FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Bezpečnostní aspekty Intel Management Engine
Student:	Bc. Michal Funtán
Vedoucí:	Ing. Josef Kokeš
Studijní program:	Informatika
Studijní obor:	Počítačová bezpečnost
Katedra:	Katedra informační bezpečnosti
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Proveďte rešerši dokumentace k Intel Management Engine, nastudujte principy této technologie a známá omezení.
Popište architekturu firmware Intel ME. Zaměřte se na zavádění a start systému, načítání modulů a závislosti mezi moduly.
Na dostupném hardwaru proveďte analýzu modulů pracujících s kryptografií, vypište jejich seznam a pokuste se určit jejich účel.
Ověřte kontrolu integrity načítaných modulů, tak jak je deklarována v dostupné literatuře.
Ověřte korektnost implementace kryptografických funkcí.Vyhodnoťte svá zjištění z pohledu bezpečnosti uživatele.

Seznam odborné literatury

RUAN, Xiaoyu. Platform embedded security technology revealed: safeguarding the future of computing with Intel embedded security and management engine. Berkeley, California: Apress Open, [2014]. ISBN 143026571X.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 12. ledna 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Bezpečnostní aspekty Intel Management Engine

Bc. Michal Funtán

Katedra informační bezpečnosti
Vedoucí práce: Ing. Josef Kokeš

6. května 2019

Poděkování

Chtěl bych poděkovat vedoucímu práce za rady, které mi při psaní práce poskytoval. Dále bych chtěl poděkovat svým rodičům za podporu nejen při psaní této práce, ale i během celého studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 6. května 2019 .....

Odkaz na tuto práci

Funtán, Michal. *Bezpečnostní aspekty Intel Management Engine*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato práce je věnována systému Intel Management Engine, který je součástí většiny čipových sad vyvíjených společností Intel. V první části práce je popsána architektura a zavádění systému. S využitím reverzního inženýrství byly analyzovány moduly, které se podílí na zavedení systému a byla ověřena kontrola integrity jednotlivých komponent systému. Ve druhé části byl reverzní analýzou zkoumán ovladač hardwarového kryptografického modulu s cílem identifikovat a popsat poskytované kryptografické funkce. V poslední kapitole jsou posouzeny získané poznatky z pohledu bezpečnosti uživatele počítačového systému.

Klíčová slova Intel Management Engine, Converged Security, spouštění systému, reverzní inženýrství, Platform Controller Hub, integrita systému

Abstract

This thesis describes Intel Management Engine, part of most chipsets developed by the Intel Corporation. First part introduces architecture and

boot process of the system. Modules which take part on booting was analyzed using reverse engineering and the system integrity check was validated. In the second part of this work, the driver of the hardware cryptographic module was analyzed using reverse engineering and implemented cryptographic functions was described. In the last part, acquired results was evaluated from the perspective of computer system security.

Keywords Intel Management Engine, Converged Security, boot process, reverse engineering, Platform Controller Hub, system integrity

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Společnost Intel má dnes nezanedbatelný podíl na produkci komponent osobních počítačů a serverů. Součastí všech jejích čipových sad pro osobní počítače a servery je Intel Management Engine. Jedná se o subsystém, jež není příliš detailně dokumentován a jehož přítomnost je známa jen omezenému množství uživatelů. Protože byl Intel ME původně vyvinut za účelem vzdálené správy, má přístup k síťovému rozhraní a zůstává aktivní i v případě, kdy je zbytek systému v hibernaci nebo vypnutý.

Z pohledu bezpečnosti představuje Intel ME dvousečnou zbraň. Na jednu stranu je autonomie a hardwarové oddělení ideální pro bezpečnostní aplikace, které je potřeba izolovat od ostatního software. Na druhou stranu, pokud by se útočníkovi podařilo infikovat Intel ME maligním kódem, pro uživatele by bylo prakticky nemožné takovou infekci detekovat. Útočník by si tímto způsobem zajistil persistenci i při reinstalaci celého operačního systému a díky přístupu k síťovému rozhraní by měl k systému vzdálený přístup.

Aby vývojáři Intel zamezili podobnému scénáři, věnovali značnou pozornost zabezpečení celého systému. Abych si nezávisle ověřil úroveň zabezpečení, rozhodl jsem se zabývat se načítáním systému a kontrole jeho integrity. A protože s tím úzce souvisí kryptografie, rozhodl jsem se druhou část práce věnovat kryptografickému subsystému Intel ME.

Práce je rozdělena do tří částí. První část je věnována architektuře počítače. Jejím cílem je ozřejmit čtenáři, kde se v počítači Intel ME nachází. Druhá část je věnována kontrole integrity systému, která je klíčová pro zajištění bezpečnosti systému. Třetí část je věnována kryptografickému koprocesoru a funkcím jež poskytuje buď přímo nebo prostřednictvím svého ovladače. V poslední kapitole jsou získané poznatky shrnuty a posouzeny z pohledu bezpečnosti uživatele počítačového systému.

Kapitola 1

Seznámení s Intel Management Engine

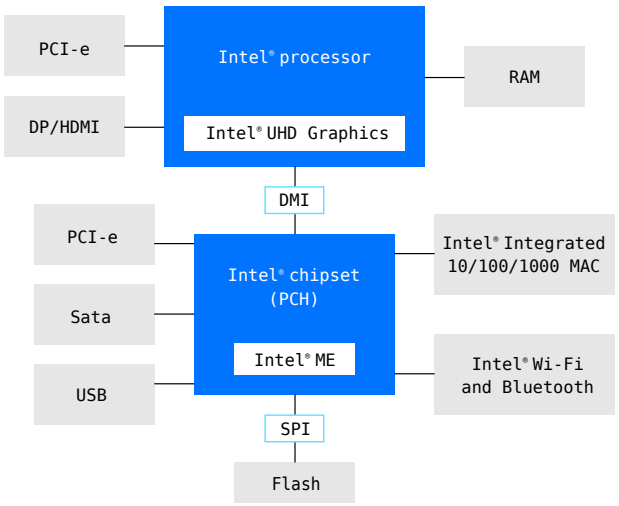
Společnost Intel se zabývá vývojem komponent počítačových systémů. Do jejího portfolia spadají, mimo jiné, procesory (Xeon, Core, Atom, Pentium, Celeron) a čipové sady. Hlavní část této práce je věnována subsystému, který se nachází uvnitř většiny aktuálních čipových sad vyvíjených společností Intel. Pro lepší orientaci v dalším textu je v první kapitole popsána architektura počítače postaveného na produktech společnosti Intel. Je zde rozebrána struktura systému a umístění jednotlivých komponent, kterým je věnován další text.

V rámci této práce je počítač sestavou procesoru, operační paměti a periferií. Procesor samotný zahrnuje výpočetní jádra, cache paměti, řadič operační paměti, řadiče periferií a v některých případech také grafické jádro. Další periferie jsou procesoru přístupné skrze čipovou sadu (*chipset*). Čipová sada je součástí základní desky a určuje, s jakými periferiemi bude moci procesor komunikovat a kolik jich bude.

Chipset na platformě Intel je již několik posledních let označován zkratkou PCH (*Platform Controller Hub*). Na obrázku 1.1 je vyobrazen blokový diagram popisující vztah mezi procesorem, PCH a periferiemi.

1.1 Intel Management Engine

Prvotní motivací pro vznik Intel Management Engine byla potřeba vzdálené správy počítačů, která by nebyla závislá na operačním systému (tzv. *out-of-band*). Tento přístup má jednu výhodu: systém je možné vzdáleně spravovat i v případě, kdy je operační systém nefunkční.



Obrázek 1.1: Čipová sada s procesorem Intel [1]

V roce 2005 byl společností Intel poprvé zaveden systém pro vzdálenou správu pod názvem Active Management Technology (AMT). První verze AMT byly součástí síťové karty. O dva roky později byl celý systém upraven. Do chipsetu (nyní PCH) byl přidán dedikovaný procesor s pamětí a periferiemi a byl nazván Intel Management Engine (ME).[2]

Intel ME v sobě kombinuje výhody hardwarové izolace a softwarové variability. Díky oddělenému hardware může ME fungovat nezávisle na hlavním operačním systému, jeho funcionalita je dostupná (v omezené míře) i v případě kdy není možné nabootovat operační systém nebo je systém vypnutý. Výhodou softwarové implementace je potom možnost instalace bezpečnostních záplat, případně možnost rozšiřování funkcionality.

Postupem času se ukázalo, že je ME vhodný nejen pro *out-of-band* správu, ale také jako platforma pro některé bezpečnostní aplikace, které mohou těžit z hardwarové izolace od hlavního procesoru. AMT v současné době není součástí všech instancí ME, na druhou stranu ME je dnes součástí všech produktů postavených na procesorech, lépe řečeno čipových sadách, společnosti Intel od stolních počítačů přes laptopy až po servery.[2]

Tato práce je věnována posledním dvěma verzím Intel ME, které jsou součástí čipových sad uvedených v tabulce 1.1. U každého chipsetu je uveden kvartál a rok, ve kterém byl daný chipset uvedený na trh. Pokud se v názvu chipsetu vyskytuje písmeno M, jedná se o chipset pro mobilní zařízení. V případě, že název obsahuje písmeno C, jedná se o chipset pro nasazení v serverech a pracovních stanicích.

V posledních době Intel pozměnil název Management Engine a v nové

Tabulka 1.1: Čipové sady s Intel ME verzí 11 a 12

Intel ME 11		**Intel ME 12**	
B365 (4'18)	C232 (4'15)	Z390 (4'18)	C621 (3'17)
Z370 (4'17)	C236 (4'15)	C242 (4'18)	C622 (3'17)
C422 (3'17)	HM170 (3'15)	C246 (3'18)	C624 (3'17)
X299 (2'17)	QM170 (3'15)	C629 (3'18)	C625 (3'17)
Q270 (1'17)	B150 (3'15)	H370 (2'18)	C626 (3'17)
H270 (1'17)	CM236 (3'15)	HM370 (2'18)	C627 (3'17)
Q250 (1'17)	Z170 (3'15)	QM370 (2'18)	C628 (3'17)
B250 (1'17)	Q170 (3'15)	B360 (2'18)	
HM175 (1'17)	Q150 (3'15)	H310 (2'18)	
QM175 (1'17)	H170 (3'15)	Q370 (2'18)	
Z270 (1'17)	H110 (3'15)	CM246 (2'18)	
CM238 (1'17)			

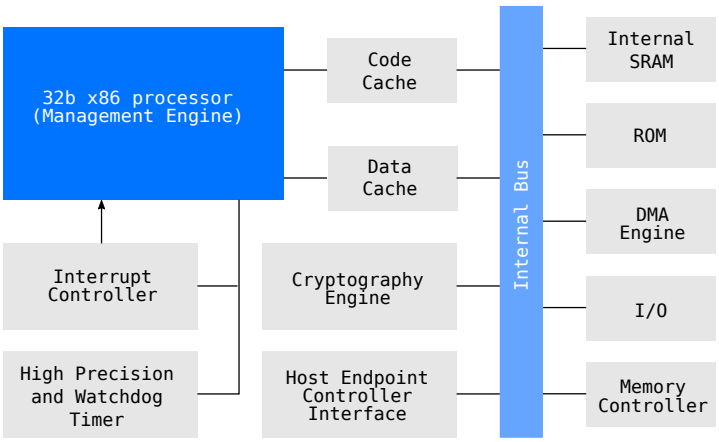
dokumentaci mluví o Intel Converged Security and Management Engine (CSME). Intel (CS)ME v současné době označuje výpočetní prostředí s dedikovaným hardware a firmware. Dřívější verze byly postaveny na procesorech různých architektur nicméně poslední verze využívají 32b x86 procesoru.

Kromě procesoru zahrnuje Intel ME subsystém také DMA řadič pro přístup k DRAM (RAM hlavního procesoru) a kryptografický engine (viz obrázek 1.2). Celý subsystém je izolovaný od hlavního procesoru, na kterém běží operační systém. Existují pouze dvě možnosti, jak mohou aplikace běžící na hlavním procesoru (OS, BIOS) komunikovat s aplikacemi běžícími na Intel ME:

- HECI (*Host-Embedded Communication Interface*) – určeno pro přenos malého množství dat mezi hlavním procesorem a ME. Obvykle se využívá pro přenos příkazů (požadavků). HECI je přístupné pouze v privilegovaném režimu. V rámci operačního systému běžícím na hlavním procesoru se ovladač pro přístup k HECI nazývá MEI (*Management Engine Interface*).

- DMA (*Direct Memory Access*) – používá se pro přenos většího objemu dat mezi hlavním procesorem a ME. DMA řadič pracuje s fyzickými adresami a programovat ho může pouze ME.

Systém má k dispozici malou paměť SRAM (256 kiB - 1 MiB) pro runtime kód a data. Pro odkládání stránek (*swap*) slouží část DRAM (4 MiB -



Obrázek 1.2: Hardware Intel ME[2]

32 MiB) hlavního procesoru, která je pro Intel ME rezervována během bootování BIOSem. DRAM přidělená ME by měla být chráněna před přístupem hlavního procesoru. Protože ale bylo v minulosti demonstrováno, že tomu tak nemusí být [3], jsou:

- kódové stránky opatřeny kontrolním součtem,
- datové stránky šifrovány a opatřeny kontrolním součtem.

