Proving the security of SVSM code via formal verification

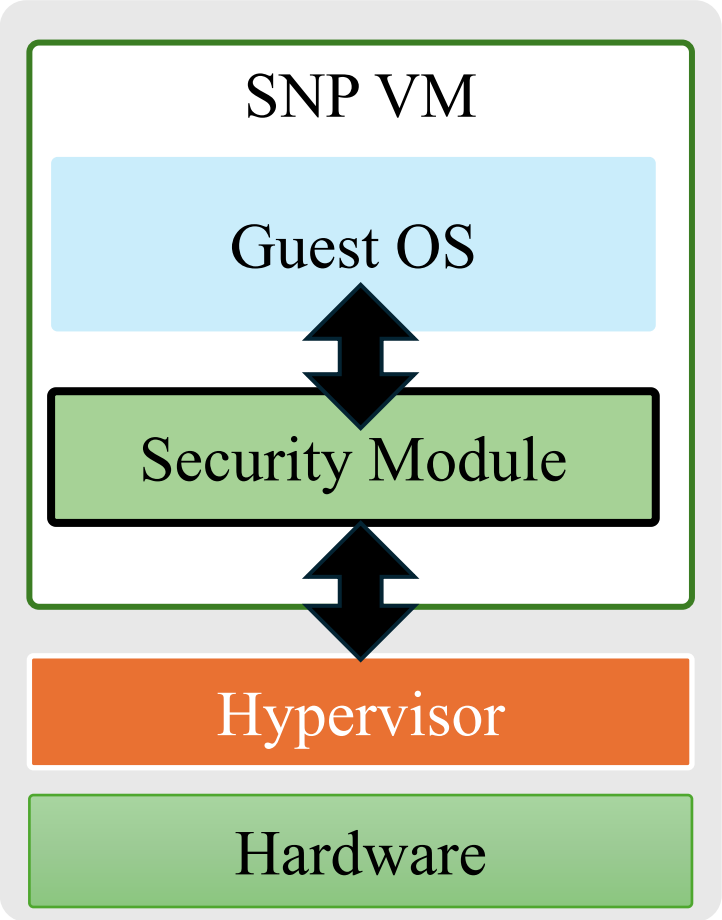
Ziqiao Zhou, Chris Hawblitzel, Weidong Cui

Microsoft Research

Outline

- Security module is not secure.
- Verus provides Rust-based formal verification
- Incremental verification allows co-existence of verified and unverified code
- VeriSMo: verifying complex security properties
  - https://github.com/microsoft/verismo

VeriSMo: A verified security module for confidential VMs

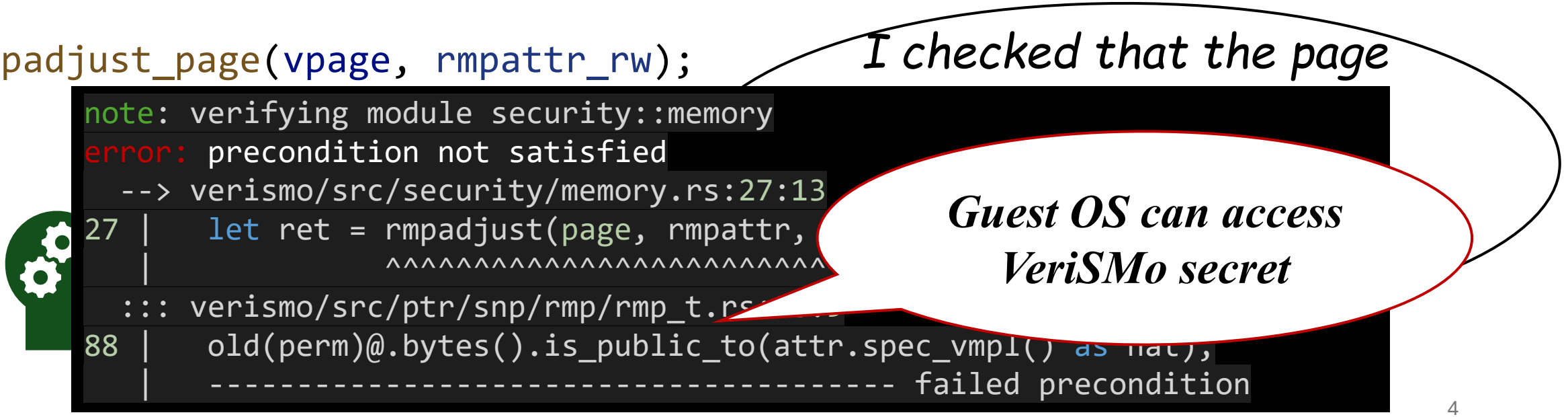


A lightweight VM firmware at highest privilege level providing APIs to the guest OS

- Replace hypervisor-based security features
  - ✓ Code integrity protection
  - ✓ vTPM (PCR extension, attestation, etc.)
- Manage security-sensitive CPU and memory changes
  - ✓ Setup CPU contexts for Guest OS
  - ✓ Manage SNP guest memory

