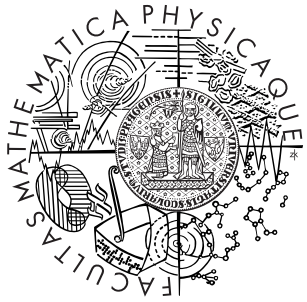
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Jiří Tlach

Moderní operační systém bez MMU

Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Martin Děcký
Studijní program: Informatika, obor Softwarové systémy

2010

Rád bych poděkoval svému vedoucímu Mgr. Martinu Děckému za rady a připomínky a rovněž všem lidem v HelenOS týmu za vytvoření tak unikátního a zajímavého operačního systému jakým HelenOS je.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 6. prosince 2010 Jiří Tlach

Obsah

1 Úvod **7**
- 1.1 Motivace . . . 7
- 1.2 Cíle . . . 7
- 1.3 Struktura textu . . . 8
- 1.4 Typografické konvence . . . 8

2 MMU jednotka a virtuální paměť **9**
- 2.1 Techniky virtuální paměti . . . 9
  - 2.1.1 Stránkování . . . 9
  - 2.1.2 Segmentace . . . 11
- 2.2 Využití v operačních systémech . . . 11
  - 2.2.1 Mapování a výměna stránek . . . 12
  - 2.2.2 Ochrana a izolace . . . 12
  - 2.2.3 Externí fragmentace . . . 12
  - 2.2.4 Paměťově mapované soubory a sdílená paměť . . . 13
  - 2.2.5 Sdílený kód . . . 13
  - 2.2.6 Detekce přetečení zásobníku . . . 13
  - 2.2.7 Detekce nulového ukazatele . . . 14

3 Paměťová ochrana bez MMU **15**
- 3.1 Hardwarové prostředky . . . 15
  - 3.1.1 MPU . . . 15
  - 3.1.2 Mondrianská ochrana paměti . . . 17
- 3.2 Softwarové prostředky . . . 20
  - 3.2.1 Bezpečné programovací jazyky . . . 20
  - 3.2.2 Softwarová MMU . . . 21
  - 3.2.3 SFI . . . 22
  - 3.2.4 XFI . . . 23
  - 3.2.5 Izolace modulů v uzlech senzorických sítí . . . 25
- 3.3 Kombinované prostředky . . . 26
  - 3.3.1 Harbor a XFI . . . 26
  - 3.3.2 Izolace na úrovni funkcí . . . 27
- 3.4 Srovnání . . . 27

4 Operační systém HelenOS **30**
- 4.1 Správa paměti . . . 30
  - 4.1.1 Fyzická paměť . . . 30
  - 4.1.2 Virtuální paměť . . . 30
- 4.2 Procesy a vlákna . . . 31
  - 4.2.1 Meziprocesová komunikace . . . 31

4.2.2 Spouštění procesů . . . 32

5 Návrh a implementace rozšíření HelenOS 33
5.1 Diskuze . . . 33
5.1.1 Kritéria návrhu a implementace . . . 33
5.1.2 Volba techniky paměťové ochrany . . . 34
5.1.3 Návrh rozšíření . . . 34
5.1.4 Procesorová architektura implementace . . . 35
5.2 Režim bez MMU . . . 35
5.2.1 Umístění jádra ve fyzické paměti . . . 36
5.2.2 Správa adresových prostorů . . . 36
5.2.3 Správa paměťových oblastí . . . 37
5.2.4 Rozhraní pro ovladače zařízení . . . 38
5.2.5 Sdílení paměti s IPC . . . 38
5.2.6 Procesy . . . 40
5.2.7 Správa paměti v uživatelském prostoru . . . 40
5.2.8 IA-32 závislý kód . . . 41
5.3 Softwarová izolace pomocí XFI . . . 41
5.3.1 ia32binrewriter . . . 41
5.3.2 Strážci a běhové kontroly . . . 42
5.3.3 Strážce zásobníku . . . 44
5.3.4 Strážce paměťových přístupů . . . 49
5.3.5 Strážce přímých skoků . . . 52
5.3.6 Strážce nepřímých skoků . . . 52
5.3.7 Stínový zásobník . . . 53
5.3.8 Pořadí strážců . . . 54
5.4 Systémová podpora pro XFI . . . 55
5.4.1 Datové struktury jádra . . . 56
5.4.2 Práce s vlákny a vlákénky . . . 56
5.4.3 Správa paměti . . . 58
5.4.4 Výjimky . . . 59
5.4.5 Spouštění procesů . . . 59
5.5 Nedostatky v implementaci XFI . . . 59
5.5.1 Přepínání kontextu vlákének . . . 59
5.5.2 Provedení nepovoleného paměťového přístupu . . . 59
5.6 Srovnání s původní implementací XFI . . . 61

6 Vyhodnocení 63
6.1 Současný stav . . . 63
6.2 Zachování funkcí MMU . . . 63
6.3 Minimální změny zdrojových kódů . . . 64
6.4 Transparentnost . . . 65
6.5 Přenositelnost . . . 65
6.6 Efektivita . . . 66
6.6.1 Srovnávací testy . . . 66
6.6.2 Režim bez MMU . . . 67
6.6.3 Softwarová izolace s XFI . . . 68
6.7 Závěr . . . 70

7 Možnosti rozšíření **72**
7.1 Režim bez MMU . . . 72
7.1.1 Alokátor fyzické paměti . . . 72
7.1.2 Provázané zásobníky . . . 72
7.1.3 Optimalizace alokátoru paměti v uživatelském prostoru . . . 73
7.1.4 Paměťová optimalizace spouštění procesů . . . 73
7.1.5 Dynamicky linkované sdílené knihovny . . . 73
7.2 Softwarová izolace s XFI . . . 73
7.2.1 Nástroj ia32binrewriter . . . 73
7.2.2 Ověření spouštěného XFI procesu . . . 74
7.2.3 Optimalizace běhových kontrol . . . 74
7.2.4 Podpora sdílených knihoven . . . 74
7.2.5 Podpora SMP . . . 74

8 Podobné projekty **76**
8.1 uClinux . . . 76
8.2 Singularity . . . 77
8.3 Mondrix . . . 77
8.4 VINO . . . 77

9 Závěr **79**
9.1 Splnění cílů . . . 79
9.2 Přínosy práce . . . 79
9.3 Navazující práce . . . 80

Literatura **81**

A Obsah přiloženého CD **84**

B Sestavení a spuštění HelenOS **85**
B.1 Sestavení ze zdrojových souborů . . . 85
B.2 Spuštění . . . 86

C Nástroj ia32binrewriter **87**
C.1 Sestavení ze zdrojových souborů . . . 87
C.2 Příkazová řádka . . . 87
C.3 Příklad použití . . . 87

D STATIF framework **89**

Název práce: Moderní operační systém bez MMU
Autor: Jiří Tlach
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: Mgr. Martin Děcký
e-mail vedoucího: Martin.Decky@mff.cuni.cz

Abstrakt: Jednotka správy paměti (MMU) je hardwarová komponenta zajišťující zejména překlad virtuálních adres na fyzické a s tím související bezpečné oddělení jádra a procesů. Řada procesorů i v dnešní době však postrádá MMU jednotku. HelenOS je výzkumný operační systém vyvíjený na MFF UK. Jádro tohoto systému využívá hardwarové podpory MMU jednotky procesoru pro překlad virtuálních adres na fyzické pomocí stránkování. Předmětem této práce je přehled technik (částečné) náhrady MMU na procesorech postrádajících tuto jednotku. Dále práce obsahuje návrh, rozbor a prototypovou implementaci rozšíření systému HelenOS, které umožní jeho běh na procesorech bez této jednotky.

Klíčová slova: MMU jednotka, ochrana paměti, softwarová izolace

Title: Modern operating system without MMU
Author: Jiří Tlach
Department: Department of Software Engineering
Supervisor: Mgr. Martin Děcký
Supervisor's e-mail address: Martin.Decky@mff.cuni.cz

Abstract: Memory management unit (MMU) is a hardware component providing above all the translation of virtual addresses to physical addresses and thus providing secure isolation of kernel and processes. HelenOS is a research operating system which is being developed at MFF UK. The kernel of HelenOS uses hardware MMU of the processor for virtual to physical memory translation using paging. The goal of this work is to provide an overview of the techniques which can be used to (partially) substitute the functionality of MMU by other means. A proposed design, analysis and prototype implementation of an extension to HelenOS is also part of this work. This extension enables functionality of HelenOS on processors without MMU.

Keywords: MMU-less, memory protection, software fault isolation

Kapitola 1

Úvod

1.1 Motivace

Pestrost dnešních počítačových systémů již takřka nezná mezí – najdeme mezi nimi nejjednodušší jednoúčelová zařízení pro měření teploty či vlhkosti prostředí přes běžné stolní PC, až po nejvýkonnější superpočítače disponující stovkami procesorových jader. Podobnou různorodost najdeme i mezi operačními systémy. I přes jejich rozdíly však mají již od vzniku počítačů stále stejnou funkci, a to provozování nejrůznějších úloh (procesů) dle přání uživatelů. Je běžnou praxí, že takových úloh běží současně až několik desítek, aniž by se navzájem přímo ovlivňovaly. Každé z nich poskytuje operační systém iluzi, že právě ona je jedinou běžící úlohou v celém systému a má veškeré prostředky systému zcela pro svou činnost. Řada operačních systémů používá k dosažení takové iluze mimo jiné virtuální paměť poskytovanou hardwarovou jednotkou pro správu paměti – tzv. MMU jednotku (*Memory Management Unit*).

Jedním z operačních systémů, které zcela spoléhají na přítomnost MMU jednotky, je také HelenOS. Jedná se o experimentální operační systém vyvíjený na Matematicko-fyzikální fakultě Univerzity Karlovy v Praze, který je populární zejména v akademické sféře mezi studenty, o čemž svědčí i četné diplomové a semestrální práce týkající se jeho rozšiřování. Tento systém podporuje řadu různých procesorových architektur a již při svém vzniku počítal s využitím MMU jednotky pro vzájemnou izolaci procesů a virtualizaci dostupné paměti. To však v současné době znemožňuje přechod na jiné procesorové architektury, které z určitých důvodů MMU jednotku postrádají. Pokusit se přizpůsobit operační systém HelenOS pro takové typy procesorů je proto určitě zajímavý námět a také důvod vzniku této diplomové práce.

1.2 Cíle

Výstupem této práce by měl být obecný návrh rozšíření operačního systému HelenOS umožňující portaci na libovolný procesor bez MMU jednotky. Tento návrh by poté měl být implementován v podobě prototypu na některé procesorové architektuře podporované systémem HelenOS v současné době, jelikož portace na zcela novou architekturu by byla sama o sobě námětem na samostatnou diplomovou práci. Cílem návrhu a implementace bylo mělo být i splnění následujících bodů:

- částečné zachování funkcí nabízených MMU jednotku,
- co nejmenší změny a zásahy do stávajícího jádra systému,
- transparentnost,
- efektivita.

Výsledné řešení by tedy mělo alespoň zčásti nahradit chybějící podporu virtuální paměti a s ní související paměťovou ochranu a izolaci procesů jinými prostředky při minimálních zásazích do stávajícího jádra a vůbec fungování celého systému. Cílem je rovněž transparentnost takového řešení, díky čemuž nebudou muset systéméví a aplikační programátoři rozlišovat při dalším vývoji, zda jejich rozšíření jádra či aplikace běží na procesoru s MMU jednotkou nebo bez ní. Motivací tohoto cíle je rovněž snaha, aby výsledky této diplomové práce byly přijaty komunitou vývojářů HelenOS a zapracovány do dalších oficiálních verzí tohoto systému.

1.3 Struktura textu

Text je členěn do 9 kapitol a provádí čtenáře od analýzy problematiky přes návrh a implementaci až po vyhodnocení výsledků a celkového zhodnocení práce. Obsah jednotlivých kapitol je následující:

Kapitola 2 stručně popisuje MMU jednotku a její význam a použití v dnešních operačních systémech.

Kapitola 3 rozebírá problematiku částečné náhrady chybějících funkcí MMU jednotky a podává přehled několika technik, které jsou dnes používány.

Kapitola 4 stručně seznámí čtenáře s operačním systémem HelenOS s důrazem na informace relevantní pro tuto diplomovou práci.

Kapitola 5 pak obsahuje návrh a implementaci rozšíření HelenOS pro běh na procesorech bez MMU jednotky.

Kapitola 6 provádí vyhodnocení výsledného řešení prezentovaného v přechozí kapitole.

Kapitola 7 shrnuje různé možnosti, jakými by se dalo prezentované řešení vylepšit tak, aby se více přiblížilo praktické použitelnosti.

Kapitola 8 uvádí příklady některých operačních systémů, které rovněž běží na procesorech bez MMU jednotky.

Kapitola 9 uzavírá celou práci a shrnuje její výsledky a přínosy.

