Formal verification of Coreboot SMI handler

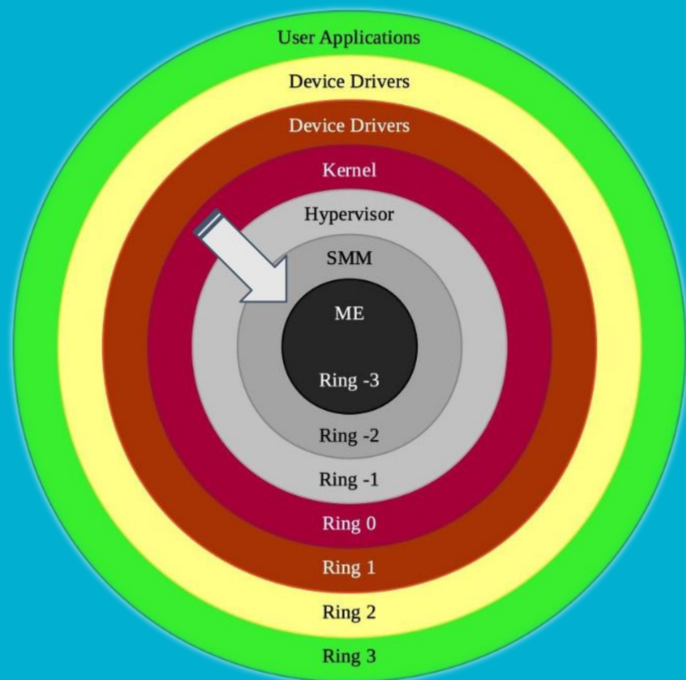
Vadim Zaliva, Digamma.ai

2023 TWISTED Quarterly Meeting @UCSC

Introduction

1. Digamma.ai: 18 years in business. Boutique consulting firm in Silicon Valley with engineering offices in Ukraine. ~100 employees, customers from small startups to Fortune 100 companies.
2. Three years ago identified Formal Verification as an interesting direction for future business opportunities. Decided to position ourselves as an early industry adopter.
3. Focus on small and medium enterprises. We are not specifically pursuing government contracts.
4. Initial focus on verifying existing software instead of rewriting and replacing it with verified code.
5. Built and trained a group of verification engineers.

System Management Mode



- SMM is a special-purpose operating mode provided for handling system-wide functions and intended for use only by system firmware.
- Can be invoked through System Management Interrupt.
- Is executed in distinct and isolated processor environment that operates transparently to the OS or user applications.
- Has full access to all physical memory, including BIOS region on the SPI flash.
- Is a target for malicious rootkits. SMM malware can do almost everything!
- It was shown that SMM rootkits are not only theoretically but also practically possible.

