PROSPER and Friends: An Overview

Mads Dam
KTH Royal Institute of Technology

Project team: Musard Balliu, Christoph Baumann, Victor Do, Christian Gehrmann, Roberto Guanciale, Jonas Haglund, Narges Khakpour, Andreas Lindner, Andreas Lundblad, Hamed Nemati, Oliver Schwarz, Arash Vahidi
The PROSPER Project

• Joint project KTH-SICS funded by Swedish Foundation for Strategic Research
• Start Jan 2012, ended Oct 2017
• Project objectives:
  – Build functional hypervisor for ARM-based systems
    • ... focus on security
  – Fully verified at system level
    • Hypervisor code
    • ... plus interaction with hardware platform
  – Support for GP OSs – RTOS, Linux, Android
    • ... plus some security services
PROSPER - Results

• Verified hypervisors:
  – Hypervisor v0 – simple separation kernel for ARMv7
  – Hypervisor v1 – memory virtualisation for ARMv7
  – Hypervisor v2, HASPOC – hypervisor for ARMv8
  – Increasing complexity and realism

• Main demonstrators:
  – Secure software update (ARMv7)
  – Secure network interface (ARMv7)
  – Red/black separation for Android (ARMv8, with Tutus AB)
  – . . .
Models and frameworks:
- Add-ons to Fox’s Cambridge HOL4/L3 models
- Compositional model framework
- Component models: MMUs, GICs, SMMUs, network devices ...
- Asynchronous device framework

Tools:
- ISA analyzers
- TreeDroid
- Info flow analysis tools EnCover (JVM) + others (binaries)
- HOL4 -> BAP lifter
Vulnerabilities and countermeasures:
  – Mismatched cache attributes
  – Countermeasures integrity, confidentiality

Systems:
  – Soft boot
  – Secure boot for ARMv8
  – Monotonic separation kernel

URLs:
  – prosper.sics.se
  – haspoc.sics.se
This Presentation

• Go through the three hypervisor generations one by one
• Explain:
  – Design rationale
  – Modelling and verification approach
  – Results
• Also discuss some of the related results:
  – ISA analyzer
  – Vulnerabilities, countermeasures, refinements
Separation Kernels

- Execution environments indistinguishable from a physically distributed system [Rushby’81]
... Or Hypervisors ...

- Execution environments indistinguishable from a physically distributed system [Rushby’81]
Provable Isolation – What Is Involved?

• Large endeavour
• Formal system model
  – Processor, devices, interrupt controllers, MMUs
  – Hypervisor, drivers, application code
  – Justification: Precision, adequacy
• Formalized security requirements
  – Security specification
  – Justification: Attack model
• Verification
  – Automated
  – Semi-automated
  – Interactive
Virtualization Target
PROSPER v0
Virtualization Target, v0, v1

- ARMv7 Processor
- MMU
- Network controller
- DMA controller
- Memory
PROSPER Kernel, v0
PROSPER Kernel, v0

• Context switch: Fixed round-robin scheduling
• Static memory allocation
• Asynchronous message passing through hypercall
• Paravirtualization

Dam, Guanciale, Khakpour, Nemati, Schwarz: Formal Verification of Information Flow Security for a Simple ARM-Based Separation Kernel, CCS’13
Verification Strategy

Approach 1: Noninterference

Confidentiality/nonexfiltration:
- No info flow from Guest$_1$ to Guest$_2$,...,Guest$_n$ or to Hypervisor

Integrity (kind of) similar
Verification Strategy

Approach 1: Vanilla noninterference

But:

• This was not the picture we wanted!
• What about communication?
Alternative Approach

- Formulate *ideal model*
- Satisfies isolation properties by construction

- Hypervisor functionality replaced by ideal functionality
- Ideal CPUs – run only user space code
- All privileged execution is idealized
- Two ideal message boxes
- Ideal timer for “activity toggling”
- Equivalence: Each guest “sees” the same observations
- When guest $G$ is active, the user mode observable parts of the ARMv7 machine state are identical
- $\Rightarrow$ Vanilla NI in the absence of communication
Unwinding Relation

Identical:
- MMU readable memory
- User mode observable registers
- Message boxes
- Time
Unwinding Relation

Weak bisimulation
- Per partition
- User mode observations to be preserved
- Weak (non-preemptive) handler transitions
- The relation? See the previous slide!
Boot Lemma

- Boot code terminates and establishes the relation
- Establish hypervisor invariant
- Machine code verification (HOL4 -> BAP)
User Lemma