1.4 Typografické konvence

V celém textu práce je dodržováno několik jednoduchých stylistických konvencí:

- Jména zdrojových souborů jsou tištěna proporcionálním písmem, např. `main.c`.
- Odkazy na zdrojové soubory začínající adresáři `uspace` nebo `kernel` směřují do adresářového stromu zdrojových souborů HelenOS, které lze najít na přiloženém CD v archivu `Sources/helenos-0.4.2-nommu-xfi.tgz`.
- Odkazy na zdrojové soubory začínající adresářem `ia32binrewriter` směřují do adresářového stromu zdrojových souborů nástroje *ia32binrewriter*, které lze najít na přiloženém CD v archivu `Sources/ia32binrewriter.tgz`.
- Názvy pojmů, funkcí a jejich parametrů jsou tištěny kurzívou, např. *thread_create()*.
- Pro snažší orientaci v anglické technické literatuře je u řady pojmů uveden také jejich originální anglický název v závorce, např. *přetečení zásobníku (stack overflow)*.
- Části zdrojových kódů jsou tištěny proporcionálním písmem, např.

```
if (condition) print(text);
return;
```

Kapitola 2

MMU jednotka a virtuální paměť

Jednotka správy paměti nebo-li MMU jednotka (*Memory Management Unit*) je dnes standardní součástí moderních procesorů pro desktopové či serverové počítačové systémy. Potřeba jejího vzniku sahá do konce 50. let 20. století, kdy se přecházelo z dávkového zpracování na koncept paralelního běhu úloh – tzv. *multitasking*. Běžící úlohy sdílely jediný adresový prostor, což mohlo v důsledku chyby jedné z nich narušit i běh ostatních, jelikož vzájemně nebyly nijak izolovány. Navíc kapacita fyzické paměti byla velmi omezená a v případě nedostatku paměti bylo nutné zavádět do paměti pouze ty části (moduly) úloh, které byly momentálně potřeba. Tato technika, známá jako *překryvné moduly (overlays)*, však vyžadovala, aby každá aplikace byla již při svém překladu rozdělena na moduly a aby sama při svém běhu dokázala provádět přesouvání modulů z paměti např. na disk a zpět.

Řešení těchto dvou problémů nakonec přinesl koncept *virtuální paměti*, jehož hlavní myšlenkou je samostatný virtuální adresový prostor pro každou spuštěnou úlohu. Vzhledem k tomu, že fyzická paměť je pouze jednorozměrná, bylo nutné zavést mapování z virtuálních adresových prostorů na tuto fyzickou paměť, což se stalo primární úlohou MMU jednotky. Její typické umístění je mezi CPU jednotkou a fyzickou pamětí, resp. adresovou sběrnicí, což jí umožňuje realizovat mapování nebo-li překlad každé virtuální adresy na fyzickou (viz obrázek 2.1).

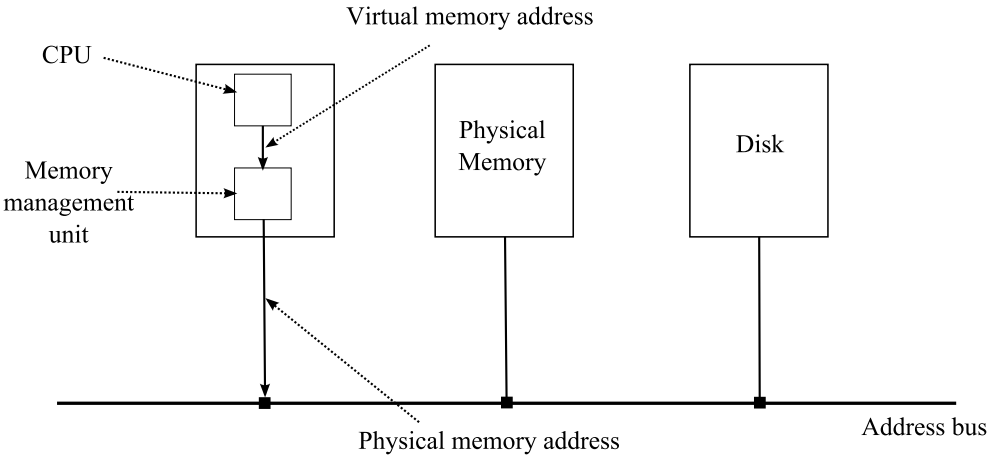
Jakým způsobem virtuální paměť implementovat a jaké výhody to nese pro operační systém postavený nad virtuální paměť bude dále rozebráno v této kapitole. Hlavním zdrojem informací zde uvedených byla publikace Moderní operační systémy od Andrew S. Tanenbauma [35].

2.1 Techniky virtuální paměti

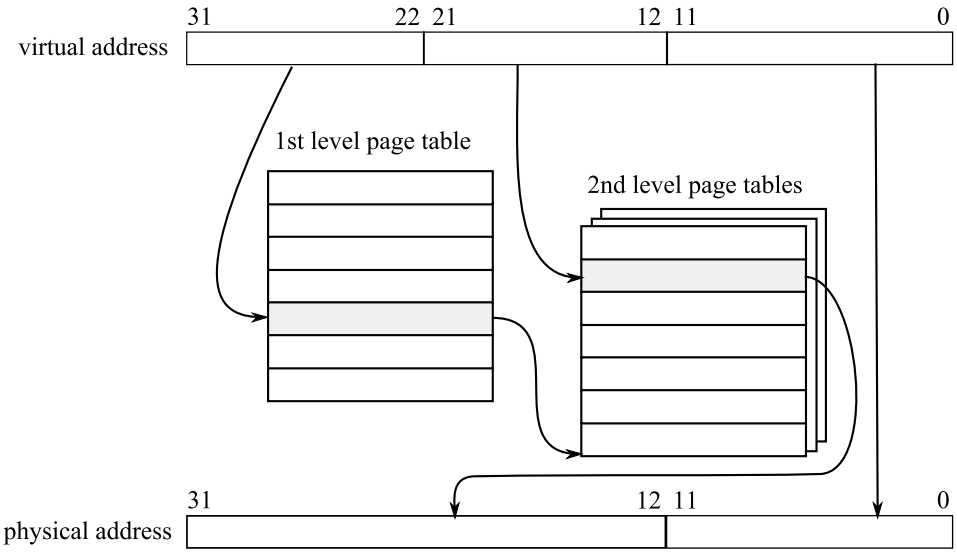
2.1.1 Stránkování

V dnešní době nejpoužívanějším způsobem mapování virtuálních adres na fyzické je technika stránkování. Virtuální adresový prostor je rozdělen na souvislé bloky stejné velikosti nazývané *stránky*. Stránky se mapují na stejně velké souvislé bloky ve fyzické paměti nazývané *rámce*. Samotný překlad virtuální adresy na fyzickou je uskutečněn průchodem datových struktur obsahujících mapování mezi virtuálními a fyzickými adresami, což může být uskutečněno hardwarově či softwarově. Nedojde-li při průchodu k nalezení cílového rámce, CPU vyvolá přerušení *výpadek stránky (page fault)* a je na uvážení operačního systému, aby tuto situaci ošetřil – např. načetl požadovanou stránku z disku nebo úlohu, která zapříčinila výpadek stránky ukončil.

Typickou implementací stránkování v hardware představují *víceúrovňové stránkovací tabulky*. Každá úroveň tabulek odpovídá určitému rozsahu bitů virtuální adresy, přičemž nejméně významné bity adresy nemají vlastní úroveň stránkovacích tabulek, ale reprezentují přímo posun (*offset*) v rámci cílového fyzického rámce. Při prohledávání se proto postupuje od nejvyšších bitů směrem k nejnižším. Nejvyšší skupina bitů adresy představuje index do stránkovacích tabulek první úrovně,



Obrázek 2.1: Funkce a umístění MMU jednotky. Schéma převzato z [35].



Obrázek 2.2: Stránkování na procesorech Intel IA-32.

v nichž je fyzická adresa tabulek druhé úrovně. V rámci druhé úrovně se jako index použije druhá skupina bitů a opět se získá fyzická adresa další úrovně tabulek. Tímto způsobem se pokračuje až do poslední úrovně, kde je uvedena přímo fyzická adresa cílového rámce.

Na obrázku 2.2 je uveden příklad stránkovacích tabulek 32bitových procesorů Intel IA-32 [16]. Virtuální adresový prostor u těchto procesorů dosahuje maximální velikosti 4 GB, a při velikosti stránky 4 KB jsou stránkovací tabulky dvouúrovňové. V první úrovni dochází k vyhledání adresy tabulky druhé úrovně, v nichž se nachází cílová adresa fyzického rámce. K této adrese se na závěr přičte dolních 12 bitů virtuální adresy a výsledkem je požadovaná fyzická adresa.

TLB

Je zřejmé, že pokud by se měl překlad adres provádět při každém přístupu CPU do (virtuální) paměti, znamenalo by to několikanásobné zpomalení celkové výkonnosti. Z tohoto důvodu je běžnou součástí MMU jednotky asociativní paměť TLB (*Translation Lookaside Buffer*). Ta funguje jako velmi rychlá vyrovnávací paměť s malým počet záznamů (např. 64 nebo 128), které lze paralelně

prohledávat. Každý záznam nese informaci o mapování stránky na rámec. Při hledání mapování MMU jednotka nejprve prohledá tuto asociativní paměť a nenalezne-li příslušné mapování, teprve pak následuje průchod mapovacími strukturami. Je-li mapování nalezeno, bude nakonec vloženo také do TLB pro zrychlení dalších přístupů na tutéž stránku.

Softwarová TLB

Pokud MMU jednotka realizuje průchod mapovacími strukturami ve vlastní režii, obvykle má pod správou také TLB paměť. U některých procesorů jako například SPARC [29] nebo MIPS [23] však návrh MMU jednotky nezahrnoval stránkovací tabulky ani jiný hardwarový mechanismus překladu adres, nýbrž pouze existenci samotné TLB. V takovém případě spravuje položky TLB přímo operační systém a opět při neexistenci mapování stránky na rámec je uvědoměn přerušením. Někdy se tato technika označuje jako *nulaúrovňové stránkování*.

Inverzní stránkovací tabulky

Běžnou implementací stránkování u systémů s 32bitovými virtuálními adresovými prostory představují stránkovací tabulky. V případě 64bitových systémů však nastává problém s velikostí stránkovacích tabulek. Možným řešením je zvýšení počtu úrovní tabulek (např. AMD64 [2] implementuje 4úrovňový model) nebo tzv. *inverzní stránkovací tabulky*, ve kterých se mapují fyzické rámce na virtuální stránky. Výhodou je malá velikost stránkovacích tabulek, protože jsou proporciální vzhledem k velikosti fyzické paměti a nikoliv virtuálního adresového prostoru. Naopak značnou nevýhodou je nemožnost použít části virtuální adresy jako indexy do stránkovacích tabulek, protože jejich primárním klíčem je fyzická adresa. Používaným řešením jsou proto hašovací funkce – vstup představuje číslo stránky a výstup je buď přímo hledaný rámec nebo kolizní řetězec možných rámců.

2.1.2 Segmentace

Jiný přístup k virtuální paměti představuje *segmentace*. Segment reprezentuje samostatný adresový prostor o dané velikosti, který se v celé své délce mapuje přímo do fyzické paměti. Každý segment je popsán bázovou adresou ve fyzické paměti, délkou a případně dalšími atributy jako např. přístupová práva. Virtuální adresa při použití segmentace je dvourozměrná – nejprve se specifikuje segment a poté posunutí (*offset*) od začátku segmentu.

Na rozdíl od stránkování, kde každá úloha má k dispozici pouze jediný adresový prostor, zde může úloha přistupovat k více prostorům současně při použití více segmentů. Toho může být např. využito při rozdělení paměti procesu – data, kód i zásobník mohou být umístěny do samostatných segmentů. Další výhodou oproti stránkování je možnost řízení přístupových práv na úrovni celých segmentů a nikoliv pouze stránek, takže např. kódový segment může být nastaven pouze pro spuštění, datový pouze pro čtení a zápis apod.

Zajímavou variantou je spojení obou přístupů – jak segmentace tak i stránkování. Typickým představitelem této varianty jsou procesory Intel IA-32. Dvousložková virtuální adresa ve tvaru *segment:offset* se nejprve překládá technikou segmentace na tzv. *lineární adresu*, která slouží jako vstup pro překlad přes stránkovací tabulky.

2.2 Využití v operačních systémech

Funkce MMU jednotky tvoří základ správy paměti v operačních systémech. Následující odstavce stručně popíší hlavní využití těchto funkcí, přičemž většina z nich se bude vztahovat k technice stránkování, jejíž výskyt dnes převažuje.

Algoritmus	Stručný popis
NRU (Not Recently Used)	vyměňuje stránku, na kterou nebyl zaznamenán žádný přístup anebo pouze přístup pro čtení
FIFO (First-In, First-Out)	vyměňuje stránku, která byla namapována jako první ze všech momentálně namapovaných stránek
Druhá šance	praktická modifikace FIFO, ve které dochází k výměně stránky jen pokud se do ní nepřistupovalo
Hodinový algoritmus	vylepšení druhé šance
LRU (Least Recently Used)	vyměňuje stránku, která byla přistupována před nejdelším časovým úsekem
NFU (Not Frequently Used)	simulace LRU softwarovými prostředky
Stárnutí	NFU s podporou stárnutí informací o přístupech

Tabulka 2.1: Výčet některých algoritmů pro výměnu stránek [35].

2.2.1 Mapování a výměna stránek

Základním stavebním kamenem správy virtuální paměti v operačních systémech je samozřejmě definice mapování mezi virtuálními adresami v adresových prostorech jednotlivých úloh a fyzickou pamětí. V případě stránkování se pro tyto účely používají dvě základní operace:

- vložení mapování – Operační systém vloží mapování stránky na rámec do příslušných překladových datových struktur (např. stránkovacích tabulek) a v závislosti na konkrétní implementaci stránkování může být nezbytné vložit toto mapování také do TLB paměti.
- zrušení mapování – Odstranění mapování z překladových datových struktur zde není postačující a vždy je nutné odstranit informaci o mapování také z TLB paměti. V případě víceprocesorových systémů je navíc nutné zajistit, že mapování je odstraněno z TLB pamětí každého procesoru, na což jsou používány různé synchronizační algoritmy.