Pokud je zařízení vypnuté nebo v hibernaci (*low power state*), DRAM není k dispozici a některé aplikace Intel ME nemusí být dostupné.[2]

Kryptografický engine obsahuje hardwarovou implementaci některých algoritmů náročných na výpočetní prostředky. Mezi implementovanými algoritmy jsou AES, SHA-256, DRNG a aritmetika s velkými čísly. Kryptografický engine není přímo přístupný hlavnímu procesoru, nicméně některé aplikace Intel ME mohou zprostředkovat jeho funkcionalitu nepřímo. V kapitole 3.6 je detailněji rozebrán ovladač kryptografického modulu.

Intel ME je vybaven speciálním nevolatilním úložištěm, které se nazývá *field programmable fuses*. Jeho hlavní charakteristikou je možnost pouze jednoho zápisu – pokud byla do buňky zapsána hodnota 1 už nemůže být změněna. Takové úložiště je ideální pro data, která musí přežít i přepsání flash paměti. Příkladem takových dat je hash klíče použitého technologií Boot Guard.[2]

Vstupním bodem firmware je kód z ROM, který spustí bootování ME. ROM slouží jako *root of trust*. Zbytek firmware (ME) je uložen na sdílené SPI flash paměti a může být v případě potřeby aktualizován.

Tabulka 1.2: Vybrané regiony SPI flash paměti

region	obsah
0	Flash descriptor
1	BIOS/UEFI
2	Intel Management Engine
3	Gigabit Ethernet

Management Engine je možné aktualizovat skrze HECI prostřednictvím aktualizačního příkazu. Pokud firmware tento příkaz obdrží nejprve ověří integritu nového firmware, který musí být podepsán klíčem společnosti Intel Veřejný klíč k ověření podpisu je součástí ME. V případě aktualizace je nejprve vytvořena záložní kopiji starého firmware. Pokud dojde během aktualizace k výpadku napájení, ROM detekuje porušenou integritu firmware a obnoví starý firmware ze zálohy.[4]

V dostupných zdrojích jsou také zmiňovány pojmy Server Platform Services (SPS) [5, 6] a Trusted Execution Engine (TXE) [7, 6]. Management Engine, SPS a TXE se mohou lišit poskytovanými funkcemi, ale v principu se jedná o jiný název pro totožný systém. O SPS se mluví v případě serverových chipsetů zatímco o TXE hovoříme v případě zařízení s procesory Intel Atom. Dokumentace k nejnovějším procesorům Intel Atom se již o TXE nezmiňuje a používá název Intel ME[8].

Firmware Management Engine je založen na modifikovaném operačním systému MINIX 3 a jako takový se skládá z modulů (**kernel**, **pm**, **vfs**...), které představují jádro operačního systému, ovladače a aplikace.[6]

1.2 SPI flash paměť

K PCH je připojena prostřednictvím rozhraní SPI flash paměť (viz obrázek 1.1). Paměť je rozdělena do několika regionů (tabulka 1.2), jejichž velikost a přístupová oprávnění jsou popsány ve flash descriptoru (FD). FD je v podstatě partition tabulkou celé flash paměti. Nachází se v prvním 4kiB bloku a může být modifikován pouze během výroby. Ve chvíli, kdy zařízení opouští výrobní linku, by měl být nastaven do režimu pouze pro čtení.

K jednotlivým regionům mají přístup čtyři jednotky: BIOS (CPU), Gigabit Ethernet, Management Engine a Embedded Controller/Baseboard Management Controller (EC/BMC). Podle dostupné dokumentace může v základní konfiguraci každá jednotka přímo číst a zapisovat pouze jí náležící region SPI flash paměti. [9]

Tabulka 1.3: Ilustrace obsahu CPD

	název souboru
1	`FTPR.Man`
2	`rbe`
3	`rbe.met`
4	`kernel`
5	`kernel.met`
6	`syslib`
7	`syslib.met`

1.2.1 Region Intel ME

Region Intel ME na flash paměti je rozdělen do několika oddílů popsaných v FPT (*flash partition table*). Pro pochopení dalšího textu není nutné znát přesnou strukturu FPT, nicméně je vhodné zmínit alespoň některé z oddílů, které se v regionu Intel ME mohou nacházet:

- `FTPR` – obsahuje základní kód firmware
- `NFTP` – další kód firmware
- `MFS` – nevolatilní úložiště pro data
- `ROMB` – oddíl pro ROMB (viz kapitola 3.1)

V případě SPS existuje také oddíl `REC` (*recovery*), který obsahuje pouze 4 základní moduly (**rbe**, **kernel**, **syslib**, **bup_rcv**). Boot z oddílu `REC` lze vynutit v případě některých základních desek pomocí jumperu[10]. Další odlišností SPS od standardního ME je změna názvu `FTPR` na `OPR` (*operation*).[6]

Každý oddíl obsahující kód firmware má hlavičku CPD (*code partition directory*), která obsahuje název oddílu, počet souborů a pole struktur popisujících jednotlivé soubory. Tabulka 1.3 představuje zjednodušenou ukázku CPD.[6] Kód některých modulů předpokládá, že soubor `rbe` je bezprostředně následován souborem `rbe.met`.

Oddíly `FTPR` a `NFTP` obsahují soubory s kódem modulů (`rbe`, `kernel`...), metasoubory (`rbe.met`, `kernel.met`...) a manifest (`FTPR.man`, `NFTP.man`). Pro úsporu místa na flash paměti jsou soubory obsahující kód komprimovány. Ke kompresi je použito buďto Huffmanovo kódování (dekodér s tabulkami je součástí řadiče flash paměti[4]), nebo LZMA (dekomprese je prováděna modulem, který načítá komprimovaný modul).

Výzkumníci z firmy Positive Technology vytvořili nástroje `unME11`[11] a `unME12`[12] pro extrakci souborů z obrazu firmware. Také se jim podařilo zrekonstruovat tabulky pro dekódování modulů kódovaných Huffmanovým kódováním [13]. Jediným modulem, jehož kód není možné analyzovat, tak zůstává modul **pavp** (*protected audio video path*)[2], který je zašifrovaný.

Data modulů/aplikací jsou ukládána na flash paměť do oddílu `MFS`. Integrita a důvěrnost uložených dat není automaticky zajištěna a aplikace ukládající data si důvěrnost a integritu musí zajistit samy[2].

1.2.1.1 Manifest

Manifest CPD obsahuje základní informace o firmware obsaženém v daném oddíle. Hlavička manifestu specifikuje, mimo jiné, verzi firmware, architekturu, datum vytvoření nebo délku samotného manifestu:

```
struct manifest {
  int32_t a;               // +  0h (4)
  int32_t header_length;   // +  4h
  int32_t chipset;         // +  8h (10000h)
  int32_t x;               // +  Ch
  int32_t arch;            // + 10h (8086h)
  int32_t date;            // + 14h
  int32_t length;          // + 18h
  int32_t w;               // + 1Ch (324E4D24h)
  int32_t z;               // + 20h
  int16_t version[4];      // + 24h
  int32_t svn;             // + 2Ch
  //...
  int32_t b;               // + 78h (40h)
  int32_t c;               // + 7Ch (1)
  int8_t  modulus[256];    // + 80h
  int32_t exponent;        // +180h
  int8_t  signature[256]; // +184h
  int32_t d;
};
```

Součástí hlavičky je také podpis a veřejný klíč, kterým byl podpis vytvořen. Délky uvedené v manifestu, tedy délka hlavičky (`manifest.header_length`) a délka celého manifestu (`manifest.length`) jsou uvedené v double wordech. Hodnota `manifest.svn` (*security version number*) je slouží k ochraně před downgrade firmware[2].

Předtím, než se začne používat kód z daného oddílu, je provedena kontrola integrity a původu manifestu. Ta začíná kontrolou podpisu manifestu, která má čtyři kroky:

1. výpočet SHA-256 hashe veřejného klíče z manifestu a porovnání s hashem v ROM,
2. výpočet SHA-256 hashe hlavičky (bez veřejného klíče a podpisu) a těla manifestu,
3. šifrování hodnoty `manifest.signature` algoritmem RSA s veřejným klíčem z manifestu
4. porovnání hodnot z 2. a 3. kroku.

Pro úsporu místa v ROM, je v ní uložen pouze hash veřejného klíče, nikoliv celý klíč.[2]

Tělo manifestu je tvořeno ze sekcí (rozšíření). Jednotlivé sekce jsou vyhledávány pomocí 4B hodnoty (`id`) na začátku každé sekce. V dalším textu budou popsány některé sekce, které byly užitečné při analýze modulů. Význam jednotlivých polí byl identifikován na základě prezentace [6], souborů extrahových programem `unME` [11, 12] a informací získaných reverzním inženýrstvím modulů pracujících s manifestem.

Init script popisuje bootování ME. Sekce začíná hlavičkou s `id` rovným `1`. Hodnota `init_script.length` definuje počet struktur v poli následujícím po hlavičce:

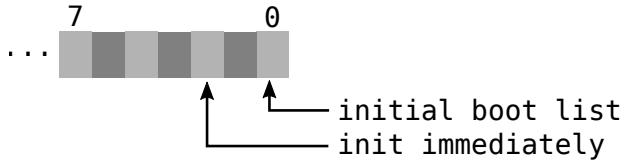
```
struct init_script {
  int32_t id = 1;
  int32_t sec_len;
  //...
  int32_t length;  // +Ch
};
```

Hodnota `init_script.sec_len` specifikuje délku celé sekce v bajtech. V poli se nechází následující struktury:

```
struct init_script_m {
  char    partition[4];
  char    module_name[12]; // + 4h
  int32_t info; // init?   // +10h
  int32_t boot;            // +14h
};
```

Každá struktura `init_script_m` obsahuje název modulu a parametry popisující způsob jakým bude spuštěn. V sekci Init script jsou uvedeny všechny moduly s výjimkou modulů **pm**, **syncman** a **vfs**, které jsou načítány modulem **bup**.

Podle kódu modulu **loadmgr** z firmware `E7B33IMS` je možné přiřadit význam dvěma bitům hodnoty `init_script_m.info`:



Obrázek 1.3: Význam bitů v sekci Init script

Partition info je podobně jako sekce Init script tvořena hlavičkou následovanou polem struktur. Hlavička obsahuje kromě identifikátoru a délky ještě hodnotu `partition_info.name` obsahují název oddílu (`FTPR`, `OPR`...):

```
struct partition_info {
  int32_t id = 3;
  char    name[4];
  int32_t size;
  //...
};
```

Hlavičku následuje pole, ve kterém má každý modul nacházející se v oddíle svůj záznam v podobě struktury:

```
struct partition_info_m {
  char    module_name[12];
  int8_t  x[4];
  int32_t metafile_size;
  int8_t  metafile_hash[32];
};
```

Obsah struktury slouží ke kontrole integrity modulu. Struktura nicméně neobsahuje hash modulu, ale hash metasouboru. Hodnotu je možné ověřit výpočtem SHA-256 hashe metasouboru extrahovaného nástrojem `unME`[11] a porotvnáním s hodnotou `partition_info_m.module_hash` uvedenou v odpovídající struktuře.

1.2.1.2 Metasoubory

Moduly nejsou spustitelné soubory. Nemají hlavičku, která by říkala, kam má být modul načten a na které adrese se nachází vstupní bod. Místo toho má každý modul metasoubor s koncovkou `.met`.[6] Metasoubor má podobnou strukturu jako manifest. Nemá hlavičku, ale je také tvořen ze sekcí, které jsou indexovány 4B hodnotou (`id`). V dalším textu budou popsány některé části, které byly potřeba při analýze modulů. Názvy byly voleny podle výstupu programu `unME`, přednášky [6] a reverzní analýzy některých modulů.

Process Manifest obsahuje základní informace o modulu. Mimo jiné je v ní uložena adresa (`priv_code_base_address`), na kterou má být modul načten a adresa vstupního bodu modulu (`main_thread_entry`):

```
struct process_manifest {
  int32_t id = 5;
  int32_t length;
  int8_t  flags;
  int8_t  padd[3];
  int32_t main_thread_id;
  int32_t priv_code_base_address;
  int32_t uncompressed_priv_code_size;
  int32_t cm0_head_size;
  int32_t bss_size;
  int32_t default_heap_size;
  int32_t main_thread_entry;
  int8_t  allowed_sys_calls[12];
  int16_t user_id;
  char    reserved[14];
  int16_t group_ids[];
};
```

Hodnota `process_manifest.length` udává celkovou délku struktury, protože pole `group_ids` nemá pevně stanovanou délku.

Module Attributes obsahuje informace užitečné v průběhu načítání modulu. V metasouboru je uveden hash dekomprimovaneho modulu, informace o kompresi a velikost modulu před a po kompresi:

```
struct module_attributes {
  int32_t id = 10;
```

```
    //...
    int8_t  comp;            // + 8h
    //...
    int32_t size;            // + Ch
    int32_t compressed_size; // +10h
    //...
    int8_t  module_hash[32]; // +18h
};
```

Dále specifikuje, zdali je modul komprimován. Pokud je hodnota `comp` rovna `1` je modul komprimován Huffmanovým kódováním, pokud je hodnota `comp` rovna `2` je modul komprimován pomocí LZMA. Při `comp=0` modul není komprimován.