- No infiltration/no exfiltration for user mode transitions, NI
- Independent of handler code, independent of guest code
- Theorem proving (HOL4)
Switch Lemma

- No infiltration/no exfiltration for exceptions/interrupts
- Independent of handler code, independent of guest code
- Theorem proving (HOL4)
Handler Lemmas

- Handlers satisfy their contracts
- Dependent on handler code, independent of guest code
- Machine code verification (HOL4 -> BAP)
# Verification Approach

<table>
<thead>
<tr>
<th>ARMv7 properties</th>
<th>Handler code</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Lemma</td>
<td>Handler Lemmas</td>
</tr>
<tr>
<td>Switch Lemma</td>
<td>Boot Lemma</td>
</tr>
<tr>
<td>Property of ARMv7</td>
<td>Code property</td>
</tr>
<tr>
<td>instruction set architecture</td>
<td>Frequently updated</td>
</tr>
<tr>
<td>HOL4 + Cambridge</td>
<td>C + assembly + gcc</td>
</tr>
<tr>
<td>ARMv7 model + L3 + MMU</td>
<td>BAP + STP</td>
</tr>
<tr>
<td>Noninterference lemmas</td>
<td>Contract verification</td>
</tr>
<tr>
<td>Automation: See later</td>
<td>“Semi”-automatic</td>
</tr>
</tbody>
</table>
PROSPER v1
PROSPER Kernel, v1

- Processor
- Memory Management Unit
- Memory
- Network controller
- DMA controller
MMU Virtualization

- MMU: Key component to virtualize commodity OSs
- L1 and L2 page tables
- Page tables map virtual addresses to intermediate addresses to physical addresses
- Control is vital
  - For virtualization
  - For sandboxing, etc.

Guanciale, Nemati, Dam, Baumann: Provably secure memory isolation for Linux on ARM, Journal of Computer Security 24(6), 2016
The Prosper v1 Hypervisor

• Primary use case:
  – Single untrusted OS guest
  – “Collaboratively” scheduled secure services
• Paravirtualization
• Memory management:
  – Direct paging, as in Xen-x86 or Secure Virtual Architecture\(^1\)
  – Page tables reside in guest memory
  – Guest can manipulate page tables when not in use
  – Hypervisor mediates access to page tables when active
  – Guest fully in charge of memory management

\(^1\): Criswell et al: Secure Virtual Architecture: A safe execution environment ... SOSP’07
The Prosper v1 Hypervisor

DMMU – the MMU virtualization API:
• Memory partitioned in physical blocks of 4 KB
• Blocks are typed: $t(block)$ in \{L1, L2, D\}
• 9 primitive API calls to activate, create or free page tables and to map or unmap memory blocks
• A reference counter keeps track of active references
• Hypervisor prevents unsound requests:
  – No access outside the guest memory
  – No writable access to a page table
• Block type can be changed if the reference counter is zero
Verification

Two stages:
1. Ideal model
   - Hypervisor state is idealized
   - Page tables stored in memory
   - Reference counter = 0 => page table can be freed
   - Hypervisor addresses physical memory
   - Correctness proof is needed
2. Implementation model
   - Algorithm + hypervisor state -&gt; hypervisor memory
   - Hypervisor addresses virtual memory
3. Refinement proof
   - Transfers info flow properties to implementation model
   - Bisimulation proof with some twists
Ideal Model Correctness Proof

Main components of proof:

• Invariant property maintained by the 9 API calls
  Needed for the below

• Complete mediation:
  Guest transitions cannot directly affect MMU behaviour

• Integrity:
  Guest transitions cannot affect hypervisor or secure guests state

• Confidentiality:
  No flow of information from hypervisor or secure guest state to insecure guest - noninterference
Implementation

Privileged components:
• Interface layer
• Linux adaptation layer
• DMMU handlers

Features:
• Small critical core
• No direct access to critical functionality from Linux layer
• Simpler to verify
PROSPER Kernel v1 - Applications

- Processor
- Memory Management Unit
- Memory
- Network controller
- DMA controller
MProsper: Executable Space Protection

- Memory blocks are executable or writeable, but not both
- Reference monitor intercepts memory attribute changes
- Pages are made executable only if they are duly signed

- Examples: OpenBSD 3.3, Linux PaX, Exec Shield, NetBSD, MS Oss with Data Execution Prevention