A bug in security module

```
fn mk_guest_priv(vpage: usize) -> bool
{
  // Reject if the page is not guest page and was not shared.
  if !is_guest_os_page(vpage) || !is_shared(vpage) {return false;}
  …
  validate_page(vpage, true);

  rmpadjust_page(vpage, rmpattr_rw);
  …
}
```



A bug in security module

```
fn mk_guest_priv(page: usize, Tracked(mperm): Tracked<&mut MemPerm<T>>) ->
bool
{
  // Reject if the page is not guest page or is not shared.
  if !is_guest_page(page) || !is_shared(vpage) {return false;}
  …
  validate_page(page, true, Tracked(mperm));
  memset(page, 0, Tracked(mperm));
  rmpadjust_page(page, rmpattr_rw, Tracked(mperm));
  …
}
```

```
note: verifying module security::memory
note: verification results: 1 verified, 0 errors
```

Fix race conditions in SVSM core protocol handling #320

Merged joergroedel merged 6 commits into `coconut-svsm:main` from `joergroedel:sec-fixes` on Apr 24

Conversation 10 | Commits 6 | Checks 3 | Files changed 1

AMDESE / linu

Code | Issue

joergroedel commented on Apr 12 — Member

There are a couple of race conditions in how the SVSM core protocol handling is implemented. These issues can be used by an attacker to leak data out of the SVSM or escalate privileges of lower VMPLs.

Commit

protocols/core: Clear page after PVALIDATE of the page

```
A malicious hypervisor can attempt to reveal data from the SVSM to lower
VMPL levels through RMP manipulation related to page validation. For
example:

  - Initially, VMPL0 has a page at GPA A which maps to SPA X
  - VMPL3 asks HV to change the state of GPA B to private
  - HV maliciously reclaims SPA X and changes the RMP entry (and NPT) to
    map it at GPA B
  - VMPL3 asks VMPL0 to validate a new page at GPA B
  - VMPL0 PVALIDATE/RMPADJUSTs GPA B, allowing VMPL3 to read the data that
    VMPL0 had previously stored at GPA A

To prevent the exposure of any data in that page, the SVSM must zero-out
the memory after the PVALIDATE but before the RMPADJUST that grants
permission to the lower VMPL levels.

Signed-off-by: Tom Lendacky <thomas.lendacky@amd.com>
```

main

tlendacky committed on Jun 19, 2023

ing guest access

```
Attack scenario:
  1) SVSM stores secrets in VMPL0 memory at GPA A
  2) HV invalidates GPA A and maps the SPA to GPA B, which is in the
     OS range of GPAs
  3) Guest OS asks SVSM to validate GPA B
  4) SVSM validates page and gives OS access
  5) OS can now read SVSM secrets from GPA B

The SVSM will not notice the attack until it tries to access GPA A
again. Prevent it by clearing every page before giving access to
other VMPLs.

Reported-by: Tom Lendacky <thomas.lendacky@amd.com>
Signed-off-by: Joerg Roedel <jroedel@suse.de>
```

main (#45)

Verification vs. testing and fuzzing

Unit Testing

Tests ***specific*** executions

Fuzzing

Dynamic test multiple executions for unexpected behavior/crash

Verification

Theorem prover statically proves properties of ***all*** possible executions

Verus: verifying the correctness of Rust code

https://verus-lang.github.io/verus/guide/overview.html

- **A tool developed by Microsoft, CMU, and VMware, etc.**
- **Function-level verification**
  - ✓NOT a push-button verification for the whole program
  - ✓Support incremental verification
- **Optimized performance**
  - ✓Utilizes the SMT solver more efficiently.
  - ✓Proof engineering can further improve verification performance

Projects using Verus for formal verification

- **Microsoft**
  - Verismo: A verified security module for AMD confidential VM
    - https://github.com/microsoft/verismo
  - Azure storage crash-consistent log
  - LLMs for C-to-Rust

- **Non-Microsoft**
  - Concurrent memory allocator (CMU)
  - NrOS (ETH-Zurich, UBC-Vancouver)
    - https://github.com/utaal/verified-nrkernel
  - Atmosphere microkernel (U-Utah)
    - https://mars-research.github.io/projects/atmo/
  - Vest: Verified Parser and Serializer
    - https://github.com/secure-foundations/vest
  - Anvil Cluster Management *(U-Illinois, U-Wisconsin, VMware)*
    - https://github.com/anvil-verifier/anvil

How to use verification with Verus

```
use builtin_macros::*;
include!("address.verus.rs");

…
#[verus_verify]
pub struct VirtAddr(InnerAddr);

#[verus_verify]
impl VirtAddr {

    #[verus_verify]
    #[requires(offset <= 0x7FFF_FFFF_FFFF, …)]
     pub const fn add(&self, offset: usize) -> Self
    {
        VirtAddr::new(self.0 + offset)
    }
}
```