Důležitou roli hrají také algoritmy výměny stránek. Ty se uplatní v situacích, kdy je požadavků na alokaci stránek ze strany běžících úloh více, než kolik je dostupných rámců ve fyzické paměti, a proto je nutné některé rámce uvolnit. To zahrnuje odložení obsahu některých rámců do sekundární paměti (obvykle disk) a možnost jejich opětovného načtení do paměti v případě potřeby. Právě výběr vhodných rámců pro odložení je předmětem těchto algoritmů. Přehled těch neznámějších se stručným popisem je uveden v tabulce 2.1.

2.2.2 Ochrana a izolace

Důležitým přínosem samostatných virutálních adresových prostorů je vzájemná izolace běžících úloh. Tedy každá úloha může přistupovat pouze do vlastního adresového prostoru, kde je umístěn její kód, data a zásobník. Navíc je možné také explicitně stanovit, které části (stránky) adresového prostoru smí úloha číst, zapisovat nebo spouštět, čímž ji lze chránit před jejími vlastními chybami, které by mohly způsobit např. přepsání kódu.

2.2.3 Externí fragmentace

K externí fragmentaci dochází přirozeně tam, kde je jediný adresový prostor omezené velikosti, ve kterém se provádí dynamická alokace různě velkých bloků paměti. Postupem času tak dochází k fragmentaci původně volného prostoru a vzniku děr mezi alokovanými bloky, které jsou důsledkem uvolňování již nepotřebných bloků. Ne vždy je možné použít tyto díry pro alokaci dalších bloků

hlavně z důvodu jejich různé velikosti. Protože se jedná o paměť externí vůči alokovaným blokům, mluvíme o tzv. *externí fragmentaci*.

Použití stránkování zcela zamezuje externí fragmentaci fyzické paměti, protože jednotkou alokace fyzické paměti je rámec pevné velikosti (typicky v jednotkách kB). Navíc bude-li úloha ve svém virtuálním adresovém prostoru alokovat souvislý blok několika stránek, jejich mapování do fyzické paměti nemusí představovat souvislý blok rámců, a proto je možné namapovat tyto stránky na kterékoli volné rámce fyzické paměti.

2.2.4 Paměťově mapované soubory a sdílená paměť

Elegantní formou práce se soubory jsou paměťové mapované soubory. Při otevírání souboru požádá aplikace operační systém o namapování jeho obsahu do svého adresového prostoru a poté místo volání funkcí čtení či zápisu přímo přistupuje do své paměti, kde má celý soubor k dispozici. To je umožněno opět díky technice stránkování a přerušení informujícím o *výpadku stránky (page fault)*. Při otevření souboru je typicky pouze vytvořena informace o mapování příslušných stránek a až v případě přístupu úlohy na tyto stránky (což vyvolá zmiňované přerušení výpadku stránky) dochází k načítání odpovídajících bloků souboru do rámců fyzické paměti a vytvoření mapování mezi nimi a stránkami. Po zavření souboru nebo ukončení úlohy jsou změněné stránky uloženy zpět na disk do souboru.

Paměťově mapované soubory se také dají využít pro efektivní meziprocesovou komunikaci a sdílení paměti. Pokud oba komunikující procesy mají namapován stejný soubor, pak každý zápis jednoho procesu je okamžitě viditelný v paměti druhého procesu. Tento princip komunikace je typický u systémů unixového typu.

Sdílení paměti lze samozřejmě dělat efektivně přímo ve fyzické paměti i bez přítomnosti MMU jednotky – k tomu zcela postačuje, aby si komunikující procesy sdělily adresu sdíleného bloku. Přidanou hodnotou použití virtuální paměti je však možnost stanovení přístupových práv a dalších atributů sdílených stránek. Tedy jeden proces může např. data sdílet v režimu čtení i zápisu, zatímco ostatní pouze v režimu čtení.

2.2.5 Sdílený kód

Paměťové mapování souborů lze kromě sdílení dat mezi procesy využít i ke sdílení kódu aplikací resp. knihoven. Při spuštění každé nové instance aplikace operační systém přidá mapování (s právy jen pro čtení a spuštění) do stránkovacích tabulek na fyzické rámce, kde je namapována první instance. V případě nekonstantních dat aplikace či sdílených knihoven je situace složitější, protože každá instance musí mít svou vlastní kopii. Tento problém lze řešit při spouštění aplikace vytvořením samostatného mapování těchto dat nebo až v případě pokusu aplikace o přístup k nim technikou *copy on write*[1]. To znamená, že nejprve jsou všechny stránky označeny příznakem *copy on write* a v případě prvního pokusu o zápis dat se provede vytvoření kopie stránky i fyzického rámce, kde budou data zapsána a kde může aplikace dále v zápisu dat pokračovat.

Např. operační systémy unixového typu implementují tento mechanismus dokonce při každém spouštění nového procesu, což sestává z vytvoření kopie původního procesu systémovým voláním `fork()`(pomocí *copy on write*) a až poté načtení obrazu nového procesu dalšími systémovými voláními.

2.2.6 Detekce přetečení zásobníku

Přetečení zásobníku se projeví pokusem procesu o zápis dat mimo stránky vyhrazené zásobníku. Reakcí operačního systému může být ukončení takového procesu anebo rozšíření zásobníku přidáním

[1] Český překlad *kopírování při zápisu* není natolik známý ani výstižný, a proto se v textu budeme držet originálního anglického termínu.

dalších mapování stránek na fyzické rámce. Používanou heuristikou zda-li se má zásobník opravdu rozšířit je kontrola, zda je paměťový zápis dostatečně „blízko“ poslední stránky zásobníku. Tím se ošetří situace náhodného zápisu procesu do paměti, který by nechtěně způsobil rozšíření zásobníku namísto řádného ukončení chybně pracujícího procesu.

2.2.7 Detekce nulového ukazatele

Častým prostředkem ochrany procesů před jejich chybnou funkcí je také detekce paměťového přístupu na adresu 0. Ten bývá obvykle způsobený dereferencí nulového ukazatele, který nebyl procesem explicitně inicializován. Technicky je to zařízeno vyhrazením první stránky adresového prostoru pro tyto účely.

Kapitola 3

Paměťová ochrana bez MMU

Přestože dnešní procesory pro desktopové a serverové systémy disponují výkonnými MMU jednotkami, najdeme i dnes celou řadu procesorů, zejména v oblasti *vestavných systémů* (*embedded systems*), které tuto jednotku postrádají. Její absence bývá záměrná a hlavními důvody jsou nižší cena procesoru, jeho menší fyzická velikost, energetická spotřeba a také výkonnost. Protože výzkum v oblasti vestavných systémů je dnes stále populárnější a stále se rozvíjí, existuje řada výzkumných pracovišť na univerzitách i v komerčních firmách, které se zaměřují na problémy spojené s chybějící podporou virtuální paměti poskytovanou MMU a snaží se najít přijatelná řešení pro praktické použití.

Je potřeba však zdůraznit, že v případě vestavných systémů by plná funkcionalita MMU jednotky nebyla obvykle využita a naopak byla spíše na obtíž zejména v případě stránkování. Tyto systémy mají velmi omezené zdroje a často vůbec nedisponují sekundárním úložištěm dat jako je pevný disk u desktopových systémů, kde by mohly stránky odkládat v případě přeplnění fyzické paměti. Dalším problémem jsou stránkovací tabulky zabírající drahocenný prostor v paměti. Navíc u systémů reálného času (*real-time systems*) může být problematický nedeterministický překlad virtuální adresy na fyzickou, pro který je v případě nenalezení mapování v TLB paměti nutné provést průchod stránkovacími tabulkami, čehož si musí být návrháři takových systémů vědomi.

Naopak velmi žádanou funkcí MMU jednotky je izolace procesů použitím více adresových prostorů, na což je však potřeba jistá forma virtuální paměti, která bohužel není k dispozici. Hardwarovým nebo softwarovým řešením bývá obvykle kompromis v podobě jediného adresového prostoru, ve kterém je možné alespoň částečně izolovat běžící procesy za použití určitých prostředků paměťové ochrany. Přehled takových prostředků je předmětem této kapitoly.

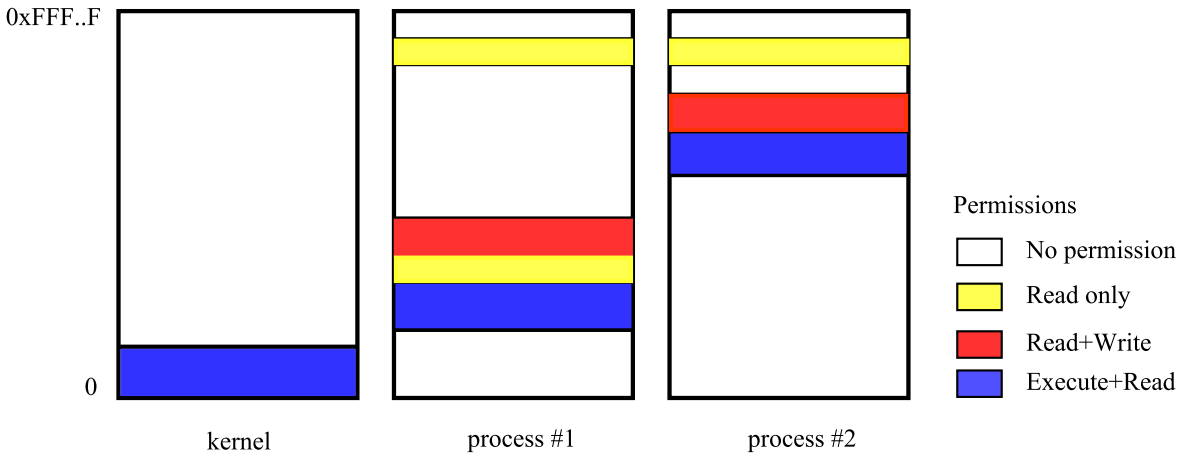
3.1 Hardwarové prostředky

3.1.1 MPU

Přirozenou alternativou k MMU je MPU (*Memory Protection Unit*), nebo-li jednotka ochrany paměti. Pokud je přítomna, pak se nachází přímo v procesoru a umožňuje kontrolovat každý přístup aktuálně běžícího kódu do fyzické paměti. Tedy nenabízí žádný mechanismus virtuální paměti, nýbrž pouze kontrolu nad paměťovými přístupy.

Základním prvkem ochrany je (*paměťový region memory region*) představující souvislou oblast fyzické paměti určenou počáteční a koncovou adresou a právy přístupu – obvykle čtení, zápis, případně i spuštění. Počet těchto regionů bývá shora omezen – např. 8 nebo 16. V závislosti na konkrétním modelu MPU se mohou regiony překrývat, mít přiřazeny různé priority apod.

Podobně jako v případě MMU provádí správu MPU jednotky operační systém, a to skrze registry procesoru. Pro každý proces i jádro může vyhradit samostatnou skupinu paměťových regionů (např. pro kód, data a zásobník) a zajistit tak jejich vzájemnou izolaci. Pokusí-li se proces zapsat data



Obrázek 3.1: Možné přiřazení paměťových regionů procesům a jádru systému, schéma převzato z [40].

do paměti mimo své paměťové regiony, bude vyvoláno přerušení procesoru podobně jako u MMU jednotky a operační systém dostane možnost její obsluhy.

Příklad možného rozdělení paměťových regionů je uveden na obrázku 3.1. Zatímco jádro systému v levé části schématu má povolen přístup pro čtení a zápis téměř do celé fyzické paměti, procesy v pravé části jsou omezeny pouze na svůj kód, konstantní data a aplikační data. Můžeme si všimnout, že oba procesy mají společný přístup k malému bloku v horní paměti, který může reprezentovat sdílenou paměť.

Problémem většiny MPU jednotek je nemožnost přiřadit určitý paměťový region konkrétnímu procesu tak, aby v čase jeho běhu byly aktivní pouze jemu přiřazené regiony. U systémů vybavených MMU obsahuje procesor speciální registr obsahující tzv. *paměťový kontext* reprezentující obvykle aktuální virtuální adresový prostor procesu. Jeho hodnotou je buď přímo ukazatel do fyzické paměti na stránkovací tabulky, nebo unikátní identifikátor adresového prostoru. Liší se pro každý proces a při přepínání kontextu procesů přepíná operační systém také paměťový kontext.

Procesory s MPU obvykle žádný takový mechanismus nenabízejí, a proto je nutné dělat přepnutí paměťového kontextu programově, tj. smazáním původního obsahu registrů paměťových regionů a naplněním pouze těmi regiony, do kterých má proces naplánovaný k běhu přístup.

MPU u procesorů ARM

ARM [3] je rodina 32bitových RISCových procesorů, jejichž uplatnění najdeme hlavně ve vestavěných systémech např. v odvětví mobilní spotřební elektroniky. Jedná se o velmi populární a poměrně rozsáhlou rodinu procesorů vybavených MMU nebo MPU jednotkou a dokonce i modely obsahující obě tyto jednotky současně, přičemž sám operační systém může zvolit, kterou z nich bude využívat.