- Here: Using the Prosper kernel to implement this in a provably secure manner
- Monitor runs as isolated with read permissions - tamperproof
- Proof extends hypervisor security proof

Chfouka, Nemati, Guanciale, Dam, Ekdahl: Trustworthy Prevention of Code Injection in Linux on Embedded Devices, ESORICS’15
Enforce W$\mathcal{DX}$ policy

On Linux request to change access rights:

- Downgrade request
- Store suspended request in table

On data/prefetch abort:

- Downgrade and store current setting
- Re-enable suspended request, if safe
PROSPER Kernel, v1, Extensions

- Processor
- Memory Management Unit
- Network controller
- Memory
- DMA controller
Devices

Issues:
• Memory-mapped IO registers
• Interrupts
• DMA
• Asynchronous operation

Virtualization:
• Virtualized register accesses
• Static memory partitioning

Modeling:
• Interleaving of processor/device memory accesses using oracle

Schwarz, Dam: Formal Verification of Secure User Mode Device Execution with DMA, HVC’14
Status

Implementation:
- Ports for Linux 2.6.34 and Linux 3.10, BeagleBone, RPi 2
- Performance comparable to Xen
- Low memory overhead compared to shadow paging
- Experimental multicore port, one hypervisor per core

Models:
- ARMv7 model in L3 extended with MMU and system functionality
- Proven ISA level non-interference properties
- NIC + DMA models

Tools:
- HOL4 for model and design verification (refined-ideal bisimulation)
- Lifter from ARMv7 to BAP, partially verified in HOL4
- Binary code verification using SMT solver (STP)

Proofs:
- Guest switch lemma, verified hypervisor design
- Full verification v0, part binary verification v1,
- Proof for NIC virtualization in progress
PROSPER v2
Virtualization Target v2, HASPOC
Red-black architecture

- Untrusted: Android and Linux Kernel
- Middle guest is trusted:
  - negotiates parameters for VPN
  - encrypts all outgoing traffic
  - decrypts all incoming traffic

System objectives:
- VPN is enforced under all conditions
- VPN parameters never compromised
Minimal COTS hypervisor for ARMv8:

- Fixed #guests, static memory allocation
- Cores and devices owned exclusively
- No device virtualisation except GIC
- Secure boot loader
- Memory isolation through HW extensions and SMMUs

Main runtime hypervisor task is GIC virtualisation

Communication only through predefined channels
Tutus demonstrator

Red-black architecture

- Untrusted: Android and Linux Kernel
- Middle guest is trusted:
  - negotiates parameters for VPN

- guest 1 core i
- guest 2 core j

write → shared memory → read

hypercall → HV core i → software-generated interrupt → HV core j

VPN

virtualized interrupt
Security Goal

• Ideal model: Secure by construction
• Bisimulation relation transfers info flow properties
• Verification: Focus on guest (user mode) execution
Status

Implementation:
- HiKey board, <64KB code base <10K LoC, <2MB DRAM
- Demonstrators stable, <15% OH (interrupt penalties)
- Inter guest communication up to 750 Mbps
- Secure boot faster than ARM Trusted Firmware

Models:
- ARMv8 model in L3 extended with MMU and system features
- Compositional model for proof reusability and refinement
- Sequential memory, cache model under development

Tools:
- Lifter from ARMv8 to BAP, verified in HOL4
- Formal BAP Intermediate Language semantics in HOL4

Proofs:
- System level HOL4 proof of guest non-interference complete
- Pen-and-paper proof of design, Common Criteria compatible
- Verified weakest precondition generation (ongoing)
- Experiments in binary ARMv8 code verification
ISA Information Flow
Recall:

This is a property of the instruction set architecture!

Is it important?

- Yes, check Meltdown/Spectre

Could we have caught Meltdown/Spectre?

- Currently have caches in model, not speculation
- Given adequate model and enough cpu cycles, maybe

Schwarz, Dam: Automatic derivation of platform noninterference properties. SEFM 2016, 27-44
Wish to determine:

– What can a given user process determine of the processor state?

Dual problem:

– Which parts of the processor state can a user process (process at privilege level $x$) influence?
– Can be solved in similar manner
ISA Info Flow Analysis: The Problem

Input:
– Initial level assignment $I$

Output:
– Provably minimal final level assignment $F$ containing $I$

Objectives:
– Soundness, precision
– Apply to HOL4 ISA spec as is
– Implement in HOL4
– Fully automatic
– Test on realistic specs
getControl s =
let m := s.mode
in
let c :=
  (if m = user
     bitmask (s.ctrl m)
  else
     s.ctrl m
  )
in (c,s)
end
end

ISA Info Flow Analysis: Complications

Tricky to map into a standard type-based setting:
• Mappings need sometimes to be evaluated, sometimes not
• Levels need sometimes to be assigned bitwise, sometimes not
• Heavy context dependency
Rewriting

– Cambridge ISA specs are large so care is needed
– Use Fox’s ARM step library whenever possible

Instruction task queue:

– Rewrite to suitable normal form
– Attempt to prove NI
– Success, move on
– Failure:
  • Failure of proof search to imply counterexample
  • Use counterexample to refine low-equivalence relation
  • This gives minimality
  • Re-enqueue validated instructions
ISA Info Flow Analysis: Results

ARMv7-A user mode, no MMU, no security or hypervisor extensions

- Initial: PC
- Final included: User reg’s, full CPSR, some FP registers, TEEHBR, SCTLR flags EE, TE, V, A, U, DZ
- Not included: Banked registers, SPSRs, some FIQ-related registers, CP15.SCTLR.{NMFI,VE}
- Running time > 21 hrs on single Xeon X3470 core

MIPS-III

- Initial: PC + some basic registers, final: all, 1 hr+

MIPS-III restricted user mode

- Initial as above, final: GP registers + some status flags, 38’
Caches, caches, caches
Caches and Stuff

Current ISA modeling tends to ignore many nasty details
- Caches and cache management
- Speculation
- Lots of system features

How much of a problem is this?

Timing and power channels
- Very difficult to close completely
- Model-external features - abstract away (?)

Cache storage channels
- Deterministic channels not relying on timing/power
- Model internal - harder to ignore

Post Meltdown/Spectre: We’re in trouble (!)
Example: Memory Incoherence

Coherent memory:
  – Observers (cores, MMUs, etc) all see the same sequence of writes, per location

Controlled incoherence:
  – If one agent can be set up to control what another agent sees, we have a potential attack

Mismatched cacheability attributes
  – Virtual aliases with conflicting cacheability
  – Reasonable scenarios exist (e.g., virtualisation)
  – If cache and memory can disagree without entry becoming dirty there is a problem
  – This is sometimes the case
  – Integrity and confidentiality attacks

Verification

Need:
- More fine-grained model with caches
- New proof machinery
- Formalised countermeasures
- Not least: Redoing work already done . . .

Approach:
- Reuse verification on cacheless model
- Use proof obligations:
  - On processor model
  - On hypervisor
  - On countermeasures
  - On application
- General multilevel dcache+icache model
- Integrity proof done for two countermeasures
- Confidentiality in progress
Challenges
Precise Hardware Models

Modern hardware is complex
- Weakly-consistent memory
- Out-of-Order and speculation
- Cache hierarchies, MMUs, DMA bus masters, TLBs
- Rich flora of devices w. rapid churn
- How to keep up and scale?

Vendor-provided models
- Lack of documentation is a big issue
- See Alastair Reid’s presentation on ARM models
- Open source hardware, e.g. RISC-V?
- Hidden instructions? Vendor-specifics? HW Trojans?
- “Unpredictable behaviour”?

Generality and reusability
- vs. side channel protection/bisimulations
Managing Complexity

Building formal HW models is hard
- Huge informal specs
- Implementation-dependent behaviour
- Hard to test

Can we make it easier?
- Domain-specific languages can help
- Decomposed models for spec and proof reuse
  - Absolutely necessary for modern architectures
- Frameworks needed to mechanise proof search
  - HOL4 good starting point for this
- Executable models
  - Generality vs executability & speed
- Automating model construction
  - Check out Heule et al: Stratified synthesis: Automatically learning the x86-64 instruction set, PLDI’16
Thank you!
ARMv8 Platform Model

- Compositional model, async message passing
ARMv8 Platform Model

- Compositional model, async message passing
- (S)MMU: Active?, page table base, current translations
ARMv8 Platform Model

- Compositional model, async message passing
- (S)MMU: Active?, page table base, current translations
- Core: Execution mode, some hypervisor ext registers
ARMv8 Platform Model

- Compositional model, async message passing
- (S)MMU: Active?, page table base, current translations
- Core: Execution mode, some hypervisor ext registers
- Device: Mostly uninterpreted, DMA enabled?
ARMv8 Platform Model

