \texttt{Subscribe}(A).
   - Otherwise, no channel.

   - Example:
Dynamic policy, observer relativity

Two basic system calls:
\textbf{Subscribe}(\alpha), \textbf{Broadcast}()

1. **Dynamic policy**: A \(\rightarrow\) B ?
   
   ✔️ Only when A calls
   
   • \textbf{Subscribe}(B), or
   • \textbf{Broadcast}() 1st time after B called \textbf{Subscribe}(A).
   
   ❌ Otherwise, no channel.

• Example:

  1. A calls \textbf{Subscribe}(B)
Dynamic policy, observer relativity

Two basic system calls:
**Subscribe(\(d\)), Broadcast()**

1. **Dynamic policy**: A \(\rightarrow\) B ?
   
   - Only when A calls
     - **Subscribe(B)**, or
     - **Broadcast()** 1st time after B called **Subscribe(A)**.
   
   - Otherwise, no channel.

- **Example:**
  
  1. A calls **Subscribe(B)**
Dynamic policy, observer relativity

Two basic system calls: 
\textbf{Subscribe(\textit{d})}, \textbf{Broadcast()}

1. Dynamic policy: A ~\rightarrow{} B ?
   - Only when A calls  
     \begin{itemize}
     \item \textbf{Subscribe(\textit{B})}, or
     \item \textbf{Broadcast()} 1st time after B called \textbf{Subscribe(\textit{A})}.
     \end{itemize}
   - Otherwise, no channel.

- Example:
  1. A calls \textbf{Subscribe(\textit{B})}
  2. B calls \textbf{Broadcast()}

---

Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems  |  R. Sison, S. Buckley, T. Murray, G. Klein, G. Heiser
Dynamic policy, observer relativity

Two basic system calls:

Subscribe(\(d\)), Broadcast()

1. Dynamic policy: A \(\rightarrow\) B?
   - Only when A calls
     - Subscribe(B), or
     - Broadcast() 1st time after B called Subscribe(A).
   - Otherwise, no channel.

- Example:
  1. A calls Subscribe(B)
  2. B calls Broadcast()
Dynamic policy, observer relativity

Two basic system calls:
\texttt{Subscribe(d)}, \texttt{Broadcast()}

1. **Dynamic policy**: A \texttt{\sim/\succ} B ?
   - Only when A calls
     - \texttt{Subscribe(B)}, or
     - \texttt{Broadcast()} 1st time after B called \texttt{Subscribe(A)}.
   - Otherwise, no channel.

- **Example:**
  1. A calls \texttt{Subscribe(B)}
  2. B calls \texttt{Broadcast()}

Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems | R. Sison, S. Buckley, T. Murray, G. Klein, G. Heiser
Dynamic policy, observer relativity

Two basic system calls:
Subscribe(d), Broadcast()

1. Dynamic policy: A \rightarrow B ?
   - Only when A calls
     - Subscribe(B), or
     - Broadcast() 1st time after B called Subscribe(A).
   - Otherwise, no channel.

- Example:
  1. A calls Subscribe(B)
  2. B calls Broadcast()

2. Property must be observer relative!
Dynamic policy, observer relativity

Two basic system calls: 
\textbf{Subscribe}(d), \textbf{Broadcast()}

1. \textbf{Dynamic policy}: A \rightarrow B ?
   
   \checkmark Only when A calls
   
   \begin{itemize}
   \item \textbf{Subscribe}(B), or
   \item \textbf{Broadcast()} 1st time after B called \textbf{Subscribe}(A).
   \end{itemize}
   
   \xmark Otherwise, no channel.

2. Property must be \textbf{observer relative}!
   
   \begin{itemize}
   \item If not, can’t prove the (bisimulation) property for unrelated user C!
   \end{itemize}

\begin{itemize}
\item Example:
\begin{itemize}
\item 1. A calls \textbf{Subscribe}(B)
\item 2. B calls \textbf{Broadcast()}
\end{itemize}
\end{itemize}
Dynamic policy, observer relativity

Two basic system calls:
\textbf{Subscribe}(\textit{d}), \textbf{Broadcast}()

1. Dynamic policy: \textit{A} \rightarrow \textit{B} ?
   - \checkmark Only when \textit{A} calls
     - \textbf{Subscribe}(\textit{B}), or
     - \textbf{Broadcast}() 1st time after \textit{B} called \textbf{Subscribe}(\textit{A}).
   - \xmark Otherwise, no channel.

   • Example:
     1. \textit{A} calls \textbf{Subscribe}(\textit{B})
     2. \textit{B} calls \textbf{Broadcast}()

2. Property must be observer relative!
   - If not, can’t prove the (bisimulation) property for unrelated user \textit{C}!

As seen by \textit{C}

\textbf{A does call Subscribe}(\textit{B})

\textbf{A doesn’t call Subscribe}(\textit{B})

\textbf{As seen by C}

No such states

\textbf{A does call Subscribe}(\textit{B})

\textbf{A doesn’t call Subscribe}(\textit{B})

\textbf{As seen by C}

No such states
Dynamic policy, observer relativity

Two basic system calls: \texttt{Subscribe(d)}, \texttt{Broadcast()}

1. Dynamic policy: A $\rightleftharpoons$ B

   - Only when A calls
     - \texttt{Subscribe(B)}, or
     - \texttt{Broadcast()} 1st time after B called \texttt{Subscribe(A)}.

   - Otherwise, no channel.

- Example:
  1. A calls \texttt{Subscribe(B)}
  2. B calls \texttt{Broadcast()}

2. Property must be observer relative!

   - If not, can’t prove the (bisimulation) property for unrelated user C!

   - \textbf{Solution}: C’s property must treat states (in the state machine) as observable only whenever
     - C is running, or
     - When d is running, d $\rightleftharpoons$ C.
How to formalise an OS enforces *time protection*?

Versus threat scenario: trojan and spy

Abstract *covert state* + *time* to reflect strategies enabled by HW:
- Partition or *flush* state; *pad* time.

Make security property *precise* enough to exclude flows from covert state.

Demonstrating these principles, we formalised in Isabelle/HOL:

1. OS security model imposing requirements on relevant parts of OS.
   (Intended for seL4, but *generic*)

3. Proof our security property holds if OS model’s requirements hold.

2. OS security property that is *dynamic*; this makes it *observer relative*.
   (Improving on seL4’s of [Murray et al. 2012])

4. Basic instantiation of OS model exercising dynamic policy.
How to formalise an OS enforces *time protection*?

**Versus threat scenario:**
- trojan and spy

Abstract *covert state + time* to reflect strategies enabled by HW:
- Partition or *flush* state; *pad* time.

**No channels!**

Make security property *precise* enough to exclude flows from covert state.

**Demonstrating these principles,**
we formalised in Isabelle/HOL:

1. OS security model imposing requirements on relevant parts of OS.
   (Intended for seL4, but *generic*)

2. OS security property that is *dynamic*; this makes it *observer relative*.
   (Improving on seL4’s of [Murray et al. 2012])

3. Proof our security property holds if OS model’s requirements hold.

4. Basic instantiation of OS model exercising dynamic policy.

**Paper:** [https://doi.org/jzwj](https://doi.org/jzwj)
**Artifact:** [https://doi.org/jzwk](https://doi.org/jzwk)