MPU definuje 8 paměťových regionů pomocí těchto vlastností:

- bázová adresa: musí být zarovnána na velikost regionu,
- velikost regionu: násobek mocniny dvou v rozsahu 4 KB až 4 GB,
- atributy: popisují vztah regionu a vyrovnávacích pamětí (write through, write back vlastnosti), zda je na region mapováno externí zařízení apod.,
- přístupová práva: čtení anebo zápis.

Přístupová práva se navíc liší v zavislosti na režimu běhu procesoru. Dle potřeby je tedy možné nakonfigurování práv takové, kdy jádro systému běžící v privilegovaném režimu smí do regionu zapisovat i z něj číst, zatímco aplikace běžící v uživatelském režimu má právo pouze pro čtení.

Regiony jsou očíslovány 0 až 7 a každému z nich je přiřazena priorita odpovídající jeho číslu. Region s nejvyšší prioritou má číslo 7, nejnižší má pak s číslem 0. Priority hrají roli při porovnávání adresy paměťového přístupu s adresy regionů. Přednostně jsou brány v úvahu regiony s vyšší prioritou a nespadne-li porovnávaná adresa do žádného z nich, následuje porovnání s regiony priority o jedna menší atd. Tímto způsobem je vybrán region, jehož atributy a přístupová práva jsou aplikovány na paměťový přístup.

V závislosti na konkrétním procesoru se může lišit počet regionů a nastavitelné atributy, princip výběru regionu a přístupová práva však zůstávají stejná. Příkladem procesoru s 16 paměťovými regiony může být procesor ARM1156T2F-S navržený pro operační systémy reálného času.

MPU u procesorů Blackfin

Dalším představitelem procesorů s vestavěnou MPU jednotkou je rodina Blackfin [5] od firmy Analog Devices. Tyto procesory jsou zvlášť vhodné pro vestavěné systémy náročné na datové výpočty jako např. kódování videa, zvuku či zpracování obrazu díky konceptu digitálního signálního zpracování.

MPU jednotka je v dokumentaci představována jako jednotka správy paměti, přestože žádný překlad virtuálních adres na fyzické ani mechanismus stránkování zde není podporován. Autoři se však drží terminologie používané u MMU jednotky, a proto můžeme v dokumentaci najít termíny jako stránka (označující paměťový region), stránkovací tabulky (registry definující vlastností paměťových regionů) apod.

Oproti architektuře ARM definuje Blackfin oddělené datové a instrukční regiony. Celkový počet regionů pro každou z těchto skupin je 16, přičemž jejich velikosti jsou omezeny na 1 KB, 4 KB, 1 MB nebo 4 MB. Technická dokumentace nepopisuje zda se regiony mohou překrývat, a který region se použije v případě, kdy adresa padne do dvou regionů současně.

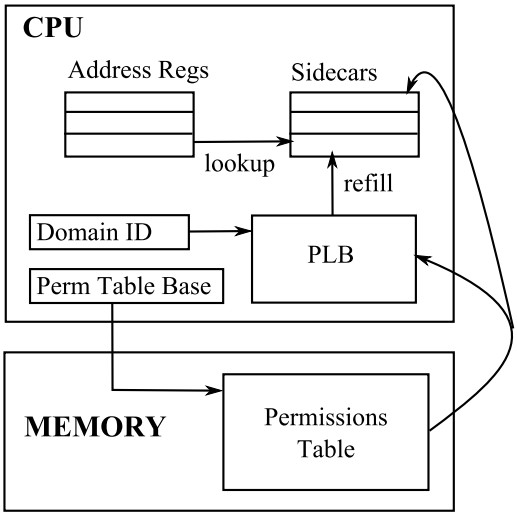
Popis regionů se uchovává v tzv. CPLB registrech (*Cacheability Protection Lookaside Buffer*) v horní části adresového prostoru fyzické paměti. To zjednodušuje práci operačnímu systému při přepínání paměťového kontextu procesů – na krátký okamžik vypne funkcionalitu MPU jednotky, nakopíruje do paměťově mapovaných registrů nové hodnoty odpovídající nastavení procesu naplánovaného k běhu a nakonec opět uvede MPU jednotku v činnost.

MMU jako MPU

Architektura některých MMU jednotek byla s ohledem na energetickou spotřebu navržena tak, aby bylo možné vypnout drahé operace překladu virtuálních adres na fyzické, ale přitom se do jisté míry zachovaly prvky ochrany na úrovni větších paměťových bloků. Příkladem takového procesoru je opět ARM s MMU jednotkou 2úrovňového stránkování. První úroveň tvoří 16 položek, každá popisující 1MB bloky paměti nebo-li sekce, které jsou dále rozmělněny do 4KB bloků v druhé úrovni stránkovacích tabulek. Vynecháme-li tuto 2. úroveň, dostáváme v podstatě velmi jednoduchou MPU jednotku s 16 paměťovými regiony resp. sekcemi s pevně danými bázovými adresami i velikostí.

3.1.2 Mondrianská ochrana paměti

Mondrianská ochrana paměti [40] (*Mondrian memory protection*, dále jen MMP) se stejně jako MPU zaměřuje na kontrolu paměťových přístupů v rámci jediného adresového prostoru reprezentovaného fyzickou pamětí. Granularita, s jakou lze ochranu specifikovat, přitom může dosahovat až 4bytových slov, což je v porovnání s technikami stránkování velký rozdíl. Lze ji navíc provozovat i nad virtuální pamětí jako nadstavbu nad MMU jednotkou, a z funkčního hlediska je proto použitelná i ve stávajících systémech. Její nevýhodou jsou samozřejmě citlivé zásahy do hardware procesoru.



Obrázek 3.2: Hlavní komponenty mondrianské ochrany paměti [40].

Architektura MMP

Základním prvkem ochrany je paměťový region nazývaný *segment*. Segmenty jsou přiřazeny do tzv. *domény ochrany* (*protection domain*) reprezentující entitu běhu operačního systému, typicky vlákno. Každá entita má přiřazenu právě jednu doménu ochrany, přičemž více entit může sdílet jednu doménu ochrany – uvažme např. vlákna patřící stejnému procesu. Obdobně i jeden segment může být sdílen více doménami ochrany, čímž lze například implementovat sdílení paměti mezi procesy.

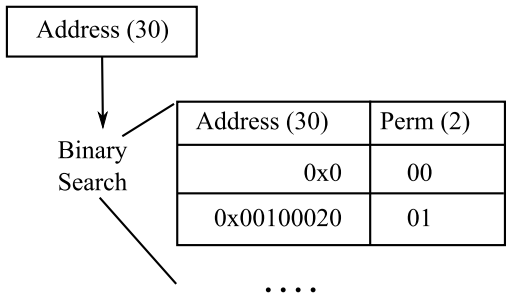
Na obrázku 3.2 jsou zobrazeny hlavní komponenty mondrianské ochrany paměti. CPU registry procesoru jsou zde rozšířeny o registr `Domain ID` identifikující doménu aktuálně běžícího vlákna a registr `Perm Table Base`. Ten obsahuje ukazatel na tabulku práv (*Permissions Table*) pro aktuální doménu (reprezentovanou běžícím vláknem). Samotná tabulka je spravována operačním systémem a umístěna mimo dosah uživatelských procesů.

Celý mechanismus ochrany je následující. U každého paměťového přístupu musí být zkontrolováno, má-li aktuální doména dostatečná přístupová práva. To je dosaženo průchodem tabulky práv (harwarově nebo softwarově) a nalezením odpovídající položky popisující práva segmentu, do kterého kontrolovaná adresa náleží. Pro urychlení této procedury se používá vyrovnávací paměť PLB (*Permissions Lookaside Buffer*) – analogie k TLB paměti. Průchod tabulkami práv se tedy provádí pouze v případě nenalezení hledané adresy v PLB paměti. Další optimalizaci představují tzv. *přívěsné registry* (*sidecar registers*). Každému adresovému registru[1] CPU jednotky je přiřazen jeden přívěsný registr uchovávající údaje o segmentu naposledy přistupovaného adresovým registrem. To urychluje kontrolu práv v situacích, kdy proces provádí průchod proměnné typu pole po jednotlivých prvcích inkrementací ukazatele v adresovém registru. Zde není dokonce nutný ani přístup do PLB paměti. Je-li adresa paměťového přístupu mimo segment v přívěsném registru a není nalezena ani odpovídající položka v PLB paměti, dochází k průchodu tabulky práv a po nalezení hledané položky k operaci naplnění PLB (*PLB refill*). Ta je však vykonána také na přívěsných registrech.

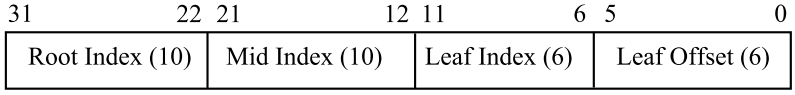
Tabulka práv

MMP poskytuje dva způsoby uchování tabulky práv v paměti. Prvním z nich je tzv. *uspořádaná tabulka segmentů* (*Sorted Segment Table*). Položky s popisem segmentů jsou uloženy lineárně za

[1] Adresový registr je takový registr, který může být součástí operandu s adresou v instrukci pracující s pamětí.



Obrázek 3.3: Uspořádaná tabulka segmentů [40].



Obrázek 3.4: Rozdělení adresy pro průchod víceúrovňovou tabulkou práv [40].

sebou v poli a seřazeny podle bázových adres segmentů. Segmenty mohou být libovolné velikosti, minimálně však 4 byty, a nesmějí se překrývat. Příklad uspořádané tabulky segmentů je uveden na obrázku 3.3. Při uvažování 32bitového adresového prostoru je každá položka složena ze dvou částí: 30bitové bázové adresy segmentu a 2bitové hodnoty s právy pro segment (žádná práva, čtení, čtení a zápis, čtení a spuštění). Délka segmentu resp. jeho koncová adresa je vždy vypočtena implicitně z bázové adresy následujícího segmentu. Výjimkou je samozřejmě poslední položka tabulky.

Výhodou tohoto uložení informací o segmentu je snadné nalezení segmentu pro zadanou adresu pomocí algoritmu binárního vyhledávání. Naopak značná nevýhoda se projeví při nutnosti vložení či smazání záznamu o segmentu, která vede na přesun větších bloků paměti. Za přepokladu malého počtu segmentů se však jedná o zanedbatelnou nevýhodu.

Jiné negativum představuje sdílení segmentů, které v modelu s uspořádanou tabulkou segmentů není proveditelné. Domény ochrany mohou sdílet buď všechny segmenty (a mít tak totožnou tabulku segmentů) nebo nemohou sdílet žádnou položku popisující segment.

Tyto nevýhody odstranila druhá metoda nazvaná *víceúrovňová tabulka práv* (*Multi-level Permissions Table*), jejíž princip je podobný jako u stránkovacích tabulek MMU jednotky. Vstupní adresa je při uvažování 32bitového adresového prostoru rozdělena na 4 části (viz obrázek 3.4):

- Root index: index do tabulky první úrovně s 1024 položkami, každá poskytuje mapování pro oblast paměti o velikosti 4 MB.
- Mid index: index do tabulky v rámci druhé úrovně, zajišťuje mapování 4KB oblastí paměti. Případně je možné tuto úroveň vynechat a místo indexu přímo zapsat práva (po 2 bitech jako u metody uspořádané tabulky segmentů) pro 8 paměťových oblastí, každý o velikosti 512 bytů.
- Leaf index: index do tabulky poslední úrovně s 64 položkami. Každá položka popisuje přístupová práva pamětových bloků o velikosti 64 bytů.
- Leaf offset: posunutí v rámci segmentu.

Podobně jako u stránkovacích tabulek i zde je nutné provést průchod všemi úrovněmi tabulek pro nalezení informací o segmentu, do kterého příslušná adresa patří. Tabulky jsou umístěny ve fyzické paměti, nejlépe mimo dosah uživatelských procesů.

Sdílení tabulek nižších úrovní lze provést pouze tím, že tabulkám vyšší úrovně různých domén ochrany nastaví stejné indexy na nižší úrovně tabulek. Tímto způsobem lze velmi efektivně zajistit sdílení paměti mezi procesy či jádrem systému při minimální režii na pomocné datové struktury. Pokud je MMP nadstavbou nad virtuální pamětí, je navíc možné, aby v každé doméně ochrany byla virtuální bázová adresa sdíleného bloku různá, avšak tabulky práv (od jisté úrovně) byly shodné.

3.2 Softwarové prostředky

Zásahy do schématu procesorů jsou v mnohých případech nákladné a u systémů, které byly již uvedeny do provozu velmi obtížné. Nejen z tohoto důvodu se výzkum zaměřil na softwarové možnosti emulace či náhrady funkcí MMU. Kromě vestavných systémů může mít takový výzkum uplatnění i v klasických monolitických operačních systémech jako je Windows či Linux. Ovladače zařízení těchto systémů jsou implementovány jako rozšíření jádra (*kernel extensions, kernel modules*), která se při svém načtení a spuštění stávají součástí jeho adresového prostoru. Jejich chybná činnost pak může způsobit pád jádra i celého systému. Téměř totožný problém lze najít u rozšiřitelných aplikací jako jsou např. prohlížeče Window Explorer či Mozilla, které načítají různá rozšíření nebo-li *pluginy* do svého adresového prostoru pro získání nové funkcionality (přehrávače videa, vykreslování obrázků) a komunikují s nimi daným rozhraním. Obvykle však pluginu nic nebrání v tom, aby přepsal data nebo i kód své hostitelské aplikace.