- Compositional model, async message passing
- (S)MMU: Active?, page table base, current translations
- Core: Execution mode, some hypervisor ext registers
- Device: Mostly uninterpreted, DMA enabled?
- Memory: Flat map, memory-mapped IO
ARMv8 Platform Model

- Compositional model, async message passing
- (S)MMU: Active?, page table base, current translations
- Core: Execution mode, some hypervisor ext registers
- Device: Mostly uninterpreted, DMA enabled?
- Memory: Flat map, memory-mapped IO
- GIC: Hypervisor-accessed registers, interrupt state
ARMv8 Platform Model

- Compositional model, async message passing
- (S)MMU: Active?, page table base, current translations
- Core: Execution mode, some hypervisor ext registers
- Device: Mostly uninterpreted, DMA enabled?
- Memory: Flat map, memory-mapped IO
- GIC: Hypervisor-accessed registers, interrupt state
- Hypervisor: Fine-grained LTS, GIC interaction
Ideal Model

- Ideal core: HV invisible / atomic hypercall semantics
• Ideal core: HV invisible / atomic hypercall semantics
• Buffer for outgoing IGC notification interrupts
• Ideal core: HV invisible / atomic hypercall semantics
• Buffer for outgoing IGC notification interrupts
• IGC shared memory duplicated and copied on write
• Ideal core: HV invisible / atomic hypercall semantics
• Buffer for outgoing IGC notification interrupts
• IGC shared memory duplicated and copied on write
• Ideal GIC: interrupt separation by construction
Ideal Model

• Ideal core: HV invisible / atomic hypercall semantics
• Buffer for outgoing IGC notification interrupts
• IGC shared memory duplicated and copied on write
• Ideal GIC: interrupt separation by construction
• Message buffers as placeholders for (S)MMUs
• Ideal core: HV invisible / atomic hypercall semantics
• Buffer for outgoing IGC notification interrupts
• IGC shared memory duplicated and copied on write
• Ideal GIC: interrupt separation by construction
• Message buffers as placeholders for (S)MMUs
• Memory: only guest portion, intermediate physical addresses
Bisimulation Relation

Diagram showing the relationship between core, buffer, IGC buffer, GIC, memory, MMU, SMMU, and device, with interrupt and DMA connections.
Bisimulation Relation
Bisimulation Relation
Bisimulation Relation
Integrity Cache Incoherence Attack

V1: $D = \text{access}(VA_c)$
.
.
.
A1: write($VA_{nc}, 1$)
.
.
V2: $D = \text{access}(VA_c)$
V3: if not policy($D$) reject
.
.
[evict $VA_c$]
.
.
V4: use($VA_c$)
Integrity Cache Incoherence Attack

V1: $D = \text{access}(VA_c)$

... 

A1: write($VA_{nc}, 1$)

... 

V2: $D = \text{access}(VA_c)$

V3: if not policy($D$) reject

... 

[evict $VA_c$]

... 

V4: use($VA_c$)
Integrity Cache Incoherence Attack

V1: \( D = \text{access}(VA\_c) \)

... 

A1: \( \text{write}(VA\_nc, 1) \)

... 

V2: \( D = \text{access}(VA\_c) \)

V3: if not policy(D)

reject

... [evict \( VA\_c \)]

... 

V4: \( \text{use}(VA\_c) \)
Integrity Cache Incoherence Attack

V1: \( D = \text{access}(VA_c) \)
    
A1: \( \text{write}(VA_{nc}, 1) \)
    
V2: \( D = \text{access}(VA_c) \)
V3: if not policy(D) reject
    
    [evict VA_c]
    
V4: use(VA_c)

Diagram:

Virtual memory | Physical memory | Cache

VA_c

PA 1

PA 0

VA_nc

D 0
Integrity Cache Incoherence Attack

V1: \( D = \text{access}(\text{VA}_c) \)

\[ \ldots \]

A1: \( \text{write}(\text{VA}_\text{nc}, 1) \)

\[ \ldots \]

V2: \( D = \text{access}(\text{VA}_c) \)

V3: if not policy\((D)\)

\[ \text{reject} \]

\[ \ldots \]

[evict \(\text{VA}_c\)]

\[ \ldots \]

V4: \( \text{use}(\text{VA}_c) \)
Integrity Cache Incoherence Attack

V1: $D = \text{access}(VA_c)$

A1: $\text{write}(VA_{nc}, 1)$

V2: $D = \text{access}(VA_c)$

V3: if not policy($D$)
   reject

   [evict $VA_c$]