1.3 Podobné technologie

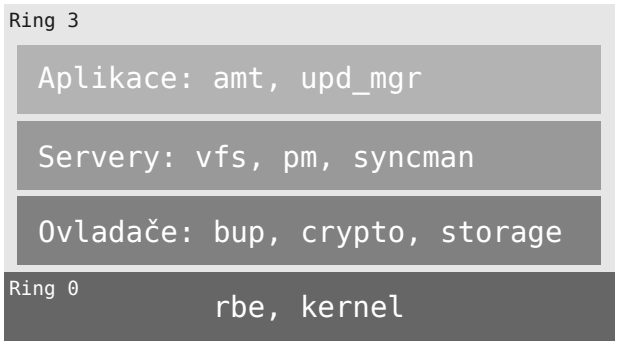
Některé serverové základní desky jsou osazeny BMC (*Baseboard Management Controller*)[14]. Jedním z výrobců podobných produktů je například ASPEED [15]. Účel BMC je stejný, jaký byl prvotní účel AMT, vzdálená správa systému bez asistence OS.

Některé chipsety společnosti Intel obsahují „druhý“ Intel ME nazývaný Intel Inovation Engine (IE). IE je k dispozici výrobcům, kteří staví své produkty na daných čipsetech a můžou na něm provozovat svůj firmware.[8, 5]

Kapitola 2

Architektura firmware Intel ME

Software Intel ME vychází z operačního systému MINIX 3[6]. Je rozdělen do vrstev, kde v nejnižší, privilegované vrstvě jsou pouze moduly **kernel** a **rbe**. Vrstvu nad jádrem tvoří ovladače a modul **bup**. V další vrstvě jsou potom servery a nejvyšší vrstvu tvoří uživatelské programy.[16] Obrázek 2.1 ilustruje jednotlivé vrstvy na několika modulech.



Obrázek 2.1: Vrstvy Intel ME

Většina modulů běží v usermode. Jedinou výjimku tvoří moduly **kernel** a **rbe**. Modul **rbe** je zavaděčem Intel ME. Modul **kernel** je jádrem Intel ME a zajišťuje plánování procesů, správu paměti[4].

Ovladače abstrahují hardwarová zařízení (DMA řadič, kryptografický koprocesor) a zprostředkovávají je ostatním procesům. Některé ovladače mohou pracovat pouze interně, zatímco jiné (LAN, WLAN, flash) mohou komunikovat i se zařízeními mimo Intel ME.[4] Příkladem ovladačů jsou moduly: **crypto**, **storage**, **gpio** nebo **heci**.

Servery, na rozdíl od ovladačů, poskytují komplexní funkcionalitu, která

může využívat různý hardware i software. [4] Příkladem jsou moduly: **evt-disp**, **vfs** nebo **pm**.

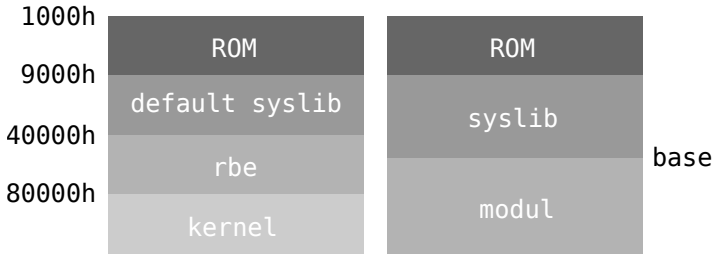
Aplikace obvykle využívají funkcí serverů a ovladačů a poskytují své služby hlavnímu procesoru nebo obsluhují požadavky přicházející po síti[4]. Příkladem aplikací jsou moduly: **upd_mgr** nebo **amt**. Modul **amt** má svůj vlastní ovladač síťové karty[2].

2.1 Paměťový prostor

Procesy **rbe** a **kernel** běží v kernel mode a mají vlastní dedikovanou knihovnu **syslib**, která je pravděpodobně načítána z ROM. Všechny ostatní moduly běží v userspace a mají totožné rozložení paměti (viz obrázek 2.2). Userspace moduly sdílí kód ROM a **syslib** – knihovny.[6]

Hodnota `base` v obrázku je specifikována hodnotou `process_manifest.priv_code_base_address` v metasouboru. V modulech jsou často prováděna volání na adresy `1000h` a `9000h`. Z výše uvedeného plyne:

- volání na adresy `1000h+` jsou voláním funkcí z ROM,
- volání na adresy `9000h+` jsou voláním funkcí ze **syslib**.



Obrázek 2.2: Rozvržení paměti Intel ME

2.2 Komunikace modulů

Velká část modulů Intel ME má v manifestu sekci Special file producer, která obsahuje seznam speciálních souborů, jež jsou modulem obsluhovány. Za tímto účelem má modul definouvanou funkci ve které je na základě čísla ioctl volání zavolána konkrétní funkce obsluhující požadavek.

Systémové volání `ioctl` obsluhuje modul **vfs**. Některá volání jsou ale obalena funkcemi ze **syslib**. Příkladem jsou kryptografické funkce, které mohou delegovat práci do modulu **crypto** právě skrze ioctl.

Speciální roli v komunikaci mezi moduly zastává modul **evtdips** (*event dispatcher*). Jednotlivé moduly se mohou v modulu **evtdisp** registrovat na nějakou událost. Jiný modul poté může danou událost vyvolat, čímž informuje registrované moduly.

2.3 Boot process

Dle dostupné dokumentace je při načítání systému ověřena integrita manifestu i načítaných modulů.[2] Cílem této části je popsat, jak a kým jsou jednotlivé moduly načítány. V dalších kapitolách je detailněji rozebrána funkcionalita modulů tykající se bootování a kontroly integrity systému. Obrázek 2.3 popisuje postup, jakým jsou jednotlivé moduly načítány:

1. bootloader z ROM zkontroluje, jestli je otisk klíče v manifestu shodný s otiskem v ROM. Pokud je klíč validní, zkontroluje se podpis manifestu. Pokud kontrola integrity manifestu selže, bude možné nabootovat operační systém, ale systém se automaticky po 30 minutách vypne[17]. Tento čas by měl být dostatečný pro obnovení firmware.

2. ROM ověří integritu modulu **rbe** pomocí otisku uloženého v manifestu a v případě úspěchu načte modul **rbe** do paměti a spustí jej. První a druhou fázi bootování nebylo možné v této práci ověřit, z důvodu nedostupnosti kódu z ROM.

3. modul **rbe** zajistí ověření integrity a načtení modulů **kernel**, **syslib**, **bup** a zavolá vstupní funkci modulu **kernel**.

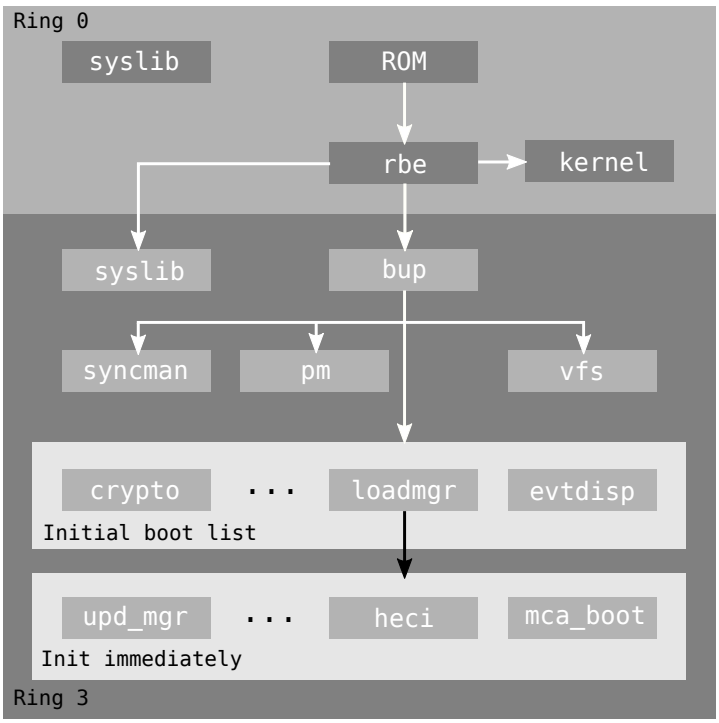
4. modul **kernel** provede nezbytnou inicializaci a spustí modul **bup**.

5. **bup** provede ověření integrity a spuštění modulu **vfs**.

6. dále jsou modulem **bup** načteny moduly, které mají v sekci Init script nastavený flag `initial boot list` a jejich jména jsou uložena do souboru `/snowball/ibl_list`.

7. dále jsou modulem **bupp** načteny a spuštěny moduly **pm** a **syncman**.

8. modul **pm** po svém spuštění a inicializaci iteruje přes názvy modulů uvedené v souboru `/snowball/ibl_list` a uvedené moduly spustí (více viz kapitola 3.4). Nakonec je modulem **pm** odeslána notifikace: `IBL load done`.



Obrázek 2.3: Závádění Intel ME

9. v návaznosti tuto zprávu jsou modulem **loadmgr** načteny zbylé moduly ze sekce Init script, které mají nastavený flag `init immediately`, ale nemají nastavený flag `initial boot list`.

V případě, že selže kontrola integrity některého z modulů, postupuje se následovně: pokud je modul *fault-tolerant*, načítání modulu se přeruší a pokračuje se dalším modulem; pokud je modul *non-fault-tolerant*, zastaví se načítání ME a vykoná se instrukce `hlt`.[2]

Kapitola 3

Moduly Intel ME

Celý systém software Intel ME je tvořen z modulů (aplikací). V této kapitole bude částečně rozebrána funkcionalita modulů, které souvisí se zaváděním systému a kryptografií. Kód modulu se v různých firmware může lišit. U každého modulu je proto uvedeno, z kterého firmware analyzovaný modul pochází.

Přednostně byla analýza prováděna na modulech, které obsahovaly ladicí výpisy, protože to velmi pomáhalo při porozumění kódu. Zároveň ale byl kód analyzovaných modulů porovnáván s kódem v jiných verzích s cílem ujistit se, že se výrazně neliší.

Každý modul je realizován jako dvojice souborů. První soubor mající název shodný s modulem obsahuje strojový kód modulu (například `rbe`). Druhý soubor se v tomto případě jmenuje `rbe.met` a obsahuje metadata souboru.

Každý modul (s výjimkou **syslib**) má vstupní bod `main_thread_entry` uvedený v metasouboru. Po spuštění modul provede inicializaci a následně ve většině případů zavolá funkci `main`. Příkladem modulu, který nemá funkci `main`, je modul **rbe** popsaný níže. Po inicializaci přejde většina modulů do stavu, kdy čeká na příchod požadavku (přerušení, ioctl, HECI).

3.1 ROM

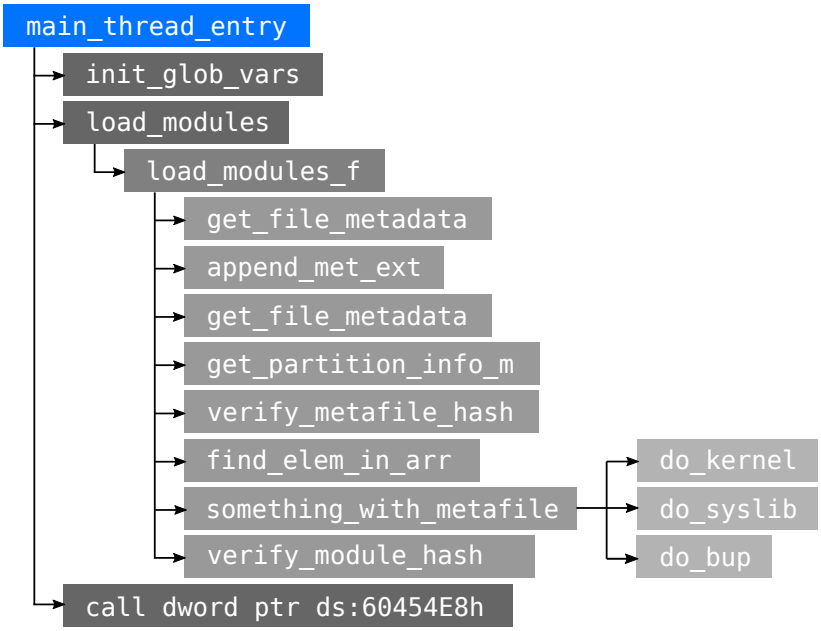
Kód z ROM není čistokrevný modul, ale zajišťuje zavedení systému Intel ME z paměti flash a posléze slouží jako knihovna. Bohužel nebyl kód z ROM k dipozici, takže nebylo možné analyzovat postup při kontrole integrity firmware. Nicméně v dokumentaci je jedna důležitá poznámka: ROM inicializuje interní paměť a zkopíruje do ní obraz firmware z flash [2].

Tabulka 3.1: Funkce knihovny ROM

adresa	název
1010h	`clock()`
102eh	`memcpy(char* ptr, char* ptr, int len)`
103dh	`memset(char* ptr, int n, char val)`
104ch	`strcmp(char* arr, char* str)`
1056h	`strlen(char* str)`
1060h	`strncmp(char* a, char* b, int len)`
1065h	`strncpy(char* dest, char* src, int n)`
106Fh	`strrchr(char* path, char c)`

Knihovna ROM by měla obsahovat pouze základní funkce ze standardní knihovny[2]. I bez přístupu ke kódu bylo možné odhadnout názvy některých funkcí (viz tabulka 3.1) z kontextu, předávaných argumentů a testů návratové hodnoty.

Některé verze firmware obsahují ROM jako modul nazvaný **ROMB** (*ROM bypass*), který pravděpodobně slouží k ladění firmware [6]. Během psaní práce jsem **ROMB** v žádném firmware nenašel. Podle prezentace [18] se ROMB liší svou funkcionalitou od ROM.