V této části textu budou uvedeny některé z prostředků pro softwarovou ochranu, které mohou pomoci zabránit výše popsaným problémům. V principu lze tyto prostředky rozdělit do dvou skupin: bezpečné programovací jazyky a techniky softwarové izolace.

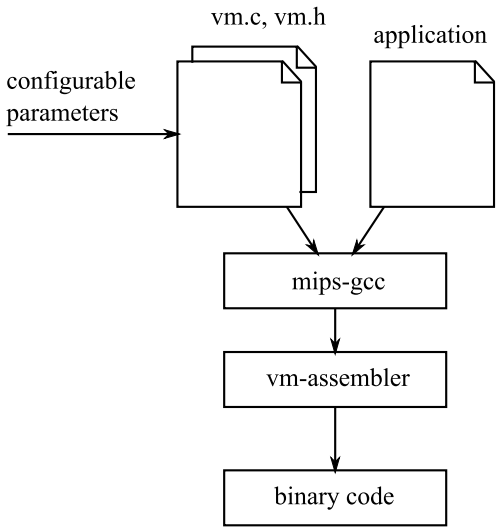
3.2.1 Bezpečné programovací jazyky

Termín *bezpečné programovací jazyky* označuje skupinu programovacích jazyků založených na silné typové kontrole a intenzivních kontrolách při běhu aplikace, např. při přístupu k položkám pole se zjišťuje, zda index není mimo povolený rozsah celého pole. Neumožňují programátorovi nízkoúrovňové operace jako je přímý přístup do paměti nebo přetypování ukazatelů. Jedná se o tzv. *sandboxing*, kdy aplikace je uzavřena do pomyslné ohrady a s okolním prostředím komunikuje pouze přes dané rozhraní. K tomu je však zapotřebí spolehlivý systém běhového prostředí (*run-time environment*) napsaný v některém z nízkoúrovňovějších jazyků jako např. C. Je-li chyba v jeho implementaci, pak i samotná aplikace řízená tímto prostředím může porušit principy ochrany a např. zapsat data do jí nepovolených paměťových oblastí.

Klasickým příkladem bezpečných programovacích jazyků jsou Java nebo C#, i když také v nich lze najít konstrukce porušující pravidla ochrany (např. *unsafe* ukazatele v C#). Doménou těchto jazyků jsou však spíše desktopové počítačové systémy.

Vzhledem k tomu, že naprostá většina aplikací pro vestavěné systémy je psána v jazyce C a C++, objevilo se několik projektů snažících se rozšířit je přidáním bezpečnostních prvků. *Cyclone* [8] představuje takové rozšíření jazyka C. Zavádí prvky ochrany zejména pro ukazatele, pole a datové struktury deklarované pomocí `struct` nebo `union`. Klasický ukazatel je označen jako *fat* a kromě cílové adresy obsahuje také spodní a horní hranici bloku, do kterého ukazuje. Při dereferenci se kontroluje jeho nenulovost a zda ukazuje na adresu v rámci hranic daného bloku. Také kontrolu nad haldou aplikace přebírá „běhové“ prostředí Cyclonu a nedovoluje tak aplikaci manipulovat s haldou přímo, nýbrž pouze přes tzv. regiony, jejichž definici musí programátor poskytnout přímo ve zdrojovém kódu.

Control-C [18] je naopak podmnožinou jazyka C a dává přednost statické analýze aplikace před kontrolami za běhu a spoléhá se na pokročilé a optimalizační techniky překladače. Klade důraz na silně typované výrazy, zákaz přetypování jiných typů na ukazatele, inicializace ukazatelů ihned při jejich definici apod. V případě kdy paměťové operace závisejí na podmínkách známých pouze za běhu aplikace, navrhuje dodatečné vkládání běhových kontrol.



Obrázek 3.5: Princip překladu aplikací pro softwarovou MMU [14].

Doposud zmiňované jazyky používaly standardní přístup kompilace aplikací. Tedy překlad do assembleru, objektových souborů a nakonec slinkování do výsledného spustitelného programu. Jazyky *CCured* [7] a *Safe-C* [4] jsou rovněž deriváty C, avšak jsou implementovány jako takzvané *source-to-source* překladače; tedy vstupem je zdrojový soubor C a výstupem jeho pozměněná verze. Principem je důsledná typová kontrola a vkládání vhodných běhových kontrol, které navíc mohou využít optimalizačních technik následného překladu a minimalizovat tak režii na běh těchto kontrol.

3.2.2 Softwarová MMU

Zajímavou technikou je pokus o čistě softwarovou emulaci MMU jednotky navrženou pro vestavěné systémy [14]. Principem je překlad adres v každé instrukci paměťového čtení a zápisu z virtuální na skutečnou fyzickou za pomocí techniky stránkování implementované ryze softwarově.

Struktura celého systému emulace se nachází na obrázku 3.5. Vstupem jsou zdrojové kódy aplikace v jazyce C/C++, ke kterým se připojuje knihovna *vm.c* s implementací překladu virtuálních adres na fyzické v čase běhu aplikace. Konfigurační nastavení jako velikost stránkovacích tabulek, velikost dostupné RAM paměti apod. jsou umístěny v hlavičkovém souboru *vm.h*. Všechny zdrojové kódy jsou přeloženy pomocí *gcc* [10] překladače do assembleru a následně zpracovány tzv. *vm-assemblerem*. Ten provádí binární instrumentaci instrukcí přistupujících do paměti tak, že jsou volány překladové funkce z knihovny *vm.c* a jejich výstup nakonec použit při běhu těchto instrukcí. Výstupem vm-assembleru je na závěr spustitelný binární soubor aplikace.

Celý systém byl autory této metody vyzkoušen na platformě MIPS se simulátorem s EEPROM a Flash pamětmi, které sloužily jako sekundární úložiště pro stránky odmapované z virtuálního adresového prostoru.

Ve snaze zoptimalizovat běh aplikace vytvořili autoři této metody dvě vylepšení. První z nich nazvané *virtuální paměť s fixní adresací* rozděluje virtuální paměť na regiony označené jako virtuální a na regiony bez tohoto označení, přičemž překlad virtuální na fyzickou adresu se provádí pouze u regionů virtuálních. Tato metoda sice zabraňuje zbytečnému překladu tam, kde není nutný, ale stále vyžaduje kontrolu u každého paměťového přístupu pro zjištění, zda-li adresa spadá do virtuálního regionu. Druhou metodou je *selektivní virtuální paměť*, která nabízí lepší granularitu z hlediska objektů pro virtualizaci. Její podstatou je označení těch datových struktur ve zdrojovém kódu, které mohou benefitovat z mechanismu virtuální paměti a mají být virtualizovány. Typicky se to bude týkat velkých datových struktur, které případně mohou překročit velikost RAM paměti

a použití virtuální adresace je pro ně nezbytné. Technicky je tato anotace provedena preprocesorovou direktivou `#pragma`.

Velkou nevýhodou této emulace MMU je značné ovlivnění rychlosti instrumentované aplikace. Je-li použita technika překladu všech paměťových přístupů, zpomalení může být až desetinásobné. Virtuální paměť s fixní adresací či selektivní virtuální paměť sice vykazují lepší výsledky, avšak stále se jedná o degradaci rychlosti ve stovkách procent.

3.2.3 SFI

Prvním představitelem metod softwarové izolace je *SFI* (*Software-based Fault Isolation*) [38] z roku 1993. Ta sa zaměřuje na izolaci pluginů a hostitelské aplikace, které sdílejí společný adresový prostor a používá k tomu metod binární instrumentace, tj. přímé úpravy binárního kódu aplikace.

Adresový prostor je rozdělen na segmenty tak, aby adresy v rámci stejného segmentu sdílely unikátní kombinaci horních bitů nazývaných *identifikátor segmentu*. Pro každý načtený plugin jsou zavedeny dva disjunktní segmenty: jeden pro jeho kód a druhý pro data, haldu a zásobník. Souhrně jsou tyto dva segmenty označovány jako *chybová doména* (*fault domain*). Principem SFI techniky je tzv. *sandboxing*, který u každé strojové instrukce pluginu zapisující data do paměti vynucuje, aby adresa zápisu spadala do segmentu chybové domény pluginu. Technicky je to provedeno nastavením identifikátoru segmentu s daty pluginu přímo do adresy zápisu. Tento postup samozřejmě není aplikován u zápisů, u kterých lze díky statické analýze kódu ověřit, že skutečným cílem je datová oblast pluginu (typicky globální proměnné).

Pseudokód sandboxing techniky je následující:

```
dedicated-register := target-register & mask-register
dedicated-register := dedicated-register | segment-register
instrukce zápisu dat na adresu v~dedicated-registeru
```

`dedicated-register` představuje rezervovaný registr procesoru, do kterého je nejprve uložen offset adresy zápisu pomocí operace binárního součinu s `mask-registerem` a následně operací binárního součtu se `segment-registerem` doplněn správný identifikátor segmentu pluginu. Analogický postup je použit u sandboxingu instrukcí nepřímých skoků a návratů z podprocedur, kdy `segment-register` obsahuje identifikátor kódového segmentu pluginu a `mask-register` zůstává stejný. Protože segmenty v rámci chybové domény jsou disjunktní a každý plugin má přiřazenu jedinou chybovou doménu, plugin nemá možnost ovlivnit mateřskou aplikaci porušením jeho datových struktur či skokem na libovolné místo jeho kódu. Zápisy do paměti jsou vždy prováděny pomocí vyhrazeného registru `dedicated-register`, který vždy ukazuje do datového segmentu chybové domény. Nic na tom nezmění ani pokus pluginu skočit doprostřed kódu či za kód implementující sandboxing.

Zřejmou nevýhodou je použití 2 resp. 3 rezervovaných registrů procesoru, které se nesmí vyskytovat v původním strojovém kódu pluginu, což také vyžaduje překladač schopný generovat kód bez těchto registrů. Tyto předpoklady lze poměrně snadno zajistit na procesorech RISCového typu disponující obvykle velkým počtem (cca 16 až 32) registrů obecného použití (*general purpose registers*), kde generování strojového kódu bez použití několika rezervovaných registrů nebude mít znatelný dopad na celkovou výkonnost aplikace.

Z popisu techniky také plyne, že velikost každého segmentu je dána při spuštění pluginu a měnit ji v průběhu běhu je problematické. Navíc autoři předpokládají stejnou velikost všech strojových instrukcí, což na architekturách typu CISC není obecně splněno. Následkem toho může při chybné funkci pluginu dojít ke skoku doprostřed kódu strojové instrukce, který může reprezentovat začátek zcela jiné sekvence strojových instrukcí obsahující např. zápisy do ostatních segmentů či přímo paměti hostitelské aplikace.

Navazující výzkum

Princip SFI se stal základem několika dalších technik softwarové izolace, z nichž jmenujme alespoň dvě:

- *MiSFIT* (*Minimal i386 SFI Tool*) [27] – nástroj pro softwarovou izolaci modulů v jádře výzkumného operačního systému VINO [28]. Pracuje pouze na architektuře x86.
- *Harbor* [19][20] – implementace SFI na RISCových procesorech v sensorických sítích, kde zajišťuje ochranu modulů současně běžících na stejných uzlech senzorické sítě.

3.2.4 XFI

Pokročilejší a propracovanější techniku představuje XFI[2] [9] vyvinutá výzkumníky z Microsoft Research. Jejím nástrojem pro softwarovou izolaci je opět instrumentace binárního kódu a je primárně určena pro ochranu hostitelské aplikace od pluginů či modulů načítaných do jejího adresového prostoru na operačních systémech Windows. XFI stojí na dvou základních stavebních kamenech:

- integrita toku řízení,
- vkládání běhových kontrol pro paměťové přístupy i nepřímé skoky.

CFI

Integrita toku řízení nebo-li *CFI* (*Control Flow Integrity*) [1] je jeden z postupů jak zabránit útočníkovi převzít řízení nad posloupností zpracovávaných instrukcí aplikace. Klasickým příkladem takového útoku je přetečení vyrovnávacích pamětí (tzv. *buffer overflow attack*), jehož principem je přepis lokálního pole procedury na zásobníku. Jelikož kromě lokálních proměnných se na zásobník ukládají také návratové adresy volaných procedur, vhodným přepisem paměti za těmito lokálními proměnnými může útočník modifikovat návratovou adresu a změnit tak tok řízení aplikace bez jejího vědomí. Obdobný útok lze provést vhodným přepisem tabulek virtuálních metod, které jsou využívány při volání virtuálních funkcí. Tato volání jsou implementována pomocí instrukcí nepřímých skoků, kdy cílová adresa skoku je načtena z paměti (tabulky virtuálních funkcí), a proto nelze cíl skoku ověřit statickou analýzou.

Dodržením integrity toku řízení dostáváme jistotu, že (napadená) aplikace nebude schopna obejít vložené běhové kontroly a zapsat tak v případě chyby či úspěšného útoku data do jí nepřístupných paměťových oblastí. Dalším důležitým hlediskem je umožnění optimalizací ve vkládání běhových kontrol. Máme-li totiž sekvenci instrukcí, ve kterých se několikrát po sobě zapisuje do stejné paměťové oblasti (např. v rozmezí několika bytů), pak stačí vložit kontrolu na začátek této sekvence místo kontroly před každým samostatným zápisem. Ušetří se jak velikost výsledného kódu aplikace, tak i procesorový čas. Podmínkou takové optimalizace je samozřejmě kvalitní statická analýza kódu, která zaručí, že adresy uvnitř této sekvence nejsou cílovou adresou žádné instrukce skoku aplikace. V teorii překladačů se takový úsek kódu označuje jako *základní blok* (*basic block*). Problém zde však může nastat u nepřímých skoků, kde nelze cílovou adresu zjistit pouhou statickou analýzou – tato problematika je popsána dále.

Cílem CFI je tedy zabránit právě takovým útokům. Jako první nástroj navrhuje zavedení tří nových instrukcí:

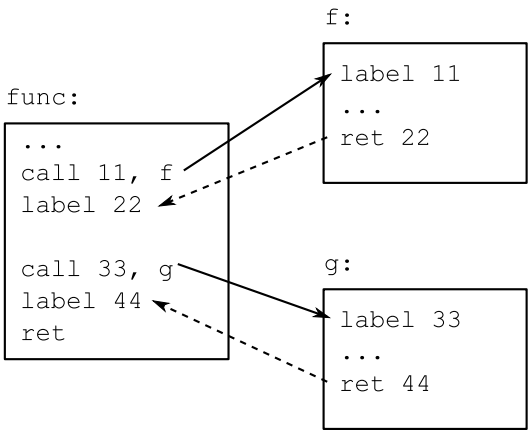
- `label ID` – instrukce bez vedlejších efektů s jediným argumentem ID,
- `call ID, dest` – upravená verze instrukce nepodmíněného volaní procedury, volání se provede pouze v případě pokud je před cílovou adresou dest uložena instrukce label ID se stejným identifikátorem ID,

[2] Přesný význam zkratky XFI nebyl autory uveden, lze se však domnívat, že se jedná o *Extended Fault Isolation*

```
int f(int x, int y)
{
    return x + y;
}

int g(int x, int y)
{
    return x - y;
}

void func(int a, int b)
{
    f(a, b);
    g(b, a);
}
```



Obrázek 3.6: Příklad použití speciálních instrukcí pro vynucení integrity toku řízení [1].

- `ret ID` – upravená verze instrukce návratu z podprocedury, návrat se uskuteční pouze pokud na návratové adrese leží instrukce label ID se stejným identifikátorem ID.

Pokud se při provádění instrukce `call ID,dest` nebo `ret ID` neshodují cílové hodnoty ID, instrukce není vykonána a místo toho je vyvolána výjimka, jelikož je podezření, že došlo k narušení integrity toku řízení. Příklad použití těchto instrukcí je na obrázku 3.6.

Problémem této metody je určení ID hodnoty u instrukcí `ret ID`, protože ne vždy lze statickou analýzou zjistit seznam všech míst aplikace odkud bude daná funkce volána.

Z tohoto důvodu byla navržena metoda chráněného *stínového zásobníku* (*shadow call stack*) rozšiřující výše zmíněné 3 nové instrukce. Metoda spočívá v konceptu dvou zásobníků. První plní roli *alokačního zásobníku*, na který se ukládají lokální proměnné funkce, její parametry apod. Druhý, tzv. *stínový zásobník*, je aplikačnímu kódu nepřístupný a přistupují do něj pouze instrukce generované během instrumentace aplikace. Ty zde ukládají návratové adresy procedur a další citlivé informace a při návratu z procedury je návratová adresa brána pouze z tohoto zásobníku.

Technické řešení stínového zásobníku představili její autoři na architektuře x86 za pomocí lokálního úložiště dat vlákna, na které odkazuje segmentový registr `fs`. Vzhledem k tomu, že běžně se ve strojovém kódu využívají pouze registry `cs` (*code segment*), `ds` (*data segment*), `ss` (*stack segment*) a případně také `es` (*extra segment*), je možné předpokládat nebo ověřit statickou analýzou, že registr `fs` není používán. V rámci lokálního úložiště je tedy uložen ukazatel na stínový zásobník.

Paměťoví strážci

Druhým stavebním kamenem XFI jsou tzv. *paměťoví strážci* (*memory guards*). Jedná se o běhové kontroly vložené před každý potencionálně nebezpečný paměťový přístup či nepřímý skok. Pro každou paměťovou oblast, do které smí aplikace přistupovat, se vede dolní a horní mez adres a tyto informace o oblastech udržuje v tabulkách práv. Vložená běhová kontrola se skládá ze dvou částí:

- *fastpath memory guard* – nepoužívá tabulky práv, nýbrž přímo kontroluje cílovou adresu s pevně stanovenou horní a dolní mezí určité paměťové oblasti,
- *slowpath memory guard* – provádí sekvenční průchod tabulkou práv a kontroluje, zda-li cílová adresa náleží do některé z přístupných oblastí.

Pseudokód využívající instrukce x86 architektury je následující:

```
    # EAX contains address where to write
    if EAX < LowerBound, goto S
    if EAX > UpperBound, goto S

M:  Mem[EAX] := 123        # memory write allowed
    ...

S:  push EAX               # parameter for slowpath guard
    call SlowPathGuard
    jmp M                  # write allowed, perform it and continue
```

Před samotný zápis uvozený návěštím `M` je vložena běhová kontrola *fast path guard*, která v případě úspěchu dovolí požadovaný zápis. V jiném případě se předává řízení na *slow path guard*, který ověří cílovou adresu v tabulkách práv a teprve poté rozhodne zda bude zápis uskutečněn (skok `jmp M`) nebo se jedná o nepovolený zápis a aplikace je násilně ukončena přímo v proceduře *slow path guard*.

Důvod, proč byl zaveden *fast path guard*, je urychlení kontrol u přístupů do určité paměťové oblasti. Autoři totiž předpokládájí, že při načítání pluginu do hostitelské aplikace poskytne hostitel pluginu paměťovou oblast, kde bude umístěna halda nebo bude probíhat jejich vzájemná výměna dat, a proto zde plugin bude provádět četné paměťové přístupy. Doplnění správných mezí do *fast path* kontrol je proto provedeno až při načítání kódu pluginu do paměti pomocí relokací, protože v době překladu ještě údaj o adrese této paměťové oblasti není k dispozici.

Optimalizace

Technika XFI klade velký důraz na statickou analýzu binárního kódu, díky níž může provádět řadu optimalizací a odstranit množství nepotřebných či duplicitních kontrol. Základem těchto optimalizací je tzv. *verifikační stav* skládající se z informace o aktuálním stavu registrů procesoru, stavu zásobníku a rozsazích adres, na které lze provádět oprávněný přístup adresací registrů či proměnných, které jsou součástí verifikačního stavu. Verifikační stav je měněn při zpracovávání jednotlivých instrukcí a základních bloků.

Důvěryhodnost

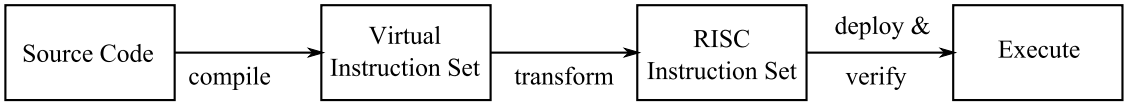
Před samotným spuštěním pluginu či obecně kódu instrumentovaného technikou XFI se provádí ověření důvěryhodnosti tohoto kódu. Za tímto účelem vytvořili autoři nástroj pro ověření na přítomnost běhových kontrol nazvaný *binary verifier*. Jeho jedinou funkcí je průchod binárním kódem a verifikace, zda nebyly po provedení instrumentace dělány (ruční) zásahy do tohoto kódu, které by mohly způsobit narušení integrity toku řízení či přístup do nepovolené paměťové oblasti.

3.2.5 Izolace modulů v uzlech senzorických sítí

Zajímavou alternativou k systémům založeným na SFI nebo XFI je metoda popsaná v [39]. Jedná se opět o implementaci ochrany na úrovni modulů (tj. aplikací nebo knihoven) v senzorických sítích, která však nepoužívá binární instrumentaci, nýbrž sofistikovaných transformací binárního kódu z virtuální instrukční sady do cílové instrukční sady. Princip této metody je popsán obrázkem 3.7.

Překlad

V první fázi jsou zdrojové soubory modulů přeloženy do virtuální instrukční sady, která je blízká obecné RISCové architektuře. Během překladu jsou do binárního kódu vloženy speciální instrukce zajišťující kontrolu paměťových přístupů a bezpečné předávání řízení mezi jednotlivými moduly.



Obrázek 3.7: Princip implementace softwarové izolace modulů v senzorických sítích.

To je obvykle uskutečněno přes rukojeti (*handles*), které slouží jako identifikátory paměťových bloků pevné velikosti. Každý přístup do paměti lze tak uskutečnit pouze prostřednictvím příslušné rukojeti.

Transformace

Binární kód virtuální instrukční sady je poté transformován na instrukce cílového RISCového procesoru. Vzhledem k tomu, že jak zdrojová tak i cílová instrukční sada patří do rodiny RISC, tak při převodu je většinou jedna virtuální instrukce vyjádřena jedinou instrukcí cílového procesoru. Problematickou částí převodu je alokace registrů, protože virtuální instrukční sada předpokládá nekonečnou množinu registrů, zatímco reálná architektura jich má pouze malý počet. Součástí alokace je také rozdělení registrů na 3 základní skupiny: registry nesoucí ukazatele, registry obecného použití (*general purpose registers*) a registry nesoucí rukojeti.

Verifikace a spuštění

Podobně jako u techniky XFI i zde je binární kód modulu podroben statické analýze před jeho spuštěním. Ta rozhodne, zda kód splňuje všechny potřebné požadavky a odhalí jakýkoli zásah do něj provedený po transformační fázi. Pro spuštění modulu je podobně jako u XFI potřeba jednoduché běhové prostředí poskytující funkce pro práci s rukojeťmi a např. také zásobníkem, který může být rozdělen do více nezávislých paměťových oblastí.

3.3 Kombinované prostředky

Pro úplnost je nutné zmínit také techniky, u kterých se autoři rozhodli zkombinovat úpravy hardware s prostředky softwarové izolace. Důvodem kombinace obou přístupů může být fakt, že každá metoda zvlášť není natolik efektivní jak by bylo požadováno. Například implementace paměťových přístupů hardwarovými kontrolami obvykle znamená, že každý paměťový přístup aplikace je zkontrolován, což v případě sekvence zápisů do stejné paměťové oblasti nemusí být nutné, pokud paměťová oblast patří dané aplikaci. To je však možné ověřit v případě použití softwarových prostředků, jmenovitě např. statickou analýzou binárního kódu, a nevkládat před tyto paměťové přístupy žádné běhové kontroly. A naopak v případě čistě softwarového přístupu jsou běhové kontroly důvodem celkového zpomalení aplikace i nárůstu jejího kódu, což lze vyřešit vytvořením nových instrukcí procesoru nahrazujících tyto běhové kontroly.

3.3.1 Harbor a XFI

Příkladem jsou již zmíněné techniky Harbor [20], která zakládá na SFI technice a poskytuje paměťovou ochranu na RISCových procesorech v senzorických sítích, a XFI [6] zaměřená na izolaci pluginů od hostitelské aplikace resp. modulů jádra operačního systému od samotného jádra. Obě byly původně navrženy jako ryze softwarová řešení, ale jak jejich autoři ukázali, vhodnou podporou v hardware je možné snížit běhovou režii až o jeden řád (viz sekce 3.4). Principem této podpory je implementace nových instrukcí procesoru, které byly původně vykonávány skupinou více jiných

instrukcí, a rovněž hardwarová implementace prohledávání tabulek s právy paměťových regionů, což značně urychluje kontroly paměťových přístupů.

3.3.2 Izolace na úrovni funkcí

Kombinaci úpravy hardware a modifikaci překladače využívá technika popsaná v článku [21], která spadá již spíše do oblasti bezpečnosti, avšak lze ji velmi snadno použít i pro izolaci procesů či jiných entit běhu v rámci operačních systémů. Její motivací je izolace pluginů od hostitelské aplikace bez vědomí operačního systému. Granularita izolace přitom dosahuje úrovně jednotlivých funkcí pluginu.

Základním prvkem ochrany je opět block paměti začínající na libovolné adrese a libovolné délky. Provede-li aplikace alokaci bloku paměti, upravený překladač vloží za tuto alokaci speciální instrukci pro registraci tohoto bloku do tabulky práv spravované speciální hardwarovou jednotkou, která kontroluje veškeré paměťové přístupy mezi CPU a vyrovnávací pamětí, která je předřazena před samotnou fyzickou paměť. Unikátnost této techniky tkví v tom, že přístup do alokovaného bloku paměti nemá automaticky celá aplikace, ale pouze ta funkce, která alokaci provedla. Právo pro přístup také dědí všechny další volané funkce do doby, než se provede návrat z původní funkce, kdy je záznam o alokovaném bloku z tabulky práv automaticky odstraněn hardwarovou jednotkou. Ta také sleduje instrukce volání a návratu z funkcí a nedovolí tak pluginu skočit na nedovolený kód například v důsledku přepisu zásobníku.

3.4 Srovnání

Tabulka 3.1 podává přehled všech uvedených technik spolu s informacemi o jejich efektivitě. Zdrojem údajů jsou odborné články, na které bylo odkazováno v předchozích sekcích textu.

Budeme-li chtít provést porovnání či vyhodnocení uvedených technik, tak pravděpodobně nejdůležitějším kritériem bude úroveň poskytované paměťové ochrany. Pro jednoduchost byly zavedeny pouze dvě skupiny úrovní: částečná a úplná. Úplná úroveň ochrany je dosažena tam, kde po správném nasazení techniky nemůže aplikace za žádných okolností (snad kromě chyby v hardware) přistupovat do paměti jiných aplikací či jádra systému, pokud jí to nemá být výslovně umožněno. Jedná se tedy o uspokojivou náhradu za funkcionalitu MMU jednotky z pohledu paměťové ochrany. V případě částečné úrovně ochrany tato podmínka není splněna vždy, v důsledku čehož aplikace může přepsat např. návratovou adresu z aktuální procedury a spustit poté zcela jiný kód, než který byl předmětem překladu. Většina uvedených technik spadá do skupiny úplné paměťové ochrany, výjimkou jsou softwarové techniky používající source-to-source překlad nebo spoléhající na ruční úpravu zdrojového kódu. Důvodem je fakt, že uvedené techniky nedokáží ve všech situacích odhalit nedovolený přepis zásobníku nebo nijak neošetřují korektnost návratových adres funkcí či ukazatelů na funkce. Například výstup CCured překladače je považován za paměťově bezpečný, ale i přesto existuje způsob jak obejít jeho běhové kontroly a zapsat hodnotu za hranice proměnné typu pole [42] a přepsat tak návratovou adresu procedury.

Dalším kritériem hodnocení je režie měřená jako zpomalení běhu aplikace po nasazení dané techniky vzhledem k běhu aplikace bez tohoto nasazení. Hodnoty uvedené v tabulce reprezentují rozpětí od nejmenšího zaznamenaného zpomalení po největší. Předmětem měření byly aplikace náročné na datové výpočty jako Fourierova transformace, pulsní kódová modulace, komprese apod. Nejlepší výsledky podle očekávání mají hardwarové techniky, případně jejich kombinace se softwarovými metodami. Softwarové metody jsou z velké většiny založeny na běhových kontrolách prováděných při přístupu (zápisu) do paměti, a proto obvykle nemají stejnou efektivitu jako ryze hardwarová řešení. Extrémním případem je metoda softwarové MMU, která poskytuje aplikaci iluzi virtuální paměti implementované stránkováním a každý paměťový přístup je předmětem softwarového překladu adresy, což způsobuje desetinásobné nebo až stonásobné zpomalení běhu takové aplikace.

Naproti tomu pozoruhodnou efektivitu má technika SFI přidávající ve všech testovaných případech režii nejvýše 20 %. Základním důvodem je malý počet dodatečných strojových instrukcí vkládaných před každý paměťový zápis. Cenou za tento výsledek je řada nevýhod spojených se správou dostupné paměti aplikace, velká interní fragmentace paměti a další.

Z tabulky lze také vyčíst, že u řady softwarových technik je nejmenší zpomalení aplikace pouze v jednotkách procent (XFI, SFI, MiSFIT) či dokonce nulové (CCured, Cyclone). To je důsledek optimalizací získaných na základě statické analýzy zdrojového nebo binárního kódu, díky kterým nedošlo ke vložení běhových kontrol tam, kde by to bylo zbytečné nebo došlo k optimalizaci běhové kontroly. Typickým příkladem je přístup ke globálním proměnným, kde je běhová kontrola zbytečná, nebo předsunutí kontroly paměťového zápisu před smyčku, jejíž řídící proměnná vymezuje dolní a horní hranici zapisované paměti. Pokud v testovaných aplikacích převažovaly optimalizace tohoto druhu, pak bylo výsledkem jen zmíněné několikaprocentní zpomalení.

Režie techniky na běh výsledné aplikace nemusí být vždy hlavním kritériem. Na vestavných systémech sensorických sítí může být prioritou velikost výsledné aplikace kvůli omezenému prostoru flash pamětí, ale již tolik nemusí vadit zpomalení aplikace. Z tohoto pohledu jsou opět ideálním řešením hardwarové techniky či jejich kombinace se softwarovým řešením, díky kterým převažují zásahy v modelu procesoru, což minimalizuje počet běhových kontrol nutných vkládat do aplikace. Na druhou stranu u desktopových systémů dnes velikost kódu aplikací není příliš sledovaný ukazatel, takže i dvojnásobný nárůst velikosti může být přijatelný.

Jak je vidět z uvedených kritérií, vybrat ideální řešení paměťové ochrany či izolace není v obecném případě pravděpodobně možné. Kromě uvedeného srovnání úrovně ochrany, režie na běh aplikace a velikosti kódu, je nutné také uvážit problémy spojené s každou z technik uvedené v posledním sloupci tabulky 3.1. Hardwarové techniky předpokládají úpravy modelu procesoru přidáním nových registrů, instrukcí či jiných prvků, což je citlivý zásah do jeho fungování a samozřejmě vyžaduje dodatečné náklady na výrobu. Naproti tomu softwarová řešení mohou být použitelná i ve stávajících systémech bez dodatečných úprav, avšak i mezi nimi jsou velké rozdíly co se použitelnosti týče. Například source-to-source překladače potřebují pro svou činnost zdrojový kód a jiné techniky vyžadují přímo úpravu kódu uživatelem. Z tohoto pohledu se jeví techniky instrumentace binárního kódu jako velmi dobré řešení, jelikož pracují přímo se strojovým kódem aplikace. I zde však bývá požadavek na dodatečná metadata v binárním kódu, bez kterých taková instrumentace nemůže být provedena. Například technika XFI pracující nad binárním kódem architektury x86 používá ke své práci systém *Vulcan* [30] a ten požaduje u zpracovávaného binárního kódu úplnou ladící informaci.

Technika	Zaměření	Prostředek implementace	Paměťová ochrana	Nárůst velikosti kódu aplikace	Zpomalení aplikace	Nevýhody
MPU	vše	úprava hardware	úplná	0 %	jednotky %	omezený počet chráněných regionů
MMP	vše	úprava hardware	úplná	0 %	méně než 1 %	údržba tabulky práv srovatelné se stránkovacími tabulkami MMU
CCured	vše	source-to-source překlad	částečná	?	0 – 140 %	pouze jazyk C, potřeba zdrojových souborů a asistence uživatele
Cyclone	vše	úprava zdrojového kódu	částečná	do 5 %	0 – 150 %	pouze jazyk C, potřeba zdrojových souborů
Softwarová MMU	vše	úprava zdrojového kódu, instrumentace	úplná	?	tisíce %	velké zpomalení, implementace stránkovacích tabulek v software
SFI	plugovatelné aplikace	instrumentace	úplná	?	10 – 20 %	omezená velikost segmentů, interní fragmentace, předpoklad pevné délky instrukcí, rezervace 3 registrů, potřeba zdrojových souborů
Harbor	moduly vestavných systémů	instrumentace	úplná	30 – 60%	400 % a více	velké zpomalení, zákaz nepřímých skoků
MiSFIT	moduly jádra OS	instrumentace	úplná	?	10 – 30 %	omezená velikost segmentů, interní fragmentace
XFI	plugovatelné aplikace, moduly jádra OS	instrumentace	úplná	30 – 150 %	5 – 120 %	potřeba metadat v binárním kódu
XFI s hardwarovou podporou	plugovatelné aplikace, moduly jádra OS	úprava hardware, instrumentace	úplná	jednotky %	jednotky %	potřeba metadat v binárním kódu
Harbor s hardwarovou podporou	moduly vestavných systémů	úprava hardware, instrumentace	úplná	jednotky %	2 - 60 %	zákaz nepřímých skoků
Izolace na úrovni funkcí	plugovatelné aplikace	úprava hardware, úprava překladače	úplná	?	jednotky %	potřeba zdrojových souborů

Tabulka 3.1: Srovnání různých prostředků paměťové ochrany. V některých odborných článcích nebyly k dispozici údaje o nárustu velikosti kódu aplikace při použití dané techniky, v takovém případě je v tabulce uveden pouze otazník.

Kapitola 4

Operační systém HelenOS

HelenOS je moderní operační systém mikrojádrového typu podporující v současné době 7 různých procesorových architektur. Kritické funkce jako správa paměti, plánování procesů či meziprocesová komunikace (IPC) jsou implementovány v relativně malém mikrojádře a další služby systému jako ovladače zařízení, síťový subsystém, souborové systémy jsou poskytovány uživatelskými procesy označovanými jako servery.

Tato kapitola se zabývá stručným popisem pouze těch funkcí jádra či serverů systému, které jsou spojeny s tématem této diplomové práce. Detailnější informace lze najít v dokumentu [11][1] nebo přímo na stránkách projektu [12] v sekci dokumentace.

4.1 Správa paměti

4.1.1 Fyzická paměť

Fyzická paměť je v HelenOS reprezentována souvislými oblastmi nebo-li *zónami*. Na většině architektur je vytvořena pouze jediná zóna reprezentující celou dostupnou fyzickou paměť. Zóny jsou dále rozděleny na pevně velké *rámce* (typicky o velikosti několika kilobytů), které jsou nejmenší jednotkou alokace. Tu zajišťuje alokátor rámců a používá k tomu všeobecně známý buddy systém.

4.1.2 Virtuální paměť

Díky přítomnosti MMU jednotky na každé podporované procesorové architektuře využívá HelenOS techniku virtuální paměti a každému procesu nabízí iluzi samostatného adresového prostoru. Překlad virtuálních adres na fyzické je zajištěn metodou stránkování a v závislosti na požadavcích konkrétního procesoru lze využít buď obecný mechanismus 4úrovňových stránkovacích tabulek, přičemž prostřední dvě úrovně mohou být vypuštěny, nebo mechanismus inverzních stránkovacích tabulek.

Každý adresový prostor procesu je z hlediska snadné správy rozdělen na souvislé oblasti (*address space areas*). Mapování virtuálních adres na fyzické se provádí pouze na těchto oblastech a procesu není dovolen přístup mimo ně. Obvykle obsahuje každá oblast data pouze jednoho typu – např. spustitelný kód, data nebo zásobník a haldu, což také umožňuje jednotné nastavení přístupových práv (čtení, zápis, spuštění, řízení vyrovnávacích pamětí). Tato práva se pak při vytváření mapování stránek na fyzické rámce přímo vepisují do odpovídajících položek stránkovacích tabulek.

V sekci 2.2 byla popsána typická využití vlastností MMU jednotky v běžných operačních systémech. Nyní bude popsána i situace v HelenOS:

- **Mapování stránek** do fyzické paměti se provádí až v případě přístupu procesu na příslušnou stránku, tj. na požádání (*on demand*). Zda a jak bude stránka namapována na fyzický

[1] Tato dokumentace se vztahuje pouze k verzi 0.2.0, avšak většina popsaných principů fungování jádra je stále aktuální i pro současnou verzi 0.4.2.

rámec je v pravomoci *paměťových backendů* pokrývajících každou paměťovou oblast. HelenOS v současnosti rozlišuje 3 typy backendů: anonymní paměti, obrazu ve formátu ELF a fyzické paměti. Anonymní backend a backend fyzické paměti mapují stránky na prázdné rámce (např. pro zásobník či haldu procesu), resp. rámce pro komunikaci s různými zařízeními přes fyzickou paměť. ELF backend pak zajišťuje mapování stránek na fyzické rámce s částmi obrazu formátu ELF, kde se nachází kód i data procesu, přičemž využívá také techniky *copy on write*.

- **Ochrana a izolace** procesů je zajištěna přirozeně tím, že každý proces je umístěn v samostatném adresovém prostoru. Interakce s ostatními procesy je možná přes sdílenou paměť, která je však pod kontrolou operačního systému.
- **Externí fragmentaci** je zcela zamezeno, protože souvislé oblasti stránek mohou být mapovány na fyzické rámce netvořící souvislý úsek fyzické paměti.
- **Sdílená paměť** je podporována prostřednictvím sdílení celých oblastí adresového prostoru. Každý z procesů může mít nastavena ke sdílené oblasti různá přístupová práva, tj. jeden proces může číst i zapisovat a druhý např. pouze číst.
- **Detekce nulového ukazatele** je rovněž podporována, protože žádná z paměťových oblastí nesmí zabírat první stránku adresového prostoru, ve které se nachází adresa 0.

Mezi vlastnosti, které čekají na svou implementaci je podpora odkládání stránek do sekundární paměti a souvisejících algoritmů výměny stránek. Pokud se zcela vyčerpá kapacita fyzické paměti, alokátor rámců se při požadavku na alokaci nového rámce může např. zablokovat až do chvíle, než se některý rámec uvolní. Dále chybí implementace podpory sdíleného kódu a knihoven[2], detekce přetečení zásobníku nebo plná podpora mechanismu *copy on write*, která se v současnosti používá pouze u procesů zaváděných při startu systému.

4.2 Procesy a vlákna

Základní jednotkou plánování a přidělování procesoru je vlákno. Vlákna jsou plánována preemptivně algoritmem round robin s víceúrovňovými frontami. Každé vlákno patří jednomu procesu, přičemž proces může zahrnovat libovolný počet vláken. Základní jednotkou běhu z pohledu procesů v uživatelském prostoru však není vlákno, nýbrž pseudo vlákno nebo-li vlákénko (*fibril*). Jejich podpora je přímo integrální součástí knihovny `libc` a namísto preempce se tato knihovna spoléhá na kooperativní plánování. Tedy vlákénko předá řízení jinému jen na bázi dobrovolnosti nebo v případě, kdy se zablokuje na některém ze synchronizačních primitiv vlákének.

4.2.1 Meziprocesová komunikace

Velmi důležitou roli hraje v HelenOS meziprocesová komunikace (*IPC – Inter-Process Comunication*) založená na asynchronním zasílání zpráv. Každý proces je vybaven několika telefony (*phones*) a frontou příchozích zpráv (*answerbox*). Telefon používá k odeslání zprávy jinému procesu a frontu příchozích zpráv pravidelně kontroluje na nově příchozí zprávy od jiných procesů. Pro zjednodušení použití meziprocesové komunikace je součástí knihovny `libc` tzv. *asynchronní framework* poskytující velmi příjemné rozhraní pro programátora, který je odstíněn od technických detailů.

IPC je v HelenOS využíváno hlavně pro komunikaci mezi běžnými procesy a servery, které nabízejí některou ze systémových služeb (přístup do souborového systému, ovladač klávesnice atd.). Každý ze serverů se při svém startu zaregistruje u tzv. *jmenné služby* (*naming service*), což je speciální server, a bude-li některý z procesů chtít komunikovat s některým konkrétním serverem, obrátí se na jmennou službu, která mu spojení zprostředkuje.

[2]Problematika dynamicky linkovaných sdílených knihoven byla řešena v jiné diplomové práci[33], avšak její výsledky ještě nejsou součástí oficiální verze 0.4.2.

4.2.2 Spouštění procesů

HelenOS umožňuje spouštění procesů, jejichž obraz je uložen ve formátu ELF. Proces může být spuštěn buď při startu systému nebo na žádost jiného procesu (např. procesu `init` po startu systému). Přestože výsledek obou těchto způsobů je stejný, technika načtení obrazu procesu a spuštění je značně odlišná.

V prvním případě je obraz procesu umístěn v modulu, který je načten do paměti bootloaderem při startu systému. Po inicializaci jádra je proces spuštěn a přitom obraz namapován pomocí ELF backendu do jeho virtuálního adresového prostoru.

V druhém případě je situace složitější. Rodičovský proces musí nejprve požádat jmennou službu o spuštění nové instance speciálního procesu nazvaného *program loader* a poté s ním navázat IPC spojení. V následné komunikaci mu sdělí identifikační údaje nového procesu a také cestu k souboru s ELF obrazem. Loader načte obraz do svého adresového prostoru a pak provede skok na jeho vstupní bod, čímž dojde ke spuštění požadovaného procesu.

Kapitola 5

Návrh a implementace rozšíření HelenOS

Předchozí kapitoly podaly přehled o využití funkcí MMU jednotky v operačních systémech a o několika způsobech, jak lze alespoň zčásti tyto funkce nahradit v případě její nepřítomnosti. Účelem této kapitoly bude zúročení těchto informací a navržení způsobu a implementace, jak rozšířit nebo přizpůsobit operační systém HelenOS tak, aby byl schopen běhu i bez MMU, ale přitom stále poskytoval určitou úroveň paměťové ochrany a izolace procesů.

5.1 Diskuze

5.1.1 Kritéria návrhu a implementace

Pro vypracování návrhu i samotné implementace bylo stanoveno několik kritérií, z nichž velká část je také dodržována samotnými autory při vývoji částí HelenOS. Tato diplomová práce se snaží být v souladu s těmito kritérii.

Zachování funkcí MMU Základním kritériem samozřejmě byla snaha zachovat co nejvíce funkcí MMU jednotky využívaných jádrem systému a poskytovaných MMU jednotkou. V ideálním případě by byly zachovány všechny funkce a jádro systému by si tak mohlo zachovat stávající funkčnost, avšak ze srovnání v sekci 3.4 plyne, že ukázkové řešení snažící se o úplnou emulaci MMU jednotky je velmi neefektivní.

Transparentnost Uživatelské procesy i aplikační programátoři by měli být zcela odstíněni od problematiky přítomnosti MMU a s ní souvisejícího rozvržení paměti v rámci procesu. Pro jejich práci by stále měl být platný předpoklad o existenci jediného adresového prostoru, který má proces zcela pro sebe.

Přenositelnost HelenOS klade také velký důraz na přenositelnost, o čemž svědčí i počet podporovaných procesorových architektur. Podpora režimu bez MMU jednotky se proto snaží následovat tento koncept, a zaměřuje se hlavně na úpravy procesorově nezávislých částí jádra systému.

Výhodou tohoto přístupu je také možnost vypnout funkcionalitu MMU jednotky i na procesorech, které ji nativně nabízejí. Například procesory z rodiny ARM disponují velkou nevýhodou při používání virtuální paměti, která spočívá v nutnosti kompletního smazání obsahu TLB paměti při přepínání kontextu mezi procesy nesdílejícími společný adresový prostor. Pokud by se MMU jednotka zcela vypnula, je pravděpodobné, že dojde nejen ke zrychlení běhu procesů, ale také úspoře energie spotřebované procesorem.

Efektivita Navržené řešení by samozřejmě mělo být efektivní z hlediska režie na běh systému, uživatelských procesů i paměťové spotřeby.

Minimální změny zdrojových kódů Přestože v jádře HelenOSu jsou implementovány pouze kritické funkce, jeho zdrojové soubory mají v součtu desítky tisíc řádek. Každá změna či nová funkčnost v těchto souborech přináší riziko zanesení chyby do fungujícího a stabilního jádra, a proto se tato práce zaměřila také na minimalizaci jeho změn.

5.1.2 Volba techniky paměťové ochrany

Jak již bylo zmíněno v předchozích kapitolách, jednou z nejcennějších funkcí MMU jednotky je ochrana a vzájemná izolace procesů, která je typicky docílena jejich oddělením do samostatných adresových prostorů. Splněním těchto cílů bez podpory MMU se zabývala řada vědeckých publikacích, jejichž výčet a srovnání bylo podáno v kapitole 3. Pro účely této diplomové práce byla z řady popsaných technik nakonec vybrána XFI (viz sekce 3.2.4), a to z několika důvodů.

V prvé řadě se jedná o zcela transparentní techniku jak pro programátory, tak pro samotné aplikace, protože paměťová ochrana a izolace se zajišťuje pouze instrumentací binárního kódu. Díky tomu je nezávislá na použitém programovacím jazyku, což může hrát roli při dalším rozvoji systému HelenOS. V současnosti je tento systém napsán v jazyce ANSI C s použitím některých rozšíření překladače *gcc* nebo přímo v assembleru v případě menších, procesorově závislých částí jádra. Před nedávnem však autoři uvažovali o přepisu některých částí do jiného jazyka – konkrétně do jazyka Objective C. Pokud by použitá technika paměťové ochrany byla přímo spjatá s konkrétním jazykem, mohlo by to do jisté míry omezit budoucí použití výsledků této diplomové práce při adaptaci na další procesory bez MMU jednotky.

Další výhodou je poměrně příznivá režie na běh instrumentovaných aplikací nepřesahující 120 %, což je srovnatelné s ostatními technikami softwarové izolace. Lepší režii nabízí jen SFI a to 20 %. Tento slibný výsledek je však vyvážen řadou omezení. Zásadní omezení představuje předpoklad pevné velikosti všech segmentů (v terminologii správy paměti HelenOS se jedná o paměťové oblasti), což nevadí u oddělených virtuálních adresových prostorů, avšak v případě jediného je to velký problém. Nevýhodou je také předpoklad procesorové architektury RISC s jednotnou délkou binárního kódu instrukcí.

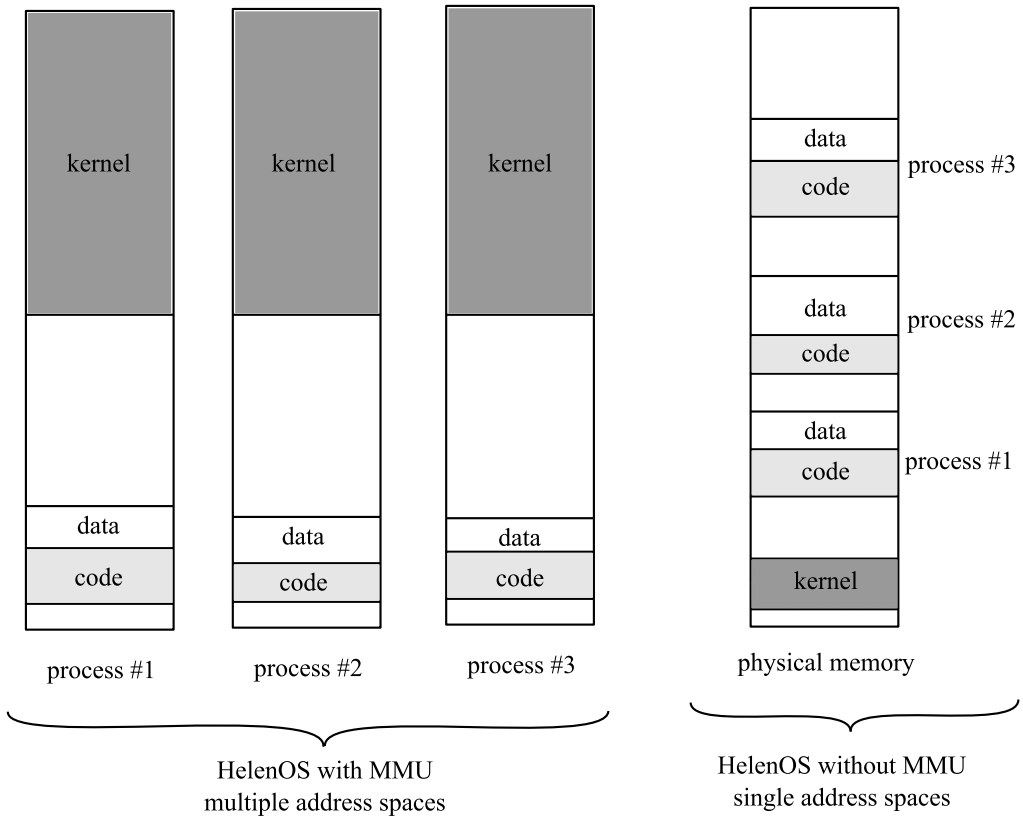
Velmi efektivní řešení skýtají hardwarové techniky paměťové ochrany, které však rovněž potřebují jistou podporu v software. Z tohoto hlediska je využití XFI techniky také velmi výhodné, protože stávající infrastrukturu podpory XFI v jádře navrženou dále v této diplomové práci je možné použít pro hardwarovou verzi XFI techniky představenou v článku [6]. Ta přesouvá výpočetně náročnou část běhových kontrol přímo do dedikovaných instrukcí procesoru a v důsledku toho dochází ke snížení režie běhu aplikace i velikosti kódu na jednotky procent, což je zanedbatelná režie porovnatelná s ostatními hardwarovými technikami.

Na druhé straně hlavní nedostatek XFI z hlediska přenositelnosti spočívá právě v technice binární instrumentace, při které je nutné pracovat s instrukční sadou konkrétní procesorové architektury.

5.1.3 Návrh rozšíření

Výsledný návrh rozšíření HelenOSu je rozdělen do dvou logických částí. První část se zaobírá problematikou chybějící virtuální paměti s libovolným počtem adresových prostorů, jejichž existenci jádro systému předpokládá pro svou činnost resp. pro běh uživatelských procesů. Cílem této části návrhu je tedy umožnění běhu systému bez MMU jednotky. Druhá část pak řeší problémy spojené s chybějící izolací a paměťovou ochranou procesů a aplikuje techniku XFI na uživatelské procesy.

Princip rozšíření je schématicky popsán obrázkem 5.1. V levé části je zobrazena současná situace v systému HelenOS, kdy každý proces má svůj vlastní adresový prostor, přičemž v každém z nich je také sdílená oblast horní paměti vyhrazená jádru systému. Bez MMU jednotky je samozřejmě



Obrázek 5.1: Návrh rozšíření systému HelenOS bez MMU jednotky.

k dispozici pouze jediný adresový prostor reprezentovaný přímo fyzickou pamětí, do kterého je nutné procesy umístit a umožnit jejich koexistenci, což je předmětem pravé části schématu. Zajištění vzájemné izolace procesů zajišťuje nasazení XFI techniky, přičemž toto nasazení je volitelné. To může být výhodné zejména v situaci, kdy potřebujeme získat maximální výkon procesu a máme záruky o jeho důvěryhodnosti. Typicky se může jednat o některé serverové procesy (např. jmenná služba) s velmi jednoduchou funkcionalitou a potřebou rychlé zpětné vazby směrem ke klientským procesům.

S nasazením XFI podpory přímo do jádra systému prozatím tento návrh nepočítá, i když u velké části jádra by to možné bylo. Vycházíme však z předpokladu, že jádro HelenOS je odladěné a stabilní a největší nebezpečí představují uživatelské procesy.

5.1.4 Procesorová architektura implementace

HelenOS podporuje v současné verzi 0.4.2 celkem 7 různých architektur, z nichž každá je vybavena MMU jednotkou. Nejblíže ke světu vestavných systémů má procesor ARM, u něhož existují i varianty bez MMU jednotky, a proto by se jednalo o ideálního kandidáta na prototypovou implementaci této diplomové práce. Z časových důvodů však musela být zvolena architektura Intel IA-32, pro kterou byly lepší podmínky k implementaci binární instrumentace než u procesoru ARM. Další výhodou IA-32 je její rozšířenost, což umožňovalo vyzkoušet prototyp i na reálném hardware.

5.2 Režim bez MMU

V této části budou popsány změny v jádře systému a uživatelském prostoru nutné pