Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor

Amit Vasudevan (CyLab-CMU), Sagar Chaki (SEI-CMU), Petros Maniatis (Google Inc.), Limin Jia, Anupam Datta (ECE/CSD-CMU)

http://uberspark.org
Problem

• Extensible Hypervisors raise significant security concerns
  • Number of bugs goes up with code size
  • Number of bugs goes up with frequency of updates
  • Number of bugs goes up with logical complexity
  • Number of bugs goes up with control-flow complexity

• Both complex VMMs and micro-hypervisors are prone to bugs
  • E.g., VMware [VMSA-2009-006, Cloudburst], Xen [CVE-2008-3687], SecVisor [Franklin et. Al, 2010]

• Verified hypervisor is accompanied by proof of desirable (security) properties
Why aren’t we already doing this?

- Cost of verification grows with
  - The size of the code-base
  - The number of separate components
  - The number of configurations
  - The rate of revisions

- Benefit of verification shrinks with
  - Steep learning curve of developer-unwieldy programming
  - Lack of commodity hardware integration
  - Magnitude of the runtime overhead

Compositionality

Commodity Compatibility

Performance
Why do this now?

- Formal C static analysis tools are very practical [Frama-C]
- Certifiable compilation tools [Compcert] are practical for moderate module sizes
- It’s trendy! [seL4, IronClad, IronFleet, FSCQ, mCertiKOS]
An extensible hypervisor

C + Assembly

Guest

Hyperdep

Sysclog

Aprvexec

Ropdet

Hypervisor

Mmu

Network

Vmx

Msr

Hardware

Challenge-1: Code size vs. HW de-privileging

- The diagram illustrates the impact of software components on performance.
- The Guest, Hypervisor, and Hardware sections show the de-privileging of system calls and control flow through hypercall emulation and privilege checking.

**Key Components**

- **Guest**
  - hyperdep
  - aprvexec

- **Hypervisor**
  - sysclog
  - ropdet
  - MMU
  - Network
  - VMX
  - MSR

- **Performance**
  - Decrease in code size as de-privileging occurs.
Challenge-2: Continuous Development

- Intro.
- Motivating Example
- Arch.
- Impl.
- Verif. Results
- Perf.
- Concl.

Diagram:
- Guest
- Hypervisor
- MMU
- VMX
- Network
- MSRs
- hyperdep
- aprvexec
- ropdet
- sysclog
Challenge-3: Shared Resources
Challenge-4: Different Configurations
Challenge-5: Verification vs. Programming Paradigm

- Programming Paradigm
  - C + Assembly is de-facto
  - C + Assembly can clobber stuff! [stack, registers, MSRs etc.]
  - HW access and ops. with multi-core

- State-of-the-art Verification Tools
  - Often impose use of “developer-unwieldy” high-level languages with steep learning curve [Coq, Haskell, Dafny]
  - Largely lack support for Assembly
  - Mainly target sequential code
  - Largely lack support for HW integration
• **Goals**
  - Compositionality
  - Commodity Compatibility
  - Performance

• **Verifiable Object Abstraction (uberObject)**
  - Security invariants
  - Commodity HW + Software Verification

---

**ÜberSpark from above**

**ÜberObjects**

[C + Assembly + ACSL]

**ÜBlueprint**

**Proofs**

- HW
- HW + SW-Verif
- SW-Verif

---

**System Resources**

[CPU (Privileged) Instructions, Memory, Device Interfaces]
The überObject

- Use Manifest + Behavior Specifications in C-like language
- System Resources [CPU (Privileged) Instructions, Memory, Device Interfaces]
überObject: Sentinel

- Sentinel
  - Establishes “call-ret” semantics
  - Object to object control-flow enforcer
- überObjects verified not to write on other stack frames
- Enables sound application of sequential source code verification to verify invariants over sequential überobject invocations
Abstract hypervisor as a non-deterministic sequential program ➔
prove invariant properties of individual objects and compose them
überObject: CASM Functions & HW Model

• CASM Functions
  • C functions composed solely of Assembly
  • (Any) Assembly instruction as macro

• HW model specifies semantics

• Custom Frama-C verification plugins
  • Inline C99 semantics to verify
  • Inline Assembly to compile down

```c
void gp_setup_vhmempgtbl1(void){
  u32 i, spatype, slabid=XMHF_SLAB_PRIME;
  u64 flags; ...
  for(i=0; i < (SZ_PDPT*SZ_PDT*SZ_PT); ++i){
    spatype = gp_getspatype(slabid, (u32)(i*SZB_4K));
    flags = gp_getptflags(slabid, (u32)(i*SZB_4K), spatype);
    vhpgtbl1t[i] = pae_make_pte((i*SZB_4K),flags);
  }
  ...
  casm_writecr3(vhsmpgtbl4t[0]);
}
```

```casm
void casm_writecr3(u32 value){
  ci_movl_mesp_eax(0x4);
  ci_movl_eax_cr3();
  ci_ret();
}
```