SMM memory corruption CVEs

CVE-2021-33626	A vulnerability exists in SMM (System Management Mode) branch that registers a SWSMI handler that does not sufficiently check or validate the allocated buffer pointer(QWORD values for CommBuffer). This can be used by an attacker to corrupt data in SMRAM memory and even lead to arbitrary code execution.	SMM memory corruption
CVE-2021-33625	An issue was discovered in Kernel 5.x in Insyde InsydeH2O, affecting HddPassword. Software SMI services that use the Communicate() function of the EFI_SMM_COMMUNICATION_PROTOCOL do not check whether the address of the buffer is valid, which allows use of SMRAM, MMIO, or OS kernel addresses.	SMM memory corruption
CVE-2021-26943	The UX360CA BIOS through 303 on ASUS laptops allow an attacker (with the ring 0 privilege) to overwrite nearly arbitrary physical memory locations, including SMRAM, and execute arbitrary code in the SMM (issue 3 of 3).	SMM memory corruption
CVE-2020-5388	Dell Inspiron 15 7579 2-in-1 BIOS versions prior to 1.31.0 contain an Improper SMM communication buffer verification vulnerability. A local authenticated malicious user may potentially exploit this vulnerability by using an SMI to gain arbitrary code execution in SMRAM.	SMM memory corruption
CVE-2006-6730	OpenBSD and NetBSD permit usermode code to kill the display server and write to the X.Org /dev/xf86 device, which allows local users with root privileges to reduce securelevel by replacing the System Management Mode (SMM) handler via a write to an SMRAM address within /dev/xf86 (aka the video card memory-mapped I/O range), and then launching the new handler via a System Management Interrupt (SMI), as demonstrated by a write to Programmed I/O port 0xB2.	SMM memory corruption
CVE-2020-27339	In the kernel in Insyde InsydeH2O 5.x, certain SMM drivers did not correctly validate the CommBuffer and CommBufferSize parameters, allowing callers to corrupt either the firmware or the OS memory. The fixed versions for this issue in the AhciBusDxe, IdeBusDxe, NvmExpressDxe, SdHostDriverDxe, and SdMmcDeviceDxe drivers are 05.16.25, 05.26.25, 05.35.25, 05.43.25, and 05.51.25 (for Kernel 5.1 through 5.5).	SMM memory corruption
CVE-2020-12890	Improper handling of pointers in the System Management Mode (SMM) handling code may allow for a privileged attacker with physical or administrative access to potentially manipulate the AMD Generic Encapsulated Software Architecture (AGESA) to execute arbitrary code undetected by the operating system.	SMM memory corruption
CVE-2018-12182	Insufficient memory write check in SMM service for EDK II may allow an authenticated user to potentially enable escalation of privilege, information disclosure and/or denial of service via local access.	SMM memory corruption
CVE-2017-5721	Insufficient input validation in system firmware for Intel NUC7i3BNK, NUC7i3BNH, NUC7i5BNK, NUC7i5BNH, NUC7i7BNH versions BN0049 and below allows local attackers to execute arbitrary code via manipulation of memory.	SMM memory corruption
CVE-2011-5174	Buffer overflow in Intel Trusted Execution Technology (TXT) SINIT Authenticated Code Modules (ACM) in Intel Q67 Express, C202, C204, C206 Chipsets, and Mobile Intel QM67, and QS67 Chipset before 2nd_gen_i5_i7_SINIT_51.BIN Express; Intel Q57, 3450 Chipsets and Mobile Intel QM57 and QS57 Express Chipset before i5_i7_DUAL_SINIT_51.BIN and i7_QUAD_SINIT_51.BIN; Mobile Intel GM45, GS45, and PM45 Express Chipset before GM45_GS45_PM45_SINIT_51.BIN; Intel Q35 Express Chipsets before Q35_SINIT_51.BIN; and Intel 5520, 5500, X58, and 7500 Chipsets before SINIT ACM 1.1 allows local users to bypass the Trusted Execution Technology protection mechanism and perform other unspecified SINIT ACM functions via unspecified vectors.	SMM memory corruption
And more...	...	SMM memory corruption

Coreboot SMI handler verification project

- We are going to verify that the SMI handler will never write to the code area in System Management RAM.
- This will prevent at least 20 known CVEs.
- Verified Software Toolchain is used for formal verification of the C code.
- Coreboot is a project to develop open-source boot firmware.
- Coreboot is used on ChromeOS devices.
- Coreboot with Gemini Lake chipset & Google Octopus mainboard are used on more than 25 Chromebook models including ASUS Chromebook CX1101, Lenovo Ideapad 3, and Samsung Chromebook 4.

Porting Coreboot SMI handler to VST

```
#ifdef __VERIFICATION_CHANGE__
    write32((void *)((uint8_t *)spibar + SPIBAR_DLOCK), dlock);
#else
    write32(spibar + SPIBAR_DLOCK, dlock);
#endif
```

- Coreboot: 11,802 files/1,769,995 LoC
- Gemini Lake SMI handler: 367 files/57,821 LoC
- Binary size of compiled SMM code: 622 Kb
- 168 modifications to the source code were introduced for compatibility with Compcert & VST.
- We have ensured that code with and without our changes compiles to exactly the same binary.