- ✓ Import Verus library
- ✓ Define the invariant that the virtual address must be in canonical format
- ✓ Set items visible to verifier
- ✓ Annotate a function to verify

```
~/svsm/kernel$ cargo verify
note: verifying module address
note: done with module address
verification results:: 1 verified, 0 errors
verification results:: 0 verified, 0 errors
   Finished dev target(s) in 3.5 s
```

Verification with light annotations can help

```
let a = 0xff_0000_0000_0000u64 as *const usize;
let _ = VirtAddr::from(a);
```

```
#[verus_verify]
impl<T> From<*const T> for VirtAddr {
    #[inline]
    #[verus_verify]
    fn from(ptr: *const T) -> Self {
    ❌  Self(ptr as InnerAddr)
    }
}
```

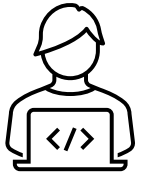
```
~/svsm/kernel$ cargo verify
note: verifying module address
error: constructed value may fail to meet its declared type invariant
  --> kernel/src/address.rs:392:9
   |
392 |       Self(ptr as InnerAddr)
   |       ^^^^^^^^^^^^^^^^^^^^^^
   |
  ::: kernel/src/address.verus.rs:134:5
   |
134 |    pub closed spec fn is_canonical_vaddr(&self) -> bool {
   |    ------------------------------------------------- type invariant declared here
note: diagnostics via expansion:
    tmp%.is_canonical_vaddr()
    |  !(tmp%.view() & !clip(140737488355327) == 0) ==>
    |     tmp%.view() & !clip(140737488355327) == !clip(140737488355327)
    +---
note: done with module address
verification results:: 44 verified, 1 errors
```

Uninterrupted code collaboration via incremental verification

- Cooperation Between Verification and Non-Verification Developers

✓ Write executable codes
✓ CI uses cargo verify –no-verify to check syntax **without** verifying it.

Run **cargo build** **to compile code for dev**



Non-verification developers

✓ Add Annotation in Executable code
✓ Write Specification + Proof in verus! macro in Rust
✓ CI on verification branch uses cargo verify

Run **cargo verify** **to trigger verus to verify the code**

Verification developer

Incremental Verification on
a function without verus annotations

```
impl VirtAddr {

fn add(&self, offset: usize) -> Self
{
    VirtAddr::new(self.0 + offset)
}

}
```

Verifier: Nothing to verify

```
~/svsm/kernel$ cargo build
Compiling: svsm v0.1.0
Finished `dev` profile target(s) in 1.25s
```

```
~/svsm/kernel$ cargo verify
Compiling: svsm v0.1.0
verification results:: 0 verified, 0 errors
verification results:: 0 verified, 0 errors
Finished `dev` profile target(s) in 1.26s
```

Incremental Verification on
a function that is set to verify

```
impl VirtAddr {

 #[verus_verify]
fn add(&self, offset: usize) -> Self
{
  ❌VirtAddr::new(self.0 + offset)
}

}
```

Verifier: Verifying add **function**

1. Basic underflow/overflow checking
2. Output satisfying VirtAddr specification

```
~/svsm/kernel$ cargo build
Compiling: svsm v0.1.0
Finished `dev` profile target(s) in 1.25s
```

```
~/svsm/kernel$ cargo verify
Compiling: svsm v0.1.0
error: possible arithmetic underflow/overflow
  --> kernel/src/address.rs:314:23
   |
314 |       VirtAddr::new(self.0 + offset)
   |                     ^^^^^^^^^^^^^^^^

note: diagnostics via expansion:
    self.0 + offset.has_type(Int(USize))
Finished in 2.26s
```

Incremental Verification on

a verified function calling another verified function

```
impl VirtAddr {
#[verus_verify]
#[requires(self@ + offset <= usize::MAX,
           offset <= 0x7FFF_FFFF_FFFF)]
fn add(&self, offset: usize) -> Self
{
    VirtAddr::new(self.0 + offset)
}

#[verus_verify]
fn foo(vaddr1: VirtAddr)
{
    let vaddr2 = vaddr1.add(0x8000_0000_0000
}
```

```
~/svsm/kernel$ cargo build
Compiling: svsm v0.1.0
Finished `dev` profile target(s) in 1.25s
```

```
~/svsm/kernel$ cargo verify
note: verifying module address
error: precondition not satisfied
  --> kernel/src/address.rs:446:17
434 |   #[requires(offset <= 0x7FFF_FFFF_FFFF,
                 ....)]
    |              ---------------------------- failed precondition
446 |   let vaddr2 = vaddr1.add(0x8000_0000_0000u64 as
usize);
    |                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
note: done with module address
verification results:: 1 verified, 1 errors
```