CASM Instructions
uberObject: Coding and Behavior Specification

- C99 + CASM (principled Assembly)
- ANSI C Specification Language (ACSL)
  - requires/assigns/ensures
- Hoare triple proven automatically via Frama-C
  - deductive verification plugins
  - ensemble of SMT solvers

//@ghost u64 gflags[SZ_PDPT*SZ_PDT*SZ_PT];
/*@ ... requires \valid(vhpgtbl1t[0..(SZ_PDPT*SZ_PDT*SZ_PT)-1]); ...
assigns vhpgtbl1t[0..(SZ_PDPT*SZ_PDT*SZ_PT)-1]; ... 
ensures (\forall all u32 x; 0 <= x < SZ_PDPT*SZ_PDT*SZ_PT =>
  ((u64)vhpgtbl1t[x] == (((u64)(x*SZB_4K)
  & 0x7FFFFFFFFFFFF000ULL) | (u64)(gflags[x])))); @*/

void gp_setup_vhmempgtbl(void){
  u32 i, spatype, slabid=XMHF_SLAB_PRIME;
  u64 flags; ... 
 //@ loop invariant 0 <= i <= (SZ_PDPT*SZ_PDT*SZ_PT); ... @*/
  for(i=0; I < (SZ_PDPT*SZ_PDT*SZ_PT); ++i){
    spatype=_gp_getspatype(slabid, (u32)(i*SZB_4K));
    flags=_gp_getptflags(slabid, (u32)(i*SZB_4K),spatype);
    //@ghost gflags[i] = flags;
    vhpgtbl1t[i] = pae_make_pte((i*SZB_4K),flags);
    //@assert vhpgtbl1t[i] == (((u64)(i*SZB_4K)
    & 0x7FFFFFFFFFFFF000ULL) | (u64)(gflags[i]))); @*/
  }
  casm_writercr3(vhsmptbl4t[0]);
}
uberObject: Resource Interface Confinement

- uberAPI uberobjects
  - Wrap a reference monitor around (shared) resource
  - MMU, IOMMU, CRs, MSRs, Devices

- Client object manifests how it will use a (shared) resource
  - Verified on client via assertions

- During integration
  - Use manifests combined into one formula
  - SMT solvers check composability
überObject: Summary

• C99 + CASM + ACSL behavior specifications and behavior restrictions
• Object invariants including basic memory safety and control-flow integrity and other properties that can be formulated as invariants
• Architecture ensures invariant composition

• Mind-Blow #1: Only need to worry about object behavior now – not implementation
• Mind-Blow #2: A compositionally verifiable C + Assembly system without hardware de-privilegimg
An über Micro-Hypervisor (ÜXMHF)

- XMHF micro-hypervisor (http://xmhf.org)
  - Core hypervisor + single extension (hypapp)
  - Ubuntu 12.04 32-bit SMP on Intel VT-x/AMD
  - Various hypapps
    - tracing, attestation, app-level integrity, trusted path etc.

- ÜXMHF
  - Multiple extensions
  - Ubuntu 12.04 32-bit SMP on Intel VT-x
  - 11 überobjects, 7001 SLoC including prime and sentinel
  - Took ~3 person months for refactoring
ÜXMHF Verification Results

• Verification Tools TCB
  • Frama-C, uberSpark Plugins (1021 SLoC), SMT Solvers (Z3, CVC3, Alt-ergo), HW Model (2079 SLoC)

• Security Invariants in core Hypervisor and Extensions
  • memory-safety, control-flow integrity, no direct writes to hypervisor memory by guest, DEP, guest syscalls n/w logging etc.

• Verification Metrics
  • 11 überobjects, 5544 SLoC total ACSL annotations
  • Annotation to code ratio 0.2:1 to 1.6:1
  • überobject verification times from 48s to 23 min; cumulative ~1hr
  • Took ~9 person months
ÜXMHF: Micro & Application Benchmarks

- Sentinel transfer cost

<table>
<thead>
<tr>
<th>Verified-Verified</th>
<th>Verified-Unverified / Unverified-Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SEG</td>
</tr>
<tr>
<td>2x</td>
<td>37x</td>
</tr>
</tbody>
</table>

- ÜXMHF vs. vanilla XMHF
  - Verified hypapps (2% avg. overhead)
  - Unverified hypapps (10% avg. overhead)
  - I/O and normal Guest performance unaffected!
So, what do we have here?

- Can prove behavior one object at a time (trace properties)
- Can compose modules and behaviors cheaply
- Can write system code in “basically” C and Assembly and behavior specifications in C-like specification language
- Can integrate HW accesses and states into verification
- Can execute with good runtime performance
So, what don’t we have, yet?

- Not “exactly” C99 + Assembly; no cowboy control flow craziness
  - God forbid no C++
- Compcert + CASM proofs
  - Semantic compatibility between Frama-C, Compcert and CASM
- HW Model to Assembly instructions refinement
- Full functional correctness
- Concurrent verification
- Broader applicability
  - Other hypervisors (Xen, KVM), BIOS, Device firmware, OS Kernel and Drivers, User-space Applications and Browser Extensions
Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor

Questions?

http://uberspark.org

Amit Vasudevan
(amitvasudevan@acm.org)