Example of integer-to-pointer casts

```
const uintptr_t pci_mmconf = (uintptr_t)CONFIG_ECAM_MMCONF_BASE_ADDRESS ;

static inline volatile union pci_bank *pci_map_bus(pci_devfn_t dev) {
   return (void *)&((u8 *)ptr_of_int(pci_mmconf))[PCI_DEVFN_OFFSET(dev)];
}

void pci_s_write_config8(pci_devfn_t dev, uint16_t reg, uint8_t value) {
   pci_map_bus(dev)->reg8[reg] = value;
}
```

- Integer to pointer casts frequently occur in coreboot code but are not supported by VST
- Enhanced Configuration Access Mechanism (ECAM) can be used for PCIe configuration reads and writes.
- `pci_s_write_config8()` function shown above is from Coreboot.

VST concrete addresses reasoning extension

```
uintptr_t int_of_ptr(void *);
```

```
Definition int_of_ptr_spec : ident * funspec :=
  DECLARE _int_of_ptr
  WITH p : val
  PRE [tptr tvoid]
    PROP ()
    PARAMS (p)
    SEP ()
  POST [tuint] EX (a : int),
    PROP (val_has_address p a)
    RETURN (Vint a)
    SEP ().
```

```
void * ptr_of_int(uintptr_t);
```

```
Definition ptr_of_int_spec : ident * funspec :=
  DECLARE _ptr_of_int
  WITH p : val, a : int
  PRE [tuint]
    PROP (val_has_address p a)
    PARAMS (Vint a)
    SEP ()
  POST [tptr tvoid]
    PROP ()
    RETURN (p)
    SEP ().
```

- VST uses abstract memory model, while we need to prove properties about concrete addresses.
- SMI handler code uses type casts between pointers and integers which are not supported by VST.
- `int_of_ptr()` and `ptr_of_int()` functions were introduced and their specifications were axiomatized.
- `val_has_address p a` says that pointer value `p` corresponds to the address `a` in concrete memory model.

Integrating concrete memory model into VST

```
Definition _pci_mmconf_OBJ : ident := $"pci_mmconf_OBJ".
Definition pci_mmconf_size : Z := 1048576.
Definition pci_mmconf_type : type :=
  tarray tuchar pci_mmconf_size.
```

```
Definition Vprog : varspecs.
 assert (vs : varspecs) by mk_varspecs prog.
 exact ((_pci_mmconf_OBJ, pci_mmconf_type) :: vs).
Defined.
```

```
Parameter has_address : block -> offset -> int ->Prop.
Axiom pci_mmconf_address_rel : ∀ (gv : globals) (b : block) (p : ptrofs),
  gv _pci_mmconf_OBJ = Vptr b p ->
  has_address b p pci_mmconf_addr.
```

- In order to apply `ptr_of_int_spec` in VST proof we need to satisfy its precondition: provide an address and a pointer, and prove that this pointer is related to this address.
- `CONFIG_ECAM_MMCONF_BASE_ADDRESS` is not related to a pointer to valid C object, it is the base address of PCIe memory-mapped configuration area.
- We create a “fake” VST object for PCIe memory-mapped configuration area and add an axiom about its relation to the base address.
- We will have several such assumptions for pointer-address relations in the SMI handler specification. From these we can deduce that memory areas used for write don’t overlap with SMRAM.

VST specification for *pci_s_write_config8*

```
Definition pci_s_write_config8_spec : ident * funspec :=
  DECLARE _pci_s_write_config8
  WITH gv : globals, lb : list byte, dev : int, reg : int, value : byte
  PRE [tuint, tushort, tuchar]
    PROP (Int.unsigned dev < pci_mmconf_size - pci_mmconf_dev_size;
          Int.unsigned reg < pci_mmconf_dev_size)
    PARAMS (Vint dev; Vint reg; Vubyte value)
    GLOBALS (gv)
    SEP (data_at Tsh tuint (Vint pci_mmconf_addr) (gv _pci_mmconf);
         data_at Tsh pci_mmconf_type (map Vubyte lb) (gv   _pci_mmconf_OBJ))
  POST [tvoid]
    PROP ()
    RETURN ()
    SEP (data_at Tsh tuint (Vint pci_mmconf_addr) (gv _pci_mmconf);
         data_at Tsh pci_mmconf_type (map Vubyte (pci_s_write_config8 lb dev reg value)) (gv   _pci_mmconf_OBJ)).
```

```
Axiom pci_mmconf_address_rel :
  ∀ (gv : globals) (b : block) (p : ptrofs),
  gv _pci_mmconf_OBJ = Vptr b p ->
  has_address b p pci_mmconf_addr.
```

```
Definition pci_mmconf_type : type :=
  tarray tuchar pci_mmconf_size.
```