Incremental Verification on

an unverified function calling another verified function

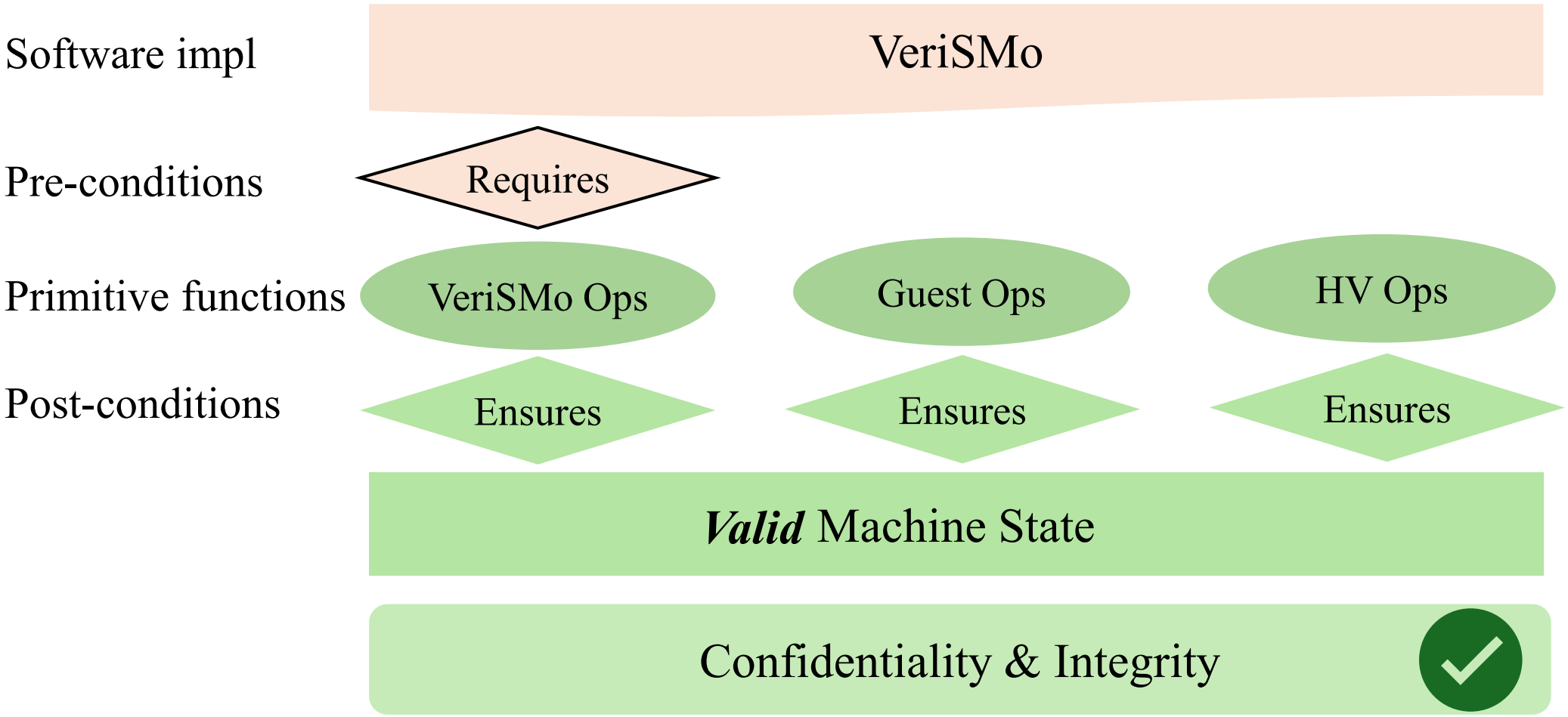
```
impl VirtAddr {
#[verus_verify]
#[requires(self@ + offset <= usize::MAX,
           offset <= 0x7FFF_FFFF_FFFF)]
fn add(&self, offset: usize) -> Self
{
    VirtAddr::new(self.0 + offset)
}

fn foo(vaddr1: VirtAddr)
{
    let vaddr2 = vaddr1.add(0x8000_0000_0000);
}
```