V4: $\text{use}(VA_c)$
Confidentiality Cache Incoherence Attack

A1: invalidate(VA_c)
A2: write(VA_nc, 0)
A3: D = read(VA_c)
A4: write(VA_nc, 1)
A5: call victim
A6: D = read(VA_c)
V1: if secr
   access(VA_3)
else
   access(VA_4)
Confidentiality Cache Incoherence Attack

A1: invalidate(VA_c)
A2: write(VA_nc, 0)
A3: D = read(VA_c)
A4: write(VA_nc, 1)
A5: call victim
A6: D = read(VA_c)

V1: if secr
    access(VA_3)
else
    access(VA_4)
Confidentiality Cache Incoherence Attack

A1: invalidate(VA_c)
A2: write(VA_nc, 0)
A3: D = read(VA_c)
A4: write(VA_nc, 1)
A5: call victim
A6: D = read(VA_c)

V1: if secr
   access(VA_3)
else
   access(VA_4)
Confidentiality Cache Incoherence Attack

A1: invalidate(VA_c)
A2: write(VA_nc, 0)
A3: D = read(VA_c)
A4: write(VA_nc, 1)
A5: call victim
A6: D = read(VA_c)

V1: if secr
    access(VA_3)
else
    access(VA_4)
Confidentiality Cache Incoherence Attack

A1: invalidate(VA_c)
A2: write(VA_nc, 0)
A3: D = read(VA_c)
A4: write(VA_nc, 1)
A5: call victim
A6: D = read(VA_c)

V1: if secr
    access(VA_3)
else
    access(VA_4)
Confidentiality Cache Incoherence Attack

A1: invalidate(VA_c)
A2: write(VA_nc, 0)
A3: D = read(VA_c)
A4: write(VA_nc, 1)
A5: call victim
A6: D = read(VA_c)

V1: if secr
   access(VA_3)
else
   access(VA_4)

Diagram:

Virtual memory

Physical memory

Cache

D | 0
VA_nc

VA_c | PA-1 | 1

VA_3 | PA-3

VA_4 | PA-4

secr | 0

PA-1 | 0

PA-4
Confidentiality Cache Incoherence Attack

A1: invalidate(VA_c)
A2: write(VA_nc, 0)
A3: D = read(VA_c)
A4: write(VA_nc, 1)
A5: call victim
A6: D = read(VA_c)

V1: if secr
    access(VA_3)
else
    access(VA_4)
Confidentiality Cache Incoherence Attack

A1: invalidate(VA_c)
A2: write(VA_nc, 0)
A3: D = read(VA_c)
A4: write(VA_nc, 1)
A5: call victim
A6: D = read(VA_c)

V1: if secr
    access(VA_3)
else
    access(VA_4)
Confidentiality Cache Incoherence Attack

A1: invalidate(VA_c)
A2: write(VA_nc, 0)
A3: D = read(VA_c)
A4: write(VA_nc, 1)
A5: call victim
A6: D = read(VA_c)

V1: if secr
  access(VA_3)
else
  access(VA_4)
Confidentiality Cache Incoherence Attack

A1: invalidate(VA_c)
A2: write(VA_nc, 0)
A3: D = read(VA_c)
A4: write(VA_nc, 1)
A5: call victim
A6: D = read(VA_c)

V1: if secr
   access(VA_3)
else
   access(VA_4)
Confidentiality Cache Incoherence Attack

A1: invalidate(VA_c)
A2: write(VA_nc, 0)
A3: D = read(VA_c)
A4: write(VA_nc, 1)
A5: call victim
A6: D = read(VA_c)

V1: if secr
   access(VA_3)
else
   access(VA_4)
Example Attacks

Three attacks implemented using mismatched cache attribute vector:

1. AES in Trustzone on RPi2
   128 bit key extracted after 850 encryptions
2. Prosper v1 on Beagleboard MX
   Attacker: Non-secure guest
   Validation of non-valid page table
   Attacker gets full control
3. Extraction of exponent from modular exponentiation procedure
   Non-pc secure procedure in Trustzone on RPi2
   Execution path detected through instruction cache attack
Countermeasures

For confidentiality:
- Standard timing approaches:
- PC-secure code, secret independent memory accesses, . . .

For integrity:
- Guarantee coherence of accessed memory
- Cache flushes, explicit eviction of cache lines, . . .

Specific for mismatched cache attributes:
- Secret independent cache line accesses
- Prevent uncacheable aliases for specific memory regions