Obrázek 3.1: Důležitá funkční volání modulu **rbe**

3.2 Modul rbe

Analýza byla prováděna na modulu z firmware: `Latitude_7x90_1.0.1` [19] *a modulu z oddílu* `OPR` *firmware:* `P10S-I-ASUS-4401`[20]. *Konkrétní adresy se vztahují k této verzi firmware a v jiných verzích se můžou lišit.*

RBE je první načtený modul a je spouštěn kódem z ROM. Jeho účelem je, mimo jiné, načíst moduly **kernel**, **syslib** a **bup**. Součástí načítání by měla být kontrola integrity načítaných modulů. Pro lepší orientaci v dalším textu jsou na obrázku 3.1 zobrazena některá z funkčních volání, která modul provede. Poslední volání v obrázku vede na funkci `main_thread_entry` z modulu **kernel**.

3.2.1 Inicializace globálních proměnných

Oblast paměti počínaje adresou `6040000h` obsahuje globální proměnné modulu a je tedy sdílená napříč funkcemi. V kódu modulu se ke globálním proměnným přistupuje pomocí adres. Bez předchozí znalosti jejich významu bylo složité porozumět kódu, který s nimi pracoval. Relevantními adresami pro další text jsou:

```
6041008h ftpr.mft
6041010h bootpart - CPD start
6041014h fs
60410A8h metainformace modulu bup
6041330h metainformace modulu syslib
60454E4h metainformace modulu kernel
60454E8h adresa main_thread_entry modulu kernel
```

Podle použití konkrétních hodnot lze usuzovat, že na adresu `6041008h` je uložen ukazatel na obsah manifestu a na adresu `6041010h` počáteční adresa CPD. Hodnota `fd` na adrese `6041014h` představuje ukazatel na pole struktur:

```
struct file {
  char[12] filename;
  int32_t  offset;
  int32_t  size;
  int32_t  y;
};
```

které reprezentují soubory v oddíle, ze kterého se bootuje. V přednášce [6] je pravděpodobně tato struktura zmíněna na slidu 12. Hodnota `file.offset`

se vztahuje k hodnotě `bootpart`. Pole struktur je pravděpodobně součástí obrazu firmware, který byl zkopírován kódem ROM z flash paměti.

3.2.2 Načítání modulů

Funkce načítající moduly je volána přímo z funkce `main_thread_entry`. Načítání modulů probíhá v cyklu. Funkce `load_modules_f` iteruje přes pole struktur:

```
struct module {
  char*   name; // nazev nacitaneho modulu
  int32_t num;  // poradove cislo souboru s~modulem
};
```

v datové části binárního souboru, které obsahuje informace týkající se tří modulů:

```
ptr -> "kernel"
3
ptr -> "syslib"
5
ptr -> "bup"
7
```

V každé iteraci jsou zavolány funkce, které provedou načtení a validaci metasouboru, jeho parsování a načtení a validaci modulu.

První volanou funkcí je `get_file_metadata(long num, char* name)`. Jako argumenty dostane hodnoty `module.num` a `module.name`. Parametr `num` je použit jako index do pole struktur na adrese `6041014h` a provede kontrolu jestli se název souboru ve strukture shoduje s parametrem `name`. Pokud se názvy shodují vrátí funkce ukazatel na daný `struct file` v opačném případě vrátí nulu.

V dalším kroku je inkrementována hodnota `module.num` a zavolána funkce `append_met_ext(char* src, char* dst)`, která připojí za název modulu koncovku `.met` a výsledek uloží do pole specifikovaného parametrem `dst`. S takto upraveným názvem a inkrementovanou hodnotou `module.num` je opět zavolána funkce `get_file_metadata`, která tentokrát nalezne metadata metasouboru načítaného modulu.

Dále je z manifestu načtena sekce Partition info odpovídající právě načítanému modulu. Hodnota `partition_info_m.metafile_hash` je použita jako parametr funkci:

```
int32_t verify_metafile_hash(
  char*    out_ptr,      // buffer pro obsah metasouboru
  char*    file_data,    // ukazatel na obsah souboru
  int32_t  file_size,    // velikost souboru
  char*    expected_hash // hash metasouboru z~manifestu
);
```

Funkce sama o sobě neprovádí výpočet hashe a modul ani neobsahuje konstanty potřebné pro výpočet SHA-256. Pro výpočet hashe je použita funkce ze **syslib** (viz sekce 3.7).

Následuje volání funkce:

```
process_metafile(
  int32_t                   i,       // číslo iterace
  char*                     module,  // obsah modulu
  char*                     metafile,// obsah metasouboru
  int32_t                   metafile_size,
  struct module_attributes* attr,
);
```

jejímž úkolem je parsovat metasoubor a v závislosti na parametru `i` vykonat konkrétní větev pro modul **bup**, **syslib** nebo **kernel**. Na konci každé větve je potom volání funkce `do_bup`, `do_syslib` nebo `do_kernel`.

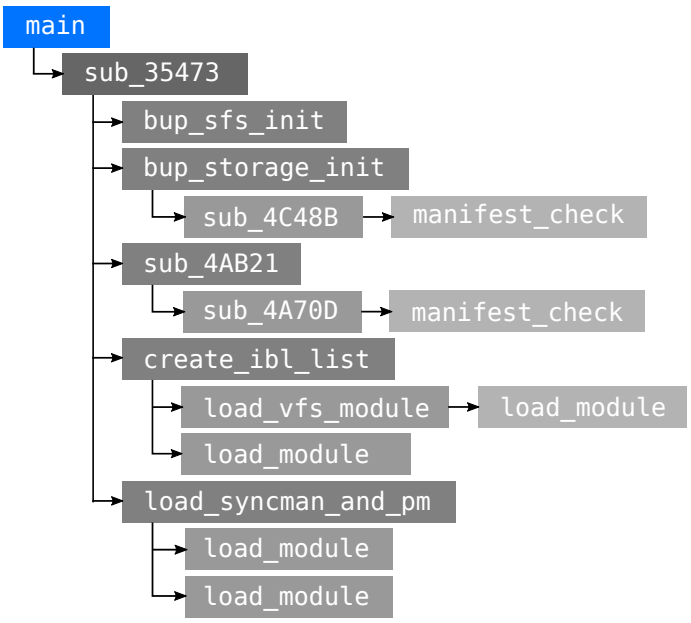
Funkce `do_bup` parsuje metasoubor a konkrétní hodnoty uloží do struktury na adrese `60410A8h`. Tato konkrétní adresa je poté předána jako jediný parametr funkci `main_thread_entry` z modulu **kernel**. Podobně fungují i další dvě funkce, `do_kernel` a `do_syslib`, parsují metasoubor a získané hodnoty ukládají na pevně stanovené adresy v paměti.

Jako poslední funkce smyčky je volána funkce `verify_module_hash`. Funkce nejprve načte modul, vypočítá jeho hash a hodnotu ověří vůči hodnotě z metasouboru. Pokud funkce `load_module` skončí chybou, tedy selže načítání některého z modulů, nespustí se **kernel** a tedy ani **bup** a nakonec je provedena instrukce `hlt`.

V modulu je často používaná funkce jejímž účelem je najít v poli `ptr` dlouhém `len` první výskyt hodnoty `val` a vrátit ukazatel na tuto hodnotu:

```
find_elem_in_arr(int32_t* ptr, int32_t len, int32_t val);
```

Pokud hodnotu nenalezne, vrátí 0. Často se používá pro hledání jednotlivých sekcí v manifestu a metasouborech. Z kódu není jasné, jak mají vývojáři ošetřené, aby se některá hodnota v manifestu/metasouboru neshodovala s identifikátorem konkrétní sekce.



Obrázek 3.2: Důležitá funkční volání modulu **bup**

Modul **rbe** už nekontroluje integritu manifestu, protože lze předpokládat, že to udělal kód z ROM. I bez analýzy funkce provádějící hashování je možné říci, že musí vracet validní SHA-256 hash shodující se s referenčním hashem v manifestu/metasouboru. Pokud by se hashe neshodovaly, modul by přerušil zavádění systému.

3.3 Modul bup

Analýza byla prováděna na modulu z firmware: `P10S-I-ASUS-4401`[20] *z oddílu:* `OPR`.

Modul **bup** je možné přirovnat k procesu `init`. Některá z funkčních volání, důležitých pro tuto práci, jsou pro lepší orientaci v textu vyobrazena na obrázku 3.2. V modulu **bup** je implementována část ovladače pro práci s hardwarovým kryptografickým modulem. Funkce `main` volá funkci `sub_35473`, která iteruje přes pole struktur:

```
struct functions {
  int32_t  len1;  // délka pole arr1
  int32_t* arr1;  // pole ukazatelů na funkce
  int32_t  len2;  // délka pole arr2
  int32_t* arr2;  // pole ukazatelů na funkce
```

```
  int32_t  ord;}; // pořadové číslo
```

v datové části binárního souboru. Každá struktura obsahuje dvě pole ukazatelů na funkce a informaci o jejich délce. V každé iteraci jsou nejprve volány všechny funkce z pole `functions.arr1` a poté z pole `functions.arr2`. Jednotlivé funkce zajišťují mimo jiné:

- kontrolu manifestu,
- načtení modulu **vfs**,
- vytvoření souboru `/snowball/ibl_list`,
- načtení modulů **pm** a **syncman**.

Modul ještě před samotným zavoláním funkce `sub_35473` načte některé vyjmenované soubory a uloží jejich obsah do spojového seznamu reprezentovaného strukturami:

```
struct link_list_node {
  char                  filename[12];
  //...
  int32_t               data; // +18h
  struct link_list_node* next; // +1Ch
};
```

Tyto soubory jsou poté dostupné ostatním funkcím bez nutnosti jejich otevírání.

3.3.1 Kontrola manifestu

Kontrolu manifestu provádí funkce:

```
int32_t manifest_check(
  char*   manifest,
  int32_t pubkey_hash,
  int32_t debug_pubkey_hash,
  int32_t y
);
```

Jako první je kontrolována hlavička manifestu. Význam některých hodnot není jasný, nicméně hlavička musí splňovat následující podmínky:

- `manifest.w` musí obsahovat hodnotu `324E4D24h`,

- `manifest.arch` musí obsahovat hodnotu `8086h`,
- `manifest.a` musí obsahovat hodnotu `4h`,
- `manifest.chipset` musí obsahovat hodnotu `10000h`,
- `manifest.header_length` musí obsahovat hodnotu `A1h`,
- `manifest.b` musí obsahovat hodnotu `40h`,
- `manifest.c` musí obsahovat hodnotu `1h`,
- velikost těla manifestu bez hlavičky nesmí být větší než `75Fh`.

Pokud není některá z výše uvedených podmínek splněna, funkce kontrolující manifest skončí s chybou.

V dalším kroku je provedena kontrola integrity manifestu, která se skládá z následujících kroků:

1. kontrola původu veřejného klíče,
2. kontrola podpisu manifestu,
   a) zašifrování hashe z manifestu,
   b) výpočet hashe hlavičky manifestu a těla manifestu,
   c) porovnání hashů.

Jako první se kontroluje hash veřejného klíče. Pro výpočet se používá funkce `hash_verify` z modulu **syslib**. Vypočítaná hodnota hashe musí být shodná s hodnotou předanou skrze parametr `pubkey_hash`. V případě neshody hashe je provedeno porovnání s druhým hashem předaným jako parametr `debug_pubkey_hash`. Hodnoty referenčních hashů jsou součástí binárního souboru modulu **bup**.

Druhým krokem je kontrola podpisu manifestu. Nejprve je zašifrován hash z manifestu. K šifrování je použita interní implementace ovladače hardwarového kryptografického modulu. Následuje výpočet hashe hlavičky manifestu. Hashuje se pouze prvních `80h` bytů hlavičky (bez veřejného klíče) a je k tomu použita funkce `hash_data_update` z modulu **syslib**. Délka hlavičky předávaná hashovací funkci je daná konstantou, která je součástí kódu.

Dále se počítá hash těla manifestu. K výpočtu počáteční adresy je použita hodnota `manifest.header_length` z hlavičky manifestu, která ještě v danou chvíli není validována. Pro výpočet hashe se používá stejná funkce

jako v případě hlavičky. Na závěr hashování se volá funkce `hash_data_final` ze **syslib**.

Výsledný hash se porovnává s hashem z manifestu, který byl získán z funkce `verify_rsa_encrypt` ze **syslib**. Na závěr je provedena kontrola paddingu doplňujícího hash manifestu.

3.3.2 Načítání modulů

Načtení konkrétního modulu, specifikovaného jménem, zajišťuje funkce:

```
int32_t load_module(
  char*   name, // např. "vfs"
  int32_t flag,
  int32_t  a,
  int32_t  b,
  int32_t  c
);
```

která nejprve ověří integritu odpovídajícího metasouboru a následně metasoubor parsuje na jednotlivé sekce. Ve druhé části funkce je prováděno načítání modulu a jeho dekomprese. Parametr `flag` pravdepodobě určuje, jestli bude modul pouze načten pro použití dalšími moduly, nebo také spuštěn.

Funkce nejprve vyhledá obsah manifestu uložený ve spojovém seznamu a nalezne v něm sekci Partition info. Následně je odvozen odpovídající název metasouboru z parametru `name`. Následuje volání funkce `load_metafile`:

```
int32_t load_metafile(
  char*     metafile_name, // název metasouboru
  char*     buffer,        // obsah metasouboru
  int32_t   num,           // 0
  int32_t   x,
  struct a* info,          // obsahuje hash metasouboru
  int32_t*  metafile_len
);
```

starající se o načtení specifikovaného metasouboru a vypočítání jeho hashe. Zajímavé je, že se pro výpočet hashe používá funkce `938Eh` ze **syslib**, která nevolá žádnou z dříve identifikovaných funkcí spojených s výpočtem SHA-256. Nezbytná velikost bufferu je získána z manifestu, ze záznamu `partition_info_m.metafile_size` patřícího načítanému modulu. Následuje porovnání vypočítaného hashe s hashem z manifestu.