Verifier: Nothing is checked inside foo

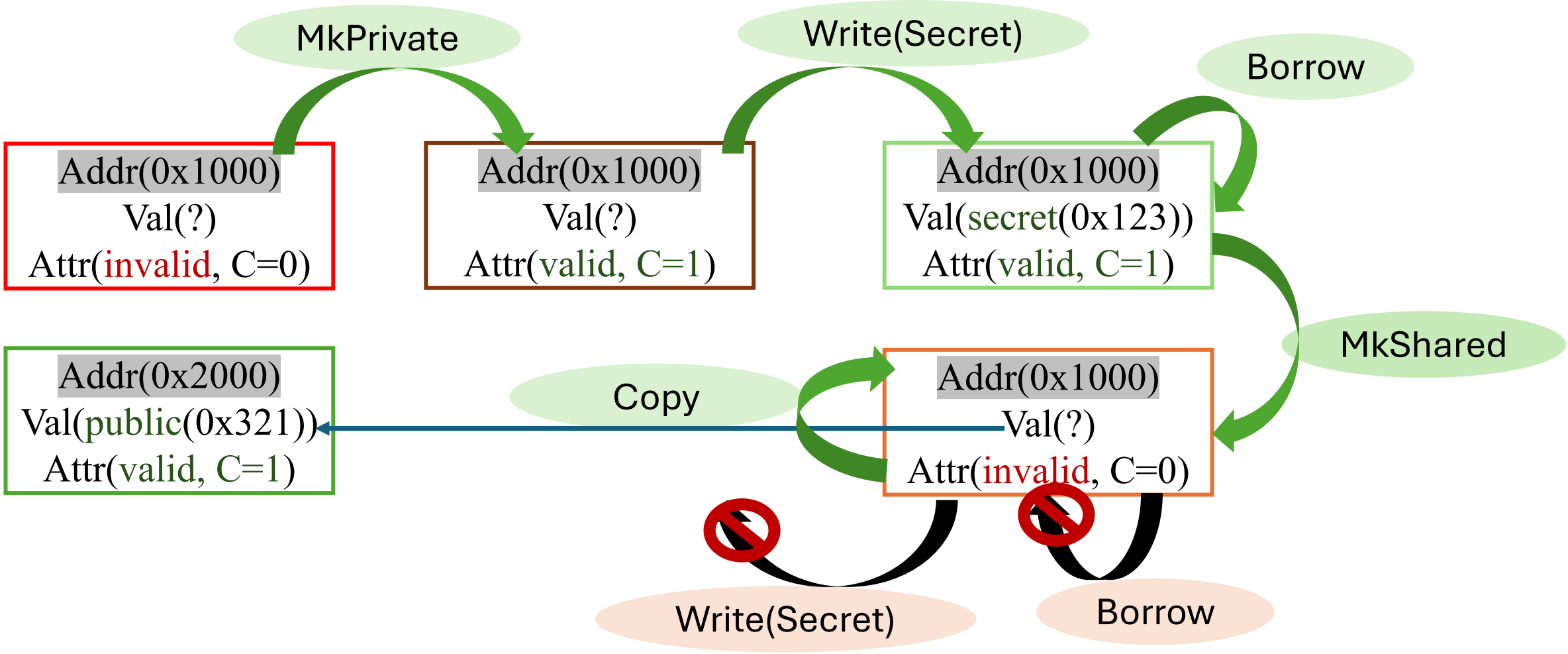
```
~/svsm/kernel$ cargo build
Compiling: svsm v0.1.0
Finished `dev` profile target(s) in 1.25s
```

```
~/svsm/kernel$ cargo verify
note: verifying module address
note: done with module address
verification results:: 1 verified, 0 errors
Finished `dev` profile target(s) in 2.26s
```