SMM region overlap runtime checking

As mentioned in the coreboot documentation [1]:

"In order to not leak SMM internals or accidentally overwrite parts of SMM, ring0 provided data (pointers, offsets, sizes, …) must be checked before using them in SMM.

There exist two methods to verify data:

/* Returns true if the region overlaps with the SMM */

```
bool smm_region_overlaps_handler(struct region *r);
```

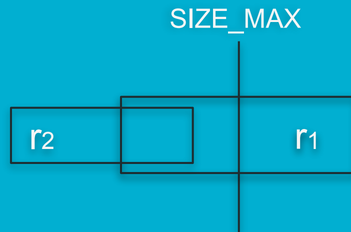
/* Returns true if the region overlaps with the SMM */

```
static inline bool smm_points_to_smram(const void *ptr, const size_t len);
```

```
static inline bool region_overlap(const struct region *r1,
const struct region *r2) {
  return (region_end(r1) > region_offset(r2)) &&
         (region_offset(r1) < region_end(r2));
}
```

Example of incorrect behaviour:

```
struct region r1 = {SIZE_MAX - 10, 20};
struct region r2 = {SIZE_MAX - 20, 15};
```



[1] https://doc.coreboot.org/security/smm.html

Corrected region overlap check

Original C code

```
static inline bool region_overlap(const struct region *r1,
  const struct region *r2) {
    return (region_end(r1) > region_offset(r2)) &&
           (region_offset(r1) < region_end(r2));
}
```

Corrected C code

```
static inline bool region_overlap_correct(const struct region *r1,
  const struct region *r2) {
    if (region_sz(r1) == 0 || region_sz(r2) == 0)
        return false;
    size_t size1 = min(region_sz(r1) - 1, SIZE_MAX - region_offset(r1));
    size_t size2 = min(region_sz(r2) - 1, SIZE_MAX - region_offset(r2));
    return (region_offset(r1) + size1 ≥ region_offset(r2)) &&
           (region_offset(r1) ≤ size2 + region_offset(r2));
}
```

Functional specification for the original:

```
Definition region_overlap (r1 r2 : region int) : bool
:=
  (Int.ltu (region_offset r2 )
           (region_end_int r1 )) &&
  (Int.ltu (region_offset r1 )
           (region_end_int r2 )).
```

Functional specification for corrected function:

```
Definition region_overlap_correct (r1 r2 : region int) : bool :=
  if Int.eq (region_sz r1 ) Int.zero || Int.eq (region_sz r2 ) Int.zero then false
  else
    let size1 := int_min (Int.sub (region_sz r1 ) Int.one)
                         (Int.sub (Int.repr Int.max_unsigned ) (region_offset r1 ))
in
    let size2 := int_min (Int.sub (region_sz r2 ) Int.one)
                         (Int.sub (Int.repr Int.max_unsigned ) (region_offset r2 ))
in
      (negb (Int.ltu (Int.add (region_offset r1 ) size1) (region_offset r2 ))) &&
      (negb (Int.ltu (Int.add (region_offset r2 ) size2) (region_offset r1 ))).
```

Abstract specification:

```
Definition in_region (a : Z) (r : region Z) : Prop := a ≥ region_offset r ∧ a < region_end_Z r .
Definition region_overlap_rel_Z (r1 r2 : region Z) : Prop := ∃ (a : Z), in_region a r1 ∧ in_region a r2 .
Definition region_overlap_rel (r1 r2 : region int) : Prop := region_overlap_rel_Z (region_Z_of_int r1 ) (region_Z_of_int r2 ).
```