V další fázi probíhá parsování obsahu metasouboru na jednotlivé sekce. Poustupně jsou extrahovány sekce:

- Module attributes,
- Process manifest,
- Threads,
- Special File Producer,
- MMIO ranges.

Ve třetí fázi je po stránkách velkých 4 kiB načten kód modulu. Každá stránka je dekomprimována odpovídajícím algoritmem a hashována algoritmem SHA-256. Pro hashování nejsou použity funkce ze **syslib** jako v případě ostatních modulů, ale interní ovladač kryptografikého hardware.

Na konci funkce je, v závislosti na parametru `flag`, načtený modul buď spuštěn, nebo je pouze uložen načtený metasoubor do složky `/snowball/`.

3.3.3 IBL list

Funkce `create_ibl_list` nemá žádný parametr a je zavolána během inicializace modulu **bup**. Na začátku funkce je provedena kontrola, zdali již byla funkce spuštěna. Pokud již byla jednou provedena vícekrát už se neprovádí. Dále proveden test, jesli již byl načtený modul **vfs**, v případě že načten nebyl, je zavolána funkce `load_vfs_module`.

V další fázi je v manifestu vyhledána sekce Init script, přes kterou je následně iterováno. V případě, že záznam o modulu splňuje kladené podmínky, je na něm zavolána funkce `load_module` s parametrem `flag=0` a do bufferu je přidán název modulu.

Co se týče zmiňovaných podmínek, ty nejsou napevno stanovené v kódu funkce, ale jsou zjišťovány za běhu. Po projití celé sekce `Init Script` je obsah bufferu se jmény uložen do souboru `/snowball/ibl_list`.

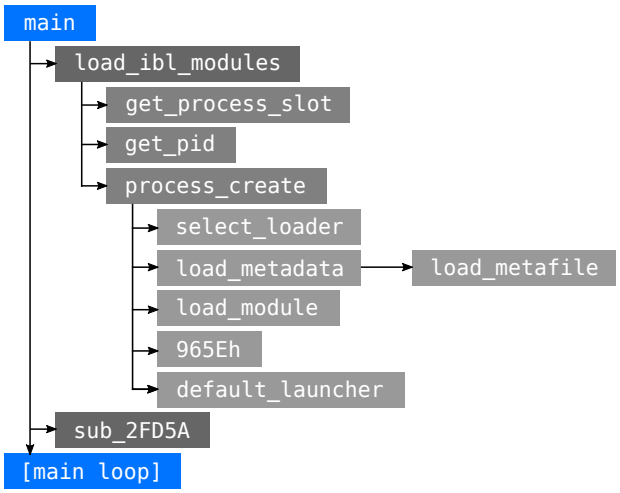
3.4 Modul pm

Analýza byla prováděna na modulech z firmware: `Latitude_7x90_1.0.1`[19] *a* `P10S-I-ASUS-4401`[20].

Jako jeden z mála je tento modul podobný modulu **pm**, který je součástí zdrojového kódu operačního systému MINIX 3. Svědčí o tom mimo jiné shodný ladící výpis:

```
PM got message from invalid endpoint
..\..\src\os\servers\pm\main.c
```

Funkce `main` je v případě Intel ME rozšířena o kód zajišťující spuštění modulů zapsaných v souboru `/snowball/ibl_list`. Na obrázku 3.3 je vykreslený strom funkčních volání, která provede funkce `main`. Blok `main_loop` je už součástí originálního **pm** a obsahuje smyčku obsluhující požadavky.



Obrázek 3.3: Důležitá funkční volání modulu **pm**

Funkce `main` zavolá funkci `load_ibl_modules`, která iteruje přes obsah souboru `/snowball/ibl_list`. Podle kódu v modulu **pm** obsahuje `ibl_list` pole názvů modulů zarovnaných na 12 B. To, které moduly jsou uvedeny v souboru, je definováno v manifestu. Bežně jsou to moduly: **kernel**, **syslib**, **rbe**, **bup**, **evtdisp**, **busdrv**, **prtc**, **crypto**, **storage** a **loadmgr**. Je možné, že v dalších verzích firmware bude seznam modulů vypadat poněkud odlišně. Na pořadí modulů uvedených v souboru záleží. Například modul **storage** je závislý na modulu **crypto** a **loadmgr** vyžaduje modul **evtdisp**.

V každé iteraci je ze souboru `ibl_list` načten název modulu a vytvořena cesta `/snowball/<nazev modulu>`. Následuje volání funkcí:

```
int32_t get_process_slot(int32_t* slot, int32_t);
int32_t get_pid(int32_t slot);
```

Poslední volanou funkcí v iteraci, důležitou pro tuto práci, je:

```
int32_t process_create(
  struct info* proc_info,
```

```
  int32_t      pid,
  int32_t      addr,
  int32_t      zero1,
  int32_t      zero2
);
```

Parametr `proc_info` je ukazatel na strukturu:

```
struct info {
  int32_t loader;
  char*   path;
};
```

Hodnota `info.loader` specifikuje loader, který se má při načítání modulu použít. V obou analyzovaných modulech se ve všech případech používá loader s `id=0`. Hodnota `info.path` obsahuje cestu k binárnímu souboru modulu. Jako první volá funkce `process_create` funkci `select_loader(struct info*)`, která vrací ukazatel na strukturu:

```
struct loader {
 int32_t (*get_id)();
 int32_t (*load_metadata)
   (struct info*, int32_t, int32_t, char*);
 int32_t (*load_module)
   (struct info*, int32_t, char*, int32_t);
 int32_t (*default_launcher)();
};
```

obsahující ukazatele na čtyři funkce. Analyzované moduly obsahovaly ve své datové části dvě takové struktury. Která struktura je zvolena závisí na hodnotě `info.loader`. Funkce `loader.get_id` pouze vrací číslo, v případě první struktury `1`, v případě druhé struktury `0`.

Následuje načtení metasouboru pomocí funkce `loader.load_metadata` z vybraného loaderu (viz následující sekce) a jeho parsování přímo ve funkci `process_create`.

Další volanou funkcí je `loader.load_module`, která načte obsah modulu do paměti. Po načtení modulu je zavolána funkce `965Eh` (`process_create`) z knihovny **syslib**. Na závěr je zavolána funkce `loader.default_launcher`.

Po návratu z funkce `process_create` následuje kód, který obsahuje speciální větve pro moduly:

1. **evtdisp** – po načtení modulu se u něj **pm** registruje

2. **loadmgr** – po načtení modulu uloží jeho PID na hardcoded adresu

3. **tty** – nic se neprovede, pravděpodobně protože analyzovaný firmware neobsahuje modul **tty**

Při opouštění funkce `load_ibl_list` se kontroluje pouze to, zdali byl načten modul **evtdisp**. Pokud načten nebyl, zavolá se `panic_suicide` a načítání ME je ukončeno. Načtení jiných modulů se nekontroluje. Na konci funkce `load_ibl_list` se notifikuje event dispatcher.

Funkce `load_ibl_list` nevrací žádnou návratovou hodnotu, je součástí sekvence volání:

```
2E238   call    load_ibl_modules
2E23D   call    dummy            ; funkce nic nedělá
2E242   call    sub_2FD5A
2E247 main_loop:
      ...
```

provedených před vstupem do hlavní smyčky.

3.4.1 Načtení metasouboru

Pro načtení metasouboru je ve funkci `process_create` zavolána funkce `load_metadata` ze zvoleného loaderu:

```
int32_t load_metadata(
  struct info* info,
  char*        addr,   // buffer pro metasoubor
  int32_t      len,    // délka bufferu
  int32_t*     out_len // délka načteného metasouboru
);
```

která následně zavolá funkci:

```
int32_t load_metafile(
  char*   path,  // cesta z~info.path
  char*   addr,  // buffer pro metasoubor
  int32_t len,   // délka bufferu
  char*   hash1, // buffer pro hash
  char*   hash2  // buffer pro hash
);
```

V případě loaderu s `id` rovným jedné jsou na závěr funkce `load_metadata` vzájemně porovnána dvě pole, jejichž adresy byly předány skrze parametry `hash1` a `hash2` funkci `load_metafile`. Pokud se obsah polí neshoduje, je zavolána funkce `panic_suicide`. Při shodě hashů vrací funkce `load_metadata` hodnotu `0`. V případě loaderu s `id` rovným nule se hashování metasouboru neprovádí.

Ve funkci `load_metafile` je nejprve, ve staticky alokovaném bufferu, sestavena cesta k metasouboru (k parametru `path` je připojena přípona `.met`). Buffer je dostatečně velký a nemůže dojít k přetečení. Následně se funkce dělí do dvou větví:

1. pokud je parametr `hash1` roven `0`, potom je metasoubor otevřen a jeho obsah načten do bufferu na adrese uložené v parametru `addr`. Hash metasouboru není počítán.

2. jestliže parametr `hash1` není roven 0, potom je metasoubor otevřen a pomocí funkce `90CD` je načten jeho obsah do pole odkazovaného parametrem `addr`. Velikost bufferu je dána parametrem `len`.

   Následuje volání funkce `90DC`. Podle ladících výpisů je účelem funkce výpočet hashe souboru. Vstupem není obsah metasouboru, ale file-descriptor otevřeného metasouboru. Druhým vstupem je buffer pro hash. Pro výpočet SHA-256 hashe se použije ioctl příkaz zavolaný na otevřeném metasouboru.

Stejným způsobem lze obejít také počítání `hash2`. Pokud ale `hash2 != 0` potom je z parametru `path` vyříznut název modulu a používá se pro hledání souboru s názvem `<module_name>.inf`.

3.4.2 Načtení modulu

Pro načtení kódu modulu a ověření jeho integrity se používá funkce:

```
load_module(
  struct info* info,
  int32_t      a,
  char*        metafile,    // obsah metasouboru
  int          metafile_len
);
```

ze zvoleného loaderu. V případě loaderu s `id` rovným nule funkce obsahuje pouze vypsání chybové hlášky:

```
ibl_load_exec called although IBL processes shouldn't
be loaded by pm
```

a zavolání funkce `panic_suicide`. Je možné, že ve starších verzích firmware byly moduly s příznakem initial boot list spouštěny modulem **pm**.

U loaderu s `id` rovným jedné se fragmenty kódu funkce `load_module` velmi podobají částem kódu funkce `load_module` z modulu **bup**. Nejprve jsou ze sekce Module attributes z metasouboru obsaženého v parametru `metafile` načteny hodnoty `csise`, `size` a `comp`.

V další části funkce se postupně načítá, po stránkách velkých 4 KiB, kód modulu (v případě LZMA komprese je dekompresimován). Po dekompresi je stránka přidána do adresního prostoru vytvářeného procesu. Po načtení kódu modulu se provede výpočet hashe modulu. K tomuto účelu se využívá:

1. funkce `90DCh` ze **syslib** v případě že byl modul komprimován pomocí LZMA,

2. funkce `hash_data_update` a `hash_data_final` ze **syslib** v případě, že byl modul komprimován pomocí Huffmanova kódování.

Na konci funkce je provedenoa porovnání hodnoty `module_attributes.module_hash` s hodnotou hashe z metasouboru a uvolnění prostředků. Pokud kontrola hashe selže vrací funkce `load_module` nenulovou hodnotu.

3.5 Modul loadmgr

Analýza byla prváděna na modulech z firmware: `P10S-I-ASUS-4401`[20].

Ve funkci `main` modulu **loadmgr** (*load manager*) je inicializována odezva na notifikace přicházející od modulu **evtdisp**.

3.5.1 Kontrola manifestu

Modul poskytuje ioctl volání `ioctl_verify_manifest`. Vstupem funkce je velikost manifestu a jeho obsah. Jako první je kontrolována hlavička manifestu. Musí splňovat podobné podmínky jako ve funkci `manifest_check` modulu **bup**:

- `manifest.w` musí obsahovat hodnotu `324E4D24h`,
- `manifest.arch` musí obsahovat hodnotu `8086h`,
- `manifest.a` musí obsahovat hodnotu `4h`,

- `manifest.chipset` musí obsahovat hodnotu `10000h`,
- `manifest.header_length` musí obsahovat hodnotu `A1h`,
- `manifest.b` musí obsahovat hodnotu `40h`,
- `manifest.c` musí obsahovat hodnotu `1h`,
- velikost hlavičky musí být menší než celková velikost manifestu.

Pokud není některá z výše uvedených podmínek splněna, funkce kontrolující manifest skončí s chybou. Následně jsou hodnoty `manifest.modulus` a `manifest.exponent` porovnány s hodnotami v paměti.

Dále je alokován prostor pro klíč RSA a výsledek dešifrování. U modulu **bup**, analyzovaném v předchozím textu, byly ukazatele získané funkcí `malloc` rovnou použity. V modulu **loadmgr** je u každého jednotlivého ukazatele otestováno, zdali alokace neselhala a funkce `malloc` nevátila nulovou hodnotu.

V dalším kroku je provedeno dešifrování podpisu manifestu. Je k tomu použita funkce `rsa_encrypt` z modulu **syslib** a klíč z manifestu v podobě hodnot `manifest.exponent` a `manifest.modulus`.