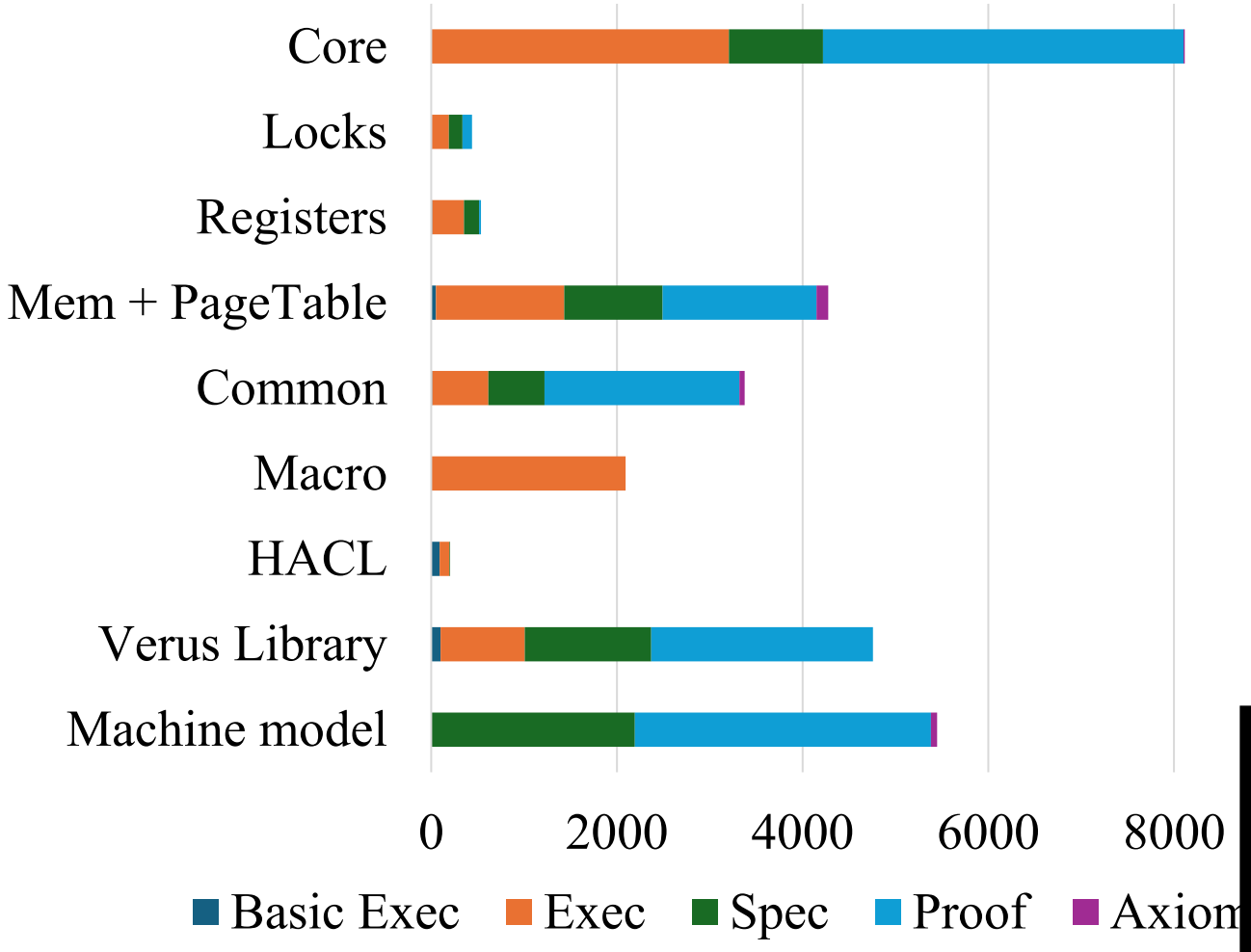
VeriSMo's verification design



Deeper security proved by Verus: An example of safe access to raw memory in VeriSMo



Verification summary of VeriSMo



- Verification performance
  - ✓ >8k lines of executable code
  - ✓ ~4min with 64 cores
- Runtime performance
  - ✓ Zero runtime overhead

```
~/verismo/source/verismo$ cargo verify
note: verifying module ...
note: verifying module ...
...
verification results:: 2144 verified, 0 errors
Finished dev [unoptimized + debuginfo] target(s) in 3m 37s
```

Verification Plan

- Stage 1: [Current] Help community developers understand the verification
  - ✓ Present how to apply verus to SVSM project.
  - ✓ Setup build environment and CI for verification: https://github.com/coconut-svsm/svsm/pull/486
- Stage 2: Start with kernel core code (closer to VeriSMo)
  - ❑ Reuse some proofs from VeriSMo.
    - May need to import some proofs or use it as an external crate.
  - ❑ Leverage VeriSMo's design for correct memory access.
    - Page table, RMP table tracking in verification mode.
  - ❑ Verified basic type -> allocator -> page table -> svsm protocol
- Stage 3: [Long-term] Identify new and critical security targets
  - ❑ Security properties that are not easy to tests via fuzzing and unit tests.
  - ❑ Page table isolation for user/kernel, for crypto vs non-crypto context, etc.
  - ❑ Future components

Backup slides

Solution: add assumption

```
#[verus_verify]
impl<T> From<*const T> for VirtAddr {
    #[inline]
    #[verus_verify]
    #[requires(ptr <= 0x7fff_ffff_ffff
               ptr >= ffff_8000_0000_0000)]
    fn from(ptr: *const T) -> Self {
        Self(ptr as InnerAddr)
    }
}

#[verus_verify]
impl<T> From<InnerAddr> for VirtAddr {
    #[inline]
    #[verus_verify]
    fn from(addr: InnerAddr) -> Self {
        sign_extend(addr)
    }
}
```

```
~/svsm/kernel$ cargo verify
note: verifying module address
note: verifying module types
note: done with module types
note: done with module address
verification results:: 45 verified, 0 errors
verification results:: 0 verified, 0 errors
   Finished dev [unoptimized + debuginfo] target(s) in 4.73s
```

Solution: fix the bug

```rust
#[verus_verify]
impl<T> From<*const T> for VirtAddr {
    #[inline]
    #[verus_verify]
    fn from(ptr: *const T) -> Self {
        Self::from(ptr as InnerAddr)
    }
}

#[verus_verify]
impl<T> From<InnerAddr> for VirtAddr {
    #[inline]
    #[verus_verify]
    fn from(addr: InnerAddr) -> Self {
        sign_extend(addr)
    }
}
```

```
~/svsm/kernel$ cargo verify
note: verifying module address
note: verifying module types
note: done with module types
note: done with module address
verification results:: 45 verified, 0 errors
verification results:: 0 verified, 0 errors
   Finished dev [unoptimized + debuginfo] target(s) in 4.73s
```

Solution: fix the bug

```
#[verus_verify]
impl<T> TryFrom<*const T> for VirtAddr {
    #[verus_verify]
    fn try_from(ptr: *const T) -> Self {
        type Error = &'static str;
        if ptr < (1<<SIGN_BIT) || ptr >= !(1<<SIGN_BIT - 1) {
            Ok(Self(ptr as InnerAddr))
        } else {
            Err("VirtAddress does not accept invalid raw pointer.")
        }
    }
}
#[verus_verify]
impl<T> From<InnerAddr> for VirtAddr {
    #[inline]
    #[verus_verify]
    fn from(addr: InnerAddr) -> Self {
        sign_extend(addr)
    }
}
```

```
~/svsm/kernel$ cargo verify
note: verifying module address
note: verifying module types
note: done with module types
note: done with module address
verification results:: 45 verified, 0 errors
verification results:: 0 verified, 0 errors
   Finished dev [unoptimized + debuginfo] target(s) in 4.73s
```