VST specifications for *region_overlap*

VST specifications

```
Definition region_overlap_spec : ident * funspec :=
  DECLARE _region_overlap
  WITH sh: share, p1 : val, p2 : val,
       r1 : region int, r2 : region int
  PRE [tptr tregion, tptr tregion]
    PROP (readable_share sh)
    PARAMS (p1; p2)
    SEP (region_at sh r1 p1 ; region_at sh r2 p2)
  POST [tbool]
    PROP ()
    RETURN (Val.of_bool (region_overlap r1 r2))
    SEP (region_at sh r1 p1 ; region_at sh r2 p2).
```

```
Definition region_overlap_correct_spec : ident * funspec :=
  DECLARE _region_overlap_correct
  WITH sh: share, p1 : val, p2 : val,
       r1 : region int, r2 : region int
  PRE [tptr tregion, tptr tregion]
    PROP (readable_share sh)
    PARAMS (p1; p2)
    SEP (region_at sh r1 p1 ; region_at sh r2 p2)
  POST [tbool]
    PROP ()
    RETURN (Val.of_bool (region_overlap_correct r1 r2))
    SEP (region_at sh r1 p1 ; region_at sh r2 p2).
```

For corrected function functional specification implements abstract specification:

```
Lemma region_rel_func (r1 r2 : region int) : region_overlap_rel r1 r2 ↔ region_overlap_correct r1 r2  = true.
```

For old (incorrect) function exists a false negative:

```
Lemma false_negative_exists : ∃ (r1 r2 : region int),
  (region_overlap r1 r2 = false) ∧ (region_overlap_correct r1 r2 = true).
```

Use of *region_overlap* in brightness function

```
static void mainboard_smi_brightness_down(void) {
  uint32_t reg32 = pci_read_config32(PCI_DEV( 1, 0, 0), PCI_BASE_ADDRESS_2)  &
~0xf;
  u8 *bar = (void *)(uintptr_t)reg32;

  /* Validate pointer before using it */
  if (!bar || smm_points_to_smram(bar, LVTMA_BL_MOD_LEVEL    + sizeof(uint8_t)))
    return;

  *(bar + LVTMA_BL_MOD_LEVEL)  &= 0xf0;
  if (*(bar + LVTMA_BL_MOD_LEVEL)  > 0x10)
    *(bar + LVTMA_BL_MOD_LEVEL)  -= 0x10;
}
```

```
bool smm_region_overlaps_handler(const struct region *r) {
  const struct region r_smm = {SMM_BASE, SMM_DEFAULT_SIZE};
  return region_overlap(&r_smm, r);
}
```

```
#define LVTMA_BL_MOD_LEVEL 0x7af9
#define SMM_BASE           0xa0000
#define SMM_DEFAULT_SIZE   0x10000
```

Integer overflow bug does not manifest with arguments used in the brightness function:

```
Lemma brightness_correct : ∀ off : int,
  region_overlap         r_smm (Int.and off (Int.not (Int.repr 0xf)), Int.add LVTMA_BL_MOD_LEVEL Int.one) =
  region_overlap_correct r_smm (Int.and off (Int.not (Int.repr 0xf)), Int.add LVTMA_BL_MOD_LEVEL Int.one).
```

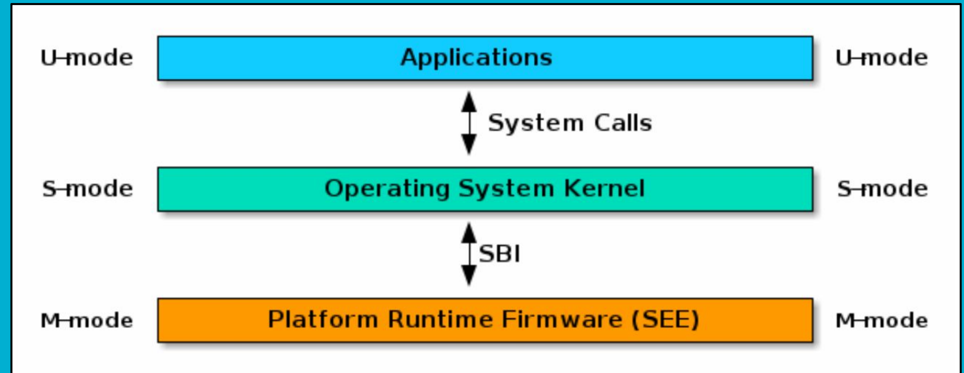
Project status

- Finished feasibility study part of the project.
- Ported Coreboot SMM code for Gemini Lake chipset to VST and verified that our changes produce the same binary as unchanged code.
- Developed an approach of dealing with pointer↔integer casts and concrete memory addresses.
- Proved some sample functions to validate the approach.
- Discussed with some potential business partners to confirm product business viability.
- Intend to proceed to the next stage: proving all of Coreboot SMI handler code for Gemini Lake chipset.

RISC-V ISA

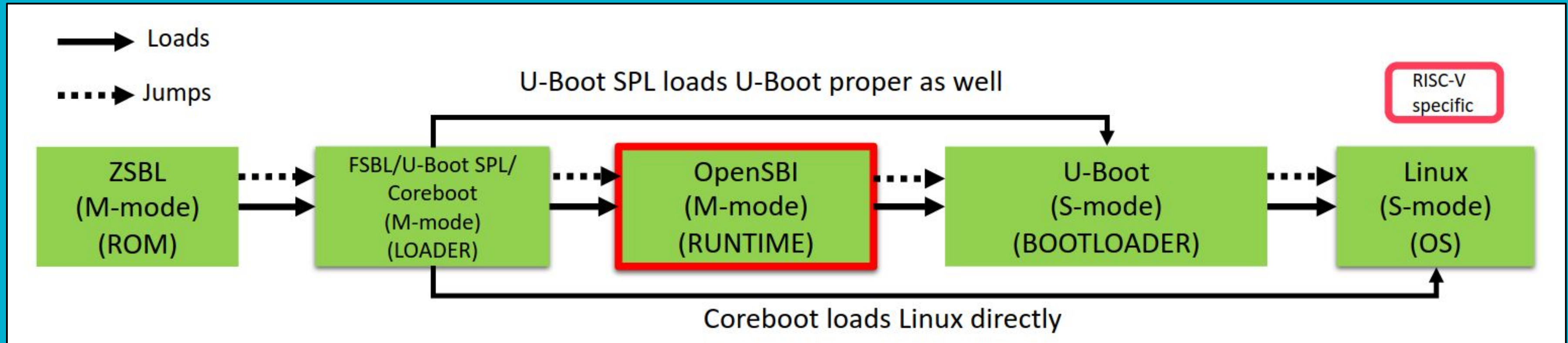
- RISC-V is new open source instruction set architecture that was originally designed to support research and education, but aims to become a standard open architecture for industry implementations.

- RISC-V modes of execution:
  - U-mode: for user applications
  - S-mode: for operating system
  - M-mode: for firmware
  - H-extension to support hypervisors



RISC-V firmware

- Supervisor Binary Interface (SBI) specification defines how firmware running in M-mode and OS running in S-mode can interact with each other.
- OpenSBI is an open-source reference implementation of RISC-V run-time firmware adhering to SBI specification.
- We conjecture that OpenSBI may be subject to similar privilege escalation vulnerabilities as x86 SMI handlers.



Questions?

Contact: Vadim Zaliva <*lord@digamma.ai*>

Backup slides

Compcert compatibility changes to SMM code

- Arithmetic on `void *` pointers: added additional type casts.
- `typeof()` operator used in macros: specialized for particular types.
- Statements in expression extension used in macros: replaced with one-line macros implementations or manually unfolded.
- `__always_inline` attribute: replaced with `inline`.
- Other unsupported attributes: replaced with supported alternatives or removed.
- Too big values in enums: replaced with `#define`d constants.
- Struct empty initializers `{}`: replaced with `{0}`.