Následuje dvoufázové hashování manifestu funkcí `hash_data_update`. Nejprve je hashována hlavička manifestu (bez klíče) a posléze tělo manifestu. Hash získaný voláním funkce `hash_data_final` je porovnán s prvními 32 bajty dešifrovaného podpisu. Zbytek podpisu (*padding*) je porovnán s hodnotou v datové části binárního souboru modulu.

Na závěr funkce je provedena dealokace všech proměnných. Pokud kontrola manifestu v některé fázi selže, vrací funkce nenulovou hodnotu. Při úspěchu vrací `0`.

3.5.2 Načítání modulů

Modul **loadmgr** poskytuje ioctl volání `ioctl_start_load_process`, která inicializuje načítání modulů. V manifestu je nalezena sekce Init script a adresa pole struktur `init_script_m` je uložena do globální proměnné. Zároveň je nastavena další globální proměnná na `1` – příznak, že byl proces načítání započat.

Načítání modulů má na starosti funkce `start_process`. Po každém zavolání funkce načte první strukturu `init_script_m` z globální proměnné a posune ukazatel na následující strukturu. Pokud je v získané struktuře nastaven příznak `init immediately` a není nastaven bit `initial boot list`

je proces spuštěn. Pokud je modul spuštěn nebo v dané struktuře nebyly nastavené požadované flagy vrací funkce `1`. Pokud se proces nepodařilo spustit nebo již byl projit celý Init script je vrácena hodnota `0`.

Spuštění procesu zajišťuje funkce `spawn_process(char* module_name)`, ve které je z argumentu sestavena cesta k binárnímu souboru modulu ve složce `bin`. Do složky `bin` je mapován obsah CPD[21].

3.6 Modul crypto

Analýza byla prováděna na modulech z firmware: `P10S-I-ASUS-4401`[20].

Kryptografický modul (**crypto**) je ovladač, který abstrahuje přístup k hardwarovému kryptografickému modulu pro ostatní aplikace. Mimo to také implementuje algoritmy, které nejsou v harwaru podporované.[4] Díky ladícím výpisům bylo možné indentifikovat názvy většiny ioctl volání, jež ovladač poskytuje prostřednictvím souboru `/dev/crypto`.

V souladu s dokumentací[2] podporuje kryptografický modul:

- šifry RSA, RC4, 3DES a AES,
- hashovací funkce MD5, SHA-1 a SHA-2,
- aritmetiku s velkými čísly,
- eliptické křivky a
- úložiště pro klíče.

Většina modulů využívá pro přístup k modulu **crypto** funkce ze **syslib**, které obalují ioctl volání. Navíc **syslib** implementuje algoritmy, které jsou potřeba ještě načtením modulu **crypto**.

3.6.1 Symetrické šifry

Ovladač podporuje tři symetrické šifrovací algoritmy. Proudovou šifru RC4 a blokové šifry 3DES a AES. Pro tento účel je definováno 8 ioctl volání (viz tabulka 3.2). Volání `4h` a `5h` nemají žádný efekt, pouze vrací hodnotu `5`.

Šifra RC4 je implementována v hardware a funkce `crypto_drv_rc4` pouze předává deleguje zpracování. Šifra 3DES je implementována v software modulu **crypto**. Každé z ioctl volání slouží jak k šifrování, tak k dešifrování v jednom z módů: ECB, CBC nebo CTR. Výše popsané informace jsou v souladu s dostupnou literaturou [2].

Tabulka 3.2: Symetrické šifry modulu **crypto**

ioctl	funkce
0h	crypto_drv_aes_ecb
1h	crypto_drv_aes_cbc
2h	crypto_drv_aes_ctr
3h	crypto_drv_aes_cfb_ofb
6h	crypto_drv_3des_ecb
7h	crypto_drv_3des_cbc
8h	crypto_drv_3des_ctr
9h	crypto_drv_rc4

U šifry AES je k dispozici šifrování a dešifrování v módech ECB, CBC a CTR. V hardware jsou implementované varianty AES-128 a AES-256. AES-192 je implementovaný v software modulu. Autoři kódu použili optimalizaci pro 32b architektury pomocí t-tabulek.[22]

Čtvrtá funkce (`crypto_drv_aes_cfb_ofb`) pravděpodobně šifruje módy OFB případně CFB[2]. Nicméně je implementovaná pouze v hardware a proto není možné hypotézu ověřit. Funkce, které zajišťují delegaci na hardware obsahují konstanty z testovacích sad NIST [23]:

```
9CF4F3Ch, 0ABF71588h, 28AED2A6h, 2B7E1516h
```

Jejich použití je dáno hodnotou globální proměnné, která ale není nastavena uvnitř modulu.

3.6.2 Hashovací funkce

Z hashovacích funkcí jsou hardwarově akcelerované tři. Podle použitých konstant (velikosti bloků) lze usuzovat, že v hardware jsou implementované algoritmy: SHA-1, SHA-256 a MD5. Modul dále obsahuje softwarovou implementaci algoritmů SHA-224, SHA-384 a SHA-512. Tabulka 3.3 obsahuje názvy ioctl volání pracujících s hashovacími algoritmy.

Pro hashování jsou k dispozici funkce:

- `crypto_drv_hash_simple` – jednorázové hashování souvislého bloku dat,
- `crypto_drv_hash_data` – hashování nesouvislého bloku dat pomocí opakovaného volání.

Každý hashovací algoritmus je interně reprezentován konstantou (viz tabulka 3.4), která se shoduje s označením v modulu **syslib**.

Tabulka 3.3: Hashovací funkce modulu **crypto**

ioctl	funkce
Ah	crypto_drv_hash_simple
Bh	crypto_drv_hash_data
Ch	crypto_drv_hash_hmac_simple
Dh	crypto_drv_hash_hmac
Eh	crypto_drv_dma_and_hash

Tabulka 3.4: Dostupné hashovací funkce

id	algoritmus
0	SHA-1
1	SHA-256
2	MD5
3	SHA-224
4	SHA-384
5	SHA 512

Kromě hashování podporuje kryptografický modul také výpočet HMAC. Funkce `crypto_drv_hash_hmac` a `crypto_drv_hash_hmac_simple` podporují pouze hardwarově akcelerovaný výpočet.

Účel funkce `crypto_drv_dma_and_hash` nebylo možné zjistit, protože je implementovaná pouze v hardware. Mohla by ale sloužit k hashování dat v paměti DRAM.

3.6.3 Aritmetika s velkými čisly

Tabulka 3.5: Aritmetické funkce modulu **crypto**

ioctl	funkce
10h	crypto_drv_big_number_add
11h	crypto_drv_big_number_subtract
12h	crypto_drv_big_number_multiply
13h	crypto_drv_big_number_mod
14h	crypto_drv_big_number_exp_mul_mod
15h	crypto_drv_big_number_mont_mul
16h	crypto_drv_big_number_divide
17h	crypto_drv_big_number_inverse_mod
18h	crypto_drv_big_number_prime_test

Funkce `crypto_drv_big_number_add` sečte dvě čísla jejichž délka je specifikována argumentem funkce. Sčítání je realizované v softwaru jednou smyčkou. Výstupem funkce je výsledek součtu a jeho délka.

Funkce `crypto_drv_big_number_subtract` má stejný vstup jako funkce pro součet, je také realizována softwarově ale její výstup tvoří trojice hodnot: pole s výsledkem, délka výsledku a znaménko.

Funkce `crypto_drv_big_number_divide` je implementována v software. Výs-tupem jsou podíl a zbytek. Operandy mohou být maximálně 512 B dlouhé.

Funkce `crypto_drv_big_number_inverse_mod` je implementována v software. Jejím účelem je najít inverzi modulo. Operandy mohou být dlouhé maximálně 512 B.

Funkce `crypto_drv_big_number_prime_test` testuje, zdali je předaný argument prvočíslo. Nejprve zkouší dělitelnost prvními 2048 prvočísly, která jsou součástí binárního souboru, a poté provádí Miller-Rabinův test. Souhlasí s tím, co je uvedeno v dostupné literatuře[2].

Ostatní funkce z této části (viz tabulka 3.5) jsou implementovány v hardware a každý operand může mít maximálně 256 B.

3.6.4 Asymetrická kryptografie

Modul **crypto** poskytuje funkce pro šifrování a dešifrování algoritmem RSA, validaci klíče RSA a generování klíče RSA. V tabulce 3.6 jsou uvedeny jednotlivé funkce a jim odpovídající číslo ioctl.

Tabulka 3.6: Asymetrické šifry modulu **crypto**

ioctl	funkce
1Bh	`crypto_drv_rsa_raw_enc_dec(...,struct ioctl_args*)`
1Ch	`crypto_drv_rsa_gen_key_start`
1Dh	`crypto_drv_rsa_gen_key_get`
1Eh	`crypto_drv_rsa_gen_key_abort`
1Fh	`crypto_drv_rsa_validate_key`

Funkce `crypto_drv_rsa_raw_enc_dec` má teoreticky až šest vstupních parametrů: `key_n`, `key_e`, `key_p`, `key_d`, `key_q` a `input_data`. Informace o ioctl argumentech funkce jsou udržovány ve struktuře:

```
struct ioctl_args {
  int8_t enc_or_dec;      // + 0h
  //...
  int32_t key_n_len;      // + 8h
```

```
  int32_t key_e_len;      // + Ch
  int32_t key_p_len;      // +10h
  int32_t key_q_len;      // +14h
  int32_t key_d_len;      // +18h
  int32_t input_data_len; // +1Ch
};
```

Vždy jsou načteny argumenty `input_data`, `key_n`, `key_e` a alokován buffer pro výstup `data_out`.

Šifrování (mocnění) je hardwarově akcelerované. Vzhledem k tomu, že je použitá stejná funkce jako ve funkci `crypto_drv_big_number_mont_mul`, používá se pravděpodoně algoritmus *Montgomery ladder*.

Pro dešifrování existují dvě varianty. Pokud je hodnota `ioctl_args.key_d_len` exponentu `key_d` rovna nule, potom se k dešifrování použije RSA-CRT a soukromým klíčem je dvojice `key_p` a `key_q`. Jinak se použije standardní RSA se soukromým klíčem `key_d`. Výstupem funkce je pole `output_data` obsahující zašifrovaná data.

Funkce `crypto_drv_rsa_validate_key` má podobné rozhraní jako funkce pro šifrování algoritmem RSA. Na rozdíl od ní však pouze kontroluje validitu předaného klíče. U privátního exponentu `key_d` je zkontrolováno, jestli je nejnižší bit roven jedné:

```
mov     eax, [ebp+arg_8]
mov     ecx, [ebp+key_d]
mov     eax, [eax]
test    byte ptr [ecx+eax-1], 1
jnz     short loc_384A1
```

tedy jestli je klíč liché číslo. Stejný test se provádí také u hodnot `key_p`, `key_q` a `key_n`. U veřejného exponentu `key_e` se kontroluje, jestli je velký čtyři bajty a má nastavený nejnižší bit:

```
mov     eax, [ebp+key_e]
test    byte ptr [eax+3], 1
jnz     short loc_3819A
```

U hodnot `key_p` a `key_q` se dále testuje prvočíselnost a dělitelnost hodnotou `key_e`. S výslednými podíly se nicméně ani v jednom z případů nijak nemanipuluje a jejich význam není zřejmý.

U modulu `key_n` je testováno, zdali je beze zbytku dělitelný čísly `key_p` a `key_q`. Pokud je definováno `key_q`, je modul nejprvě vydělen `key_q` a

u výsledku se testuje prvočíselnost. V případě, že `key_q` není definován, ale `key_p` definován je, dělí se modul číslem `key_p`.

Funkce `crypto_drv_rsa_gen_key_start` iniciuje asynchronní generování RSA klíče. Generování klíče je provedeno funkcí `crypto_drv_rsa_gen_key`. Funkci je možné zavolat dvěma způsoby. První varianta generuje pouze: `key_n`, `key_p`, `key_q`, `key_e`. Druhá varianta navíc generuje i dešifrovací exponent `key_d`.

Zbylé dvě funkce, ioctl `1Dh` a `1Eh`, slouží pro zjištění stavu generování a zrušení požadavku na generování klíče.

3.6.5 Eliptické křivky

Tabulka 3.7: Funkce pro práci s eliptickými křivkami v modulu **crypto**

ioctl	funkce
20h	`crypto_drv_ecc_gen_key`
21h	`crypto_drv_ecc_val_key`
22h	`crypto_drv_ecc_sign`
23h	`crypto_drv_ecc_verify`
25h	`crypto_drv_ecc_ecdh`
26h	`crypto_drv_ecc_add_point`
27h	`crypto_drv_ecc_scalar_mult`
28h	`crypto_drv_ecc_is_point_on_curve`

Ovladač kryptografického modulu poskytuje několik funkcí pro práci s eliptickými křivkami. K dispozici jsou funkce pro generování a validaci klíčů, pro počítání nad eliptickými křivkami a také implementace algoritmů ECDSA a ECDH[2]. Všechny funkce a odpovídající ioctl čísla jsou uvedená v tabulce 3.7. Exportované funkce dostávají jako třetí argument strukturu:

```
struct ecc_ioctl_args {
  int32_t curve;
  //...
};
```

která, specifikuje křivku, na které bude výpočet probíhat, a délky vstupních parametrů (souřadnice bodů, celá čísla). Hodnota `ecc_ioctl_args.curve` může nabývat hodnot `0` až `7` (viz tabulka 3.8) V souladu s dostupnými informacemi knihovna poskytuje křivky definované ve standaru DSS[2, 24]. Navíc je k dispozici křivka P-160, která není součástí žádné verze standardu DSS. Některé identifikátory křivky se potom mapují na totožnou křivku

Tabulka 3.8: Dostupné eliptické křivky

id	0	1	2	3	4	5	6	7
křivka	P-192	P-224	P-256	P-384	P-521	P-160	P-256	P-256

avšak s jinými parametry. Každá křivka má dedikované funkce specifikované ve struktuře (tabulce):

```
struct ec_operators {
  int (*is_point_on_curve)(); // +10h
  int (*add_point)();         // +20h
  int (*scalar_mul)();        // +24h
};
```

v datové části binárního souboru.

Funkce `crypto_drv_ecc_is_point_on_curve` má pouze dva vstupní argumenty, kterými jsou souřadnice testovaného bodu. Funkce ověří, zdali se bod nachází na křivce dané hodnotou `ecc_ioctl_args.curve`. K výpočtu se používá funkce `ec_operators.is_point_on_curve` z odpovídající struktury.

Funkce `crypto_drv_ecc_scalar_mult` má tři parametry: celočíselnou hodnotu k a souřadnice bodu P. Výstupem funkce jsou souřadnice bodu $Q = kP$. Pro výpočet se použije funkce `ec_operators.scalar_mul` odpovídající eliptické křivky.

Funkce `crypto_drv_ecc_add_point` má čtyři parametry – souřadnice bodů P a Q. Výstupem funkce jsou dvě hodnoty – souřadnice bodu $S = P + Q$. K výpočtu se použije funkce `ec_operators.add_point` odpovídající křivky.

Funkce `crypto_drv_ecc_verify` slouží ke kontrole podpisu vytvořeného algoritmem ECDSA. Funkce `crypto_drv_ecc_sign` potom slouží k výpočtu podpisu algoritmem ECDSA.

Funkce `crypto_drv_ecc_val_key` je určena pro validaci klíče pro algoritmus ECDSA. Klíčem je v tomto případě číslo d a bod $Q = (Q_x, Q_y)$. U soukromého klíče d se kontroluje jeho délka.

Účelem funkce `crypto_drv_ecc_gen_key` je generování klíče pro ECDSA. Vstupem funkce může být hodnota d. Pokud není specifikována, je automaticky vygenerována. Výstupem funkce generující klíč jsou tři hodnoty: d, Q_x a Q_y.

3.6.6 Úložiště pro klíče

Hardwarový modul SKS (*secure key storage*) poskytuje aplikacím Intel ME bezpečné úložiště pro šifrovací klíče. Funkce pouze obstarávají předání argumentů do funkcí z ROM, které nebyly během analýzy k dispozici. Nebylo tak možné analyzovat, k čemu jsou jednotlivé funkce určeny. Z ladicích hlášek bylo možné odvodit pouze názvy dvou funkcí, viz tabulka 3.9.

Tabulka 3.9: Funkce pro práci s úložištěm klíčů v modulu **crypto**

2Bh	`crypto_drv_sks_copy_key`
2Ch	`crypto_drv_sks_install_wrapped_key`
2Dh	`crypto_drv_sks_sub_3E6D0`
2Eh	`crypto_drv_sks_sub_3E6E1`

3.7 Modul syslib

Analýza byla prováděna na modulu z firmware: `P10S-I-ASUS-4401` *a oddílu:* `OPR`.

Modul **syslib** je jediný modul bez vstupního bodu. Jedná se v podstatě o knihovnu funkcí pro práci s: haldou, procesy, kryptografií... Knihovna **syslib** je dostupná všem userspace modulům na adrese `9000h`. Binární soubor začíná tabulkou instrukcí `jmp`, které vedou na implementaci konkrétních funkcí ve druhé části souboru.

3.7.1 Kryptografické funkce

V předchozích kapitolách již byly zmíněny některé kryptografické funkce pracující s algoritmy SHA-256 nebo RSA. V této kapitole rozebrány funkce, které implementují nějaký šifrovací nebo hashovací algoritmus. Funkce, které pouze delegují práci na ovladač **crypto**, zde rozebírány nebudou.

Protože jsou některé hashovací a šifrovací funkce potřeba dříve, než je načten ovladač kryptografického koprocesoru, obsahuje **syslib** implementaci algoritmů SHA-256 a RSA. Dále poskytuje modul **syslib** funkce obalující všechny funkce poskytované ovladačem **crypto**. Pro komunikaci s ovladačem slouží soubor `/dev/crypto`. Po otevření souboru je filedescriptor uložen do paměti pro pozdější použití.

V analyzovaném firmware jsou níže popsané funkce použity v modulech **bup**, **loadmgr**, **mca_boot**, **pm**, **storage**, **sys_ctrl** a **vfs**. Pokud se podařilo určit smysl volání je to u dané funkce uvedeno.

3.7.1.1 hash_file

Funkce na adrese 90DC není součástí kryptografických funkcí. Neimplementuje žádný hashovací algoritmus, ani nepoužívá funkcí modulu **crypto**. Nicméně podle použití slouží k hashování otevřeného souboru. Funkce má signaturu:

```
hash_file (int32_t fd, char* hash, int32_t hash_len);
```

Vstupem je file descriptor otevřeného souboru a buffer pro hash a délka hashe. Ioctl volání na otevřených souborech je pravdepodobně obsluhováno modulem **vfs**.

3.7.1.2 invalidate_crypto_device

Funkce exportovaná na adrese 9320h má signaturu:

```
invalidate_crypto_device();
```

Pravděpodobně se jedná pouze o inicializační funkci, protože je volána z funkce main_thread_init každého modulu a je součástí sekvence:

```
call    near ptr 9000h
call    near ptr 9159h
push    dword ptr ds:34394h ; adresa se může lišit
call    near ptr 9235h
call    near ptr 91FEh
call    near ptr 9433h
call    near ptr 9320h
call    near ptr 919Fh
call    near ptr 9758h
```

Funkce invalidate_crypto_device nastaví hodnotu uloženého filedescriptoru na FFFFFFFFh. Funkce se nepokouší použít předešlou uloženou hodnotu, například k zavření souboru.

3.7.1.3 open_crypto_device

Funkce exportovaná na adrese 9325h má signaturu:

```
open_crypto_device();
```

Jedná se pouze o wrapper funkce open_crypto_device_f, který nemá žádný efekt:

```
push    ebp
mov     ebp, esp
pop     ebp
jmp     open_crypto_device_f
```

Funkce `open_crypto_device_f` je exportovaná na adrese `932Fh` a jejím účelem je otevřít soubor `/dev/crypto` a uložit filedescriptor do paměti:

```
mov     eax, large ds:1020h
add     eax, 104h
mov     [eax], <fd>
```

3.7.1.4 `close_crypto_device`

Funkce na adrese `932Ah` se nazývá `close_crypto_device` a jejím jediným úkolem je zavolání funkce `fclose` na filedescriptor souboru `/dev/crypto` uložený v paměti.

3.7.1.5 `set_crypto_device`

Funkce `set_crypto_device(int32_t* fd)` na adrese `9334h` pouze nastaví uloženou hodnotu filedescriptoru na hodnotu parametru `fd`.

3.7.1.6 `hash_verify`

Funkce exportovaná na adrese `9339h` má signaturu:

```
int32_t hash_verify(
  int32_t flags,
  int32_t arr_len,
  char*   arr,
  char*   hash_buffer
);
```

Hodnota `flag` specifikuje použitý hashovací algoritmus:

- 0 – softwarová implementace algoritmu SHA-1,
- 1 – softwarová implementace algoritmu SHA-256,
- 2 – softwarová implementace algoritmu MD5,
- jinak je zavolána funkce `crypto_drv_hash_simple` z ovladače **crypto** a použije se hardwarová implementace požadovaného algoritmu.

Kód obsahuje také horní hranice pro velikost vstupu, od které se vždy použije hardwarová implementace. Pro MD5 je hranice stanovená na `5967 B`, pro SHA-1 je to `1368 B` a pro SHA-256 je to `655 B`.

V modulu **bup** je funkce `hash_verify` volána ve funkci `manifest_check`

3.7.1.7 `hash_data_update`

Funkce exportovaná na adrese `933Eh`:

```
hash_data_update(
  int32_t          flags,
  int32_t          flag,
  int32_t          arr_len,
  char*            arr,
  struct hash_ctx* ctx
);
```

slouží k hashování nesouvislého bloku dat. Struktura `ctx`:

```
struct hash_ctx {
  //...
  char  hash[32]; // + 50h
  dword hash_len; // + 70h
};
```

uchovává kontext mezi jednotlivými voláními funkce `hash_data_update`.

Parametr `flag` může nabývat čtyř hodnot:

- `0` – první volání – provede se inicializace struktury `ctx`,
- `1` – další volání – pouze se hashuje obsah `arr`,
- `2` – poslední volání
- `3` – výpočet se deleguje na funkci `hash_verify`.

Parametr `flags` říká, jaký algoritmus se použije pro hashování:

- `2` – MD5
- `0` – SHA-1
- `1` – SHA-256
- v ostatních případech se výpočet deleguje na modul **crypto**, který zajistí HW akceleraci výpočtu.

V modulu **pm** je funkce `hash_data_update` volána při načítání modulu ve funkci `load_module`, přičemž se používá algoritmus SHA-256. V modulech **loadmgr** a **bup** je funkce použita při kontrole manifestu.

3.7.1.8 `hash_data_final`

Funkce na adrese `9352h`:

```
dword hash_data_final(
  struct hash_ctx* ctx,
  char*            buffer
);
```

pouze použije `memcpy` na vykopírování hashe dlouhého `ctx->hash_len` ze struktury `hash_ctx` do pole `buffer`.

V případě, že `ctx` nebo `buffer` je `0`, vrací funkce konstantu `16h`. Odpovídající velikost pole `buffer` si musí zajistit volající a není kontrolována. V případě úspěchu funkce vrací `0`.

Funkce `hash_data_final` je komplementární k funkci `hash_data_update` a proto je využívána stejnými moduly.

3.7.1.9 `hash_final_and_compare`

Ve funkci exportované na adrese `9389h`:

```
dword hash_final_and_compare(
  struct sha_256* ctx,
  char*           hash
);
```

je pouze zavolána funkce `hash_data_final` na `ctx`. Výsledný hash je porovnán funkcí `compare_arrays_secure` s hodnotou `hash`. Použitá funkce vždy porovná celé pole tak, aby nedošlo k vytvoření postranního časového kanálu.

3.7.1.10 `hmac_simple`

Funkce dostupná na adrese `9366h` má signaturu:

```
dword hmac_simple(
  int32_t flags,
  int32_t key_len,
  int32_t a,
```

```
  int32_t data,
  int32_t data_len,
  int32_t b
);
```

Parametr `flags` určuje použitou hashovací funkcí. Pro 0, 1 a 2 se použije funkce `hash_data_update`. Navíc bylo možné v této funkci odlišit také možnosti: 3, 4 a 5. Podle délky bloku hashovací funkce, které se v závislosti na hodnotě `flags` nastavují, je možné usuzovat následující:

- 3 – SHA-224
- 4 – SHA-384
- 5 – SHA-512

Funkce `hmac_simple` volání těchto hashovacích funkcí deleguje na modul **crypto**. Při pozdější analýze modulu **crypto** byla ověřena hypotéza o použití konkrétních hashovacích algoritmů.

V modul **bup** je funkce `hmac_simple` použita k odvození klíče, který slouží k zajištění integrity úložiště. Modul **loadmgr** pravdepodobně používá funkci `hmac_simple` k manipulaci s oddílem `IVBP`. Oddíl `IVBP` je „iv + bring-up cache“ [6].

Modul **vfs** obsahuje čtyři volání funkce `9366h`. Z dosud zjištěných informací lze říci pouze to, že se používají při práci s SPI flash pamětí.

3.7.1.11 `rsa_encrypt`

Funkce exportovaná na adrese `93D9h`:

```
verify_rsa_encrypt(
  int32_t   keysize, // velikost klíče v~bytech
  char*     data,    // otevřený text
  char*     out,     // buffer pro šifrový text
  char*     mod,     // modul
  char*     exp      // exponent
);
```

pouze deleguje šifrování dat na funkci `crypto_drv_rsa_raw_enc_dec` z modulu **crypto**. Funkce je volána v modulu **loadmgr**, kde je použita při kontrole integrity manifestu.

Kapitola 4

Vyhodnocení

Ve čtvrté kapitole jsou shrnuty poznatky získané z dokumentace a reverzního inženýrství modulů Intel ME. V kombinaci s dalšími dostupnými informacemi jsou zde zhodnocena bezpečnostní rizika spojená s Intel ME.

4.1 Kvalita kódu

Při reverzní analýze kód modulů vzbuzoval dojem, že si vývyvojáři dali záležet na bezpečnosti celého systému. Jsou testovány návratové hodnoty, délky staticky/dynamicky alokovaných polí. Zásobník funkce je opatřen ochranou v podobě kanárka.

Nicméně i přes veškeré úsilí už bylo v minulosti identifikováno několik zranitelností. V roce 2017 bylo nalezeno osm zranitelností (tabulka 4.1). Pro zneužtí většiny uvedených zranitelností je třeba fyzický přístup k počítači. Jedinou výjimkou je zranitelnost CVE-2017-5712 v modulu AMT, která je zneužitelná vzdáleně, po síti.[25]

Tabulka 4.1: CVE z roku 2017 spojené s Intel ME

CVE-2017-5705	CVE-2017-5706	CVE-2017-5707
CVE-2017-5708	CVE-2017-5709	CVE-2017-5710
CVE-2017-5711	CVE-2017-5712	

V roce 2018 bylo indentifikováno dvanáct zranitelností (tabulka 4.2). Většina jich byla nalezena při interním auditu. Pouze dvě CVE byla identifikována nezávislými výzkumníky.

Dvě zranitelnosti byly nalezeny v ovladači rozhraní HECI a tři byly v modulu AMT. U zbytku nebylo upřesněno v které části systému se nachází.

Tabulka 4.2: CVE z roku 2018 spojené s Intel ME

CVE-2018-12188	CVE-2018-12189	CVE-2018-12190
CVE-2018-12191	CVE-2018-12192	CVE-2018-12199
CVE-2018-12198	CVE-2018-12208	CVE-2018-12200
CVE-2018-12187	CVE-2018-12196	CVE-2018-12185

Ke zneužití uvedených zranitelností musí mít útočník fyzický nebo lokální přístup k zranitelnému zařízení. Pouze zranitelnost CVE-2018-12187 v AMT je možné zneužít vzdáleně.[26]

Je evidentní, že v každém programu je možné najít bezpečnostní zranitelnost. Velmi účinnou ochranou by bylo snížení počtu řádků kódu odebráním modulů, jejichž přítomnost není nezbytně nutná pro funkci počítače. Z pohledu útočníka jsou zajímavé především moduly **amt** nebo **heci**, které zajišťují komunikaci s prostředím vně Management Engine. Uživatel počítačového systému bohužel oficiálně nemá možnost ovlivnit složení firmware a přítomnost jednotlivých modulů.

4.2 Integrita systému

V rámci práce bylo ověřeno, že kontrola integrity při startu systému je prováděna tak jak je popsáno v dostupné literatuře[2]. K ověření původu manifestu (podpisu) je použit algoritmus RSA s klíčem o velikosti 2048 bitů, který je dle standardu FIPS považován za bezpečný[24]. K ověření integrity jednotlivých modulů je použita hashovací funkce SHA-256, která je taktéž v současné době považována za bezpečnou [27].

V rámci práce byl analyzován firmware Inte ME. Aby mohla fýt funkcionalita Intel ME snadněji rozšiřována, byl vytvořen modul DAL (*dynamic application loader*). Jedná se o modul určený k načítání a spouštění appletů napsaných v programovacím jazyce Java. Každé rozšíření by mělo být opatřeno podpisem společnosti Intel.[2] V této práci nebyl modul DAL rozebírán, přestože by teoreticky mohl být zneužit ke spuštění cizího kodu v Intel ME.

4.3 Dostupné kryptografické funkce

Kryptograficky modul Intel ME poskytuje kvůli zpětné kompatibilitě některé kryptografické algoritmy, které již nejsou považovány za bezpečné. Mezi ně patří šifry 3DES[28] a RC4[29] a hashovaci funkce MD5 a SHA-1. V ana-

lyzovaném kódu nicméně nebyly zmíněné funkce použity. Ostatní dostupné algoritmy: AES, RSA, ECDH, ECDSA, SHA-2 jsou považovány za bezpečné a byly implementovány dle standardu.

Z pohledu bezpečnosti uživatele počítačového systému proto existuje jen malé riziko, že by byla bezpečnost systému narušena použitím slabého nebo nestandardního kryptografického algoritmu.

4.4 Přínos pro uživatele

Intel ME je základním kamenem několika technologií, jejichž cílem je zvýšit bezpečnost počítačového systému a jeho uživatele. Jednou z technologií, která je postavena na Intel ME je Intel Boot Guard.

Technologie Intel Boot Guard poskytuje ochranu před útokem typu Evil Maid. Útoku je veden na vypnutý počítač nabootováním z přenosného média a injekcí maligního kódu do zavaděče operačního systému. Pokud by útočník modifikoval zavaděč systému s aktivovaným Intel Boot Guard, bylo by při startu systému detekováno narušení integrity zavaděče a bootování operačního systému by bylo zastaveno.[2]

Nevýhodou popsaného řešení je, že útočník sice nezíská přístup k systému, ale může uživateli odepřít přístup k systému a uloženým datům. Z pohledu běžného uživatele poskytují lepší ochranu pevné disky s vestavěnou podporou šifrování (*self-encrypting drives*), protože poskytují ochranu před modifikací zavaděče vypnutého stroje.

Další technologií založenou na Intel ME je Platform Trust Technology (PTT). Jedná se o implementaci *firmware-based* TPM. Z pohledu Intel ME je PTT aplikací, která využívá, mimo jiné, funkcí modulu **crypto**. Z pohledu bezpečnosti je hardwarový TPM modul vždy lepší volbou. PTT ale najde uplatnění v aplikacích, kdy je důležitá úspora místa, například ve vestavných zařízeních.[2]

Poslední zde zmíněnou technologií založenou na Intel ME je Advanced Management Technology. Z pohledu bezpečnosti počítačového systému je AMT aplikací, která může bezpečnost systému pouze snížit. Vzdálená správa je nepochybně užitečná v případě serverů nebo pracovních stanic. Vzhledem k tomu, že AMT je jako součást Intel ME zranitelná i vzdálenými útoky, je potřeba zvážit jestli by modul **amt** měl být součástí součástí firmware laptopů a osobních počítačů.

Systém jako je Intel ME určitě najde své opodstatnění ať už se jedná o bezpečnost systému nebo usnadnění vzdálené správy. Jejich funkcionalita by ale měla být vždy maximálně omezena tak, aby bylo minimalizováno ri-

ziko napadení. V případě zneužití zranitelnosti v Intel ME se totiž podobný systém stává dokonalým útočištěm pro škodlivý kód.

Závěr

Úvodní část práce je věnována seznámení s Intel Management Engine. Jejím cílem je uvést čtenáře do problematiky a měla by pomoci s orientací v dalším textu. V první kapitole je popsán počítač postavený na technologiích společnosti Intel a hardwarová struktura Intel ME. Dále je v první kapitole popsáno, kde a jakým způsobem je uložen firmware Intel ME.

Druhá část popisuje architekturu softwarové části Intel ME, komunikaci mezi moduly a zavádění systému. Ve třetí kapitole je popsána funkcionalita modulů, které se podílí na zavádění firmware. Reverzní analýzou bylo ověřeno, že je při bootování kontrolována integrita jednotlivých komponent tak, jak je popsáno v dostupné literatuře.

Ve třetí části jsou popsány kryptografické funkce dostupné aplikacím běžícím v rámci Intel ME. Byla provedena reverzní analýza ovladače hardwarového kryptografického modulu a části systémové knihovny implementující kryptografické funkce. U ostaních modulů Intel ME bylo prozkoumáno, zdali a za jakým účelem používají některou z nabízených kryptografických funkcí.

V poslední kapitole jsou poznatky získané v praktické části práce shrnuty a posouzeny s ohledem na bezpečnost uživatele počítačového systému. Dále jsou ve čtvrté kapitole popsány přínosy a rizika technologií, které mají vliv na bezpečnost počítačového systému a jsou postaveny na Intel ME. Všechny cíle práce byly splněny.

Dále by se tato práce mohla stát odrazovým můstkem pro zájemce o další zkoumání Intel ME. Management Engine je komplexní systém s mnoha funkcemi a jeho bezpečnost je klíčová pro bezpečnost celého počítačového systému.

Literatura

[1] Z390 Technical Specifications. [cit. 2019-01-10]. Dostupné z: https://www.intel.com/content/www/us/en/products/chipsets/desktop-chipsets/z390.html

[2] Ruan, X.: *Platform embedded security technology revealed: safeguarding the future of computing with Intel embedded security and management engine*. Berkeley, California: Apress Open, 2014, ISBN 978-1-4302-6571-9.

[3] Alexander Tereshkin, R. W.: BHUSA: Introducing Ring -3 Rootkits. červenec 2009.

[4] Sarangdhar, N.: FIDO* Support on Intel® Platforms. [cit. 2019-03-29]. Dostupné z: https://www.intel.com/content/www/us/en/security/hardware/fido-support-white-paper.html

[5] Intel: *Intel C620 Series Chipset Platform Controller Hub*. 2019, [cit. 2019-04-07]. Dostupné z: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/c620-series-chipset-datasheet.pdf

[6] Maxim Goryachy, D. S., Mark Ermolov: Intel ME: The Way of the Static Analysis. březen 2017. Dostupné z: https://www.youtube.com/watch?v=2_aokrfcoUk

[7] Intel: *Intel Atom Z8000 Processor Series*. 2016, [cit. 2019-04-07]. Dostupné z: https://www.intel.co.uk/content/dam/www/public/us/en/documents/datasheets/atom-z8000-datasheet-vol-1.pdf

[8] Intel: *Intel Atom Processor C3000 Product Family*. 2018, [cit. 2019-04-07]. Dostupné z: https://www.intel.co.uk/content/dam/www/public/us/en/documents/technical-specifications/c3000-family-datasheet.pdf

[9] Intel: *300 Series Chipset Family On-Package Platform Controller Hub*. 2019, [cit. 2019-01-10]. Dostupné z: https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/300-series-chipset-on-package-pch-datasheet-vol-1.pdf

[10] *P10S-I*. 2017. Dostupné z: https://dlcdnets.asus.com/pub/ASUS/mb/socket1151/P10S-I/Manual/E13687_P10S-I_UM_V5_WEB.pdf

[11] Sklyarov, D.: Intel ME 11.x Firmware Images Unpacker. [cit. 2019-02-11]. Dostupné z: https://github.com/ptresearch/unME11

[12] Sklyarov, D.: Intel ME 12.x Firmware Images Unpacker. [cit. 2019-02-11]. Dostupné z: https://github.com/ptresearch/unME12

[13] Sklyarov, D.: Recovering Huffman tables in Intel ME 11.x. prosinec 2017. Dostupné z: http://blog.ptsecurity.com/2017/12/huffman-tables-intel-me.html

[14] G481-H80. [cit. 2019-03-21]. Dostupné z: https://www.gigabyte.com/cz/High-Performance-Computing-System/G481-H80-rev-100

[15] AST2500. [cit. 2019-03-21]. Dostupné z: https://www.aspeedtech.com/products.php?fPath=20&rId=440

[16] Tanenbaum, A. S.: *Modern operating systems*. Upper Saddle River, 1780 N.J.: Pearson/Prentice Hall, třetí vydání, 2008, ISBN 978-0136006633.

[17] Corna, N.: me_cleaner: How does it work? [cit. 2019-04-28]. Dostupné z: https://github.com/corna/me_cleaner/wiki/How-does-it-work%3F

[18] Mark Ermolov, M. G.: 34C3 - Inside Intel Management Engine. 12 2017, [cit. 2019-02-14]. Dostupné z: https://www.youtube.com/watch?v=JMEJCLX2dtw

[19] Podpora pro produkt Latitude 7490. [cit. 2019-04-28]. Dostupné z: https://www.dell.com/support/home/cz/cs/czbsd1/product-support/product/latitude-14-7490-laptop/drivers

[20] Produktová podpora pro P10S-I. [cit. 2019-04-28]. Dostupné z: https://www.asus.com/cz/Commercial-Servers-Workstations/P10S-I/HelpDesk_BIOS/

[21] Intel ME: Flash File System Explained. 2018, [cit 2019-03-12. Dostupné z: https://speakerdeck.com/defcon/dmitry-sklyarov-intel-me-flash-file-system-explained

[22] Joan DAEMEN, V. R.: *The Design of Rijndael: AES - The Advanced Encryption Standard*. New York: Springer, 2002, ISBN 3-540-42580-2.

[23] AES OFB examples. [cit. 2019-04-20]. Dostupné z: https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/AES_OFB.pdf

[24] Federal Inf. Process. Stds. (NIST FIPS) - 186-4 Digital Signature Standard. doi:10.6028/NIST.FIPS.186-4. Dostupné z: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

[25] Intel® Management Engine Critical Firmware Update (Intel-SA-00086). [cit. 2019-04-29]. Dostupné z: https://www.intel.com/content/www/us/en/support/articles/000025619.html

[26] Intel® CSME, SPS, TXE and Intel® AMT 2018.4 QSR Advisory. [cit. 2019-04-29. Dostupné z: https://www.intel.com/content/www/us/en/security-center/advisory/INTEL-SA-00185.html

[27] Secure Hash Standard. doi:10.6028/NIST.FIPS.180-4. Dostupné z: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[28] Update to Current Use and Deprecation of TDEA. [cit. 2019-04-28]. Dostupné z: https://csrc.nist.gov/News/2017/Update-to-Current-Use-and-Deprecation-of-TDEA

[29] Popov, A.: Prohibiting RC4 Cipher Suites. únor 2015, doi:10.17487/RFC7465. Dostupné z: https://tools.ietf.org/html/rfc7465

Příloha A

Seznam použitých zkratek

AMT Active Management Technology

CPD Code Partition Directory

CSME Converged Security Management Engine

DMA Direct Memory Access

DRAM Dynamic Random Access Memory

FD Flash Descriptor

FPT Flash Partition Table

HECI Host Endpoint Controller Interface

IE Inovation Engine

ME Management Engine

MEI Management Engine Interface

PCH Platform Controller Hub

PID Process Identifier

ROM Read Only Memory

SKS Secure Key Strorage

SPI Serial Peripheral Interface

SPS Server Platform Services

SRAM Static Random Access Memory

TPM Trusted Platform Module

TXE Trusted Execution Engine

Příloha B

Obsah přiloženého CD

- `firmware` ..... analyzovaný firmware
- `tex` ..... zdrojová forma práce ve formátu LaTeX
- `README.md` ..... stručný popis obsahu CD
- `thesis.pdf` ..... text práce ve formátu PDF