Coding efforts by verification developers:
Define specifications for VirtAddr

```
verus! {
broadcast use bit_properties;

impl VirtAddr {
    #[verifier::type_invariant]
    pub closed spec fn is_canonical_vaddr(&self) -> bool {
        vaddr_upper_all_zeros(self) || vaddr_upper_all_ones(self)
    }

}
```

✓ **Import basic proofs for the module**

✓ **Define type invariants**

Incremental Verification on

a verified function calling another function under development

```
impl VirtAddr {
#[verus_verify(external_body)]
#[requires(self@ + offset <= usize::MAX,
           offset <= 0x7FFF_FFFF_FFFF)]
fn add(&self, offset: usize) -> Self
{
    unimplemented!()
}

#[verus_verify]
fn foo(vaddr1: VirtAddr)
{
    let vaddr2 = vaddr1.add(0x8000_0000_0000
}
```

```
~/svsm/kernel$ cargo build
Compiling: svsm v0.1.0
Finished `dev` profile target(s) in 1.25s
```

```
Verifier[add]: Adding spec while ignoring
               function body.
Verifier[foo]: Verifying foo
```

```
~/svsm/kernel$ cargo verify
note: verifying module address
error: precondition not satisfied
  --> kernel/src/address.rs:446:17
434 |   #[requires(offset <= 0x7FFF_FFFF_FFFF,
              ....)]
    |              ---------------------------- failed precondition
446 |   let vaddr2 = vaddr1.add(0x8000_0000_0000u64 as
usize);
    |                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
note: done with module address
verification results:: 0 verified, 1 errors
```

Example: Annotate the binary search code

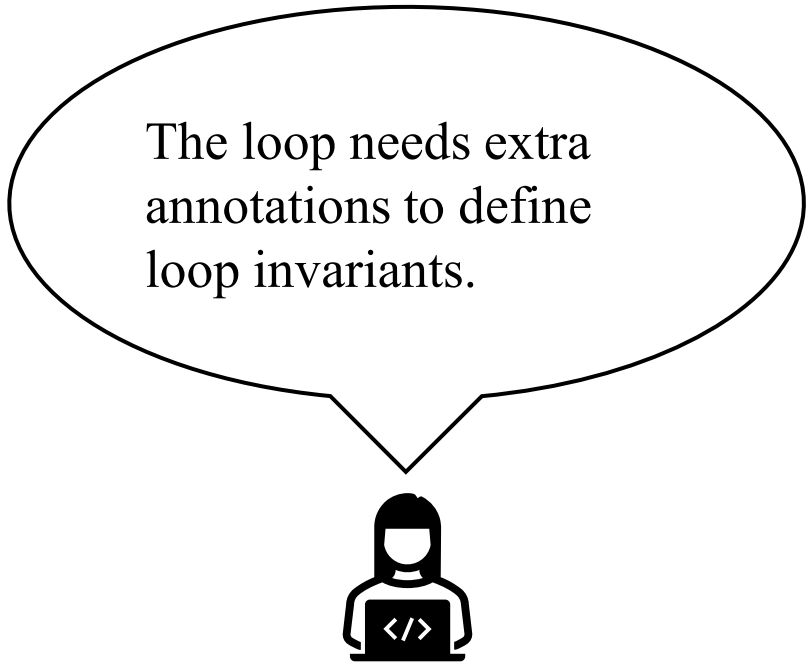
```
#[requires(
  forall|i: int, j: int| 0 <= i <= j < v.len() ==> v[i] <= v[j],
  exists|i: int| 0 <= i < v.len() && k == v[i],
)]
#[ensures(
  r < v.len(),
  val == v[r as int],
)]
// Given a sorted v, find the index r that v[r] equals to val
fn binary_search(v: &Vec<u64>, val: u64) -> (r: usize)
{
  let mut left: usize = 0;
  let mut right: usize = v.len() - 1;
  while left != right
  {
    let i = left + (right - left) / 2;
    if v[i] < val { left = i + 1;} else { right = i; }
  }
  left
}
```

I specify the algorithm pre-condition (requires) the expected post-condition (ensures), instead of writing unit tests.

Guarantee all corner cases.

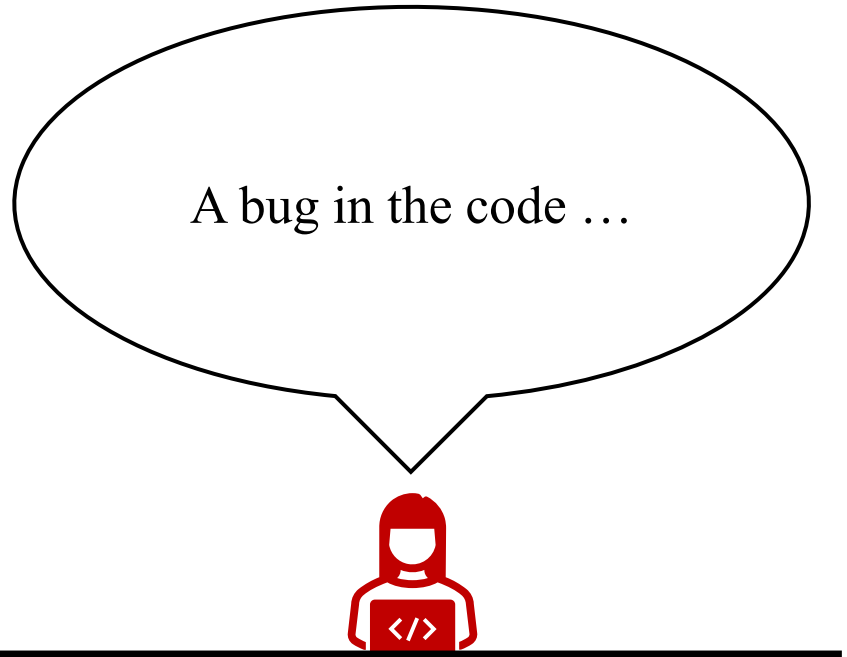
Example: Annotate the binary search code

```
#[requires(
  forall|i: int, j: int| 0 <= i <= j < v.len() ==> v[i] <= v[j],
  exists|i: int| 0 <= i < v.len() && k == v[i],
)]
#[ensures(
  r < v.len(),
  val == v[r as int],
)]
// Given a sorted v, find the index r that v[r] equals to val
fn binary_search(v: &Vec<u64>, val: u64) -> (r: usize)
{
  let mut left: usize = 0;
  let mut right: usize = v.len() - 1;
  #[invariant(
    forall|i: int, j: int| 0 <= i <= j < v.len() ==> v[i] <= v[j],
    right < v.len(),
    exists|i: int| left <= i <= right && val == v[i],
  )]
  while left != right {
    let i = left + (right - left) / 2;
    if v[i] < val { left = i + 1;} else { right = i; }
  }
  left
}
```



Example: Detect a bug in binary search

```
// Given a sorted v, find the index r that v[r] equals to val
fn binary_search(v: &Vec<u64>, val: u64) -> (r: usize)
requires
  forall|i: int, j: int| 0 <= i <= j < v.len() ==> v[i] <= v[j],
  exists|i: int| 0 <= i < v.len() && k == v[i],
ensures
  r < v.len(),
  val == v[r as int],
{
  let mut left: usize = 0;
  let mut right: usize = v.len() - 1;
  while left != right
  invariant
    forall|i: int, j: int| 0 <= i <= j < v.len() ==> v[i] <= v
    right < v.len(),
    exists|i: int| left <= i <= right && val == v[i],
  {
    let i = left + (right - left) / 2;
    if v[i] < val { left = i + 2;} else { right = i; }
  }
  left
}
```




```
C:\verus\source> verus.exe binsearch.rs
error: invariant not satisfied at end of loop body
 --> .\ binsearch.rs:29:13
29 |         exists|i: int| left <= i <= right && val == v[i],
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
error: aborting due to previous error

verification results:: 1 verified, 1 errors
```

Permission-based verification

- Uses the ***tracked resource permission*** to protect raw resource access
  - ✓Raw memory
  - ✓Page table
  - ✓RMP
  - ✓Lock
  - ✓Control registers

A bug in security module

```
fn mk_guest_priv(page: usize, Tracked(mperm): Tracked<&mut MemPerm<T>>) ->
bool
{
  // Reject if the page is not guest page or is not shared
  if !is_guest_page(page) || !is_shared(page) {retu
  …
  validate_page(page, true, Tracked(mperm));
✕
  rmpadj
  …
}
```

```
note: verifying module security::memory
error: precondition not satisfied
  --> verismo/src/security/memory.rs:27:13
27 |    let ret = rmpadjust(page, rmpattr, Track
   |              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  ::: verismo/src/ptr/snp/rmp/rmp_t.rs:88:9
88 |    old(perm)@.bytes().is_public_to(attr.spec_vmpl() as nat),
   |    ------------------------------------ failed precondition
```

Hypervisor sets the page mapping to a secret physical page

Guest OS can access VeriSMo secret

Protect raw memory access in VeriSMo

```
unsafe fn ptr_borrow<T>(
      addr: usize,
) → &T
{
      …
}
```

Protect raw memory access in VeriSMo

```
#[tracked(mperm: Tracked<&MemPerm<T>>, …)]
#[requires(
      mperm.id == vaddr,
      mperm.attr_valid_borrow()
)]
#[ensures(*ret == mperm.value)
fn ptr_borrow<T>(vaddr: usize) → (ret: &T)
{
      unsafe {…}
}
```

- An unforgeable object used by verification without runtime overhead.
- Similar to Rust's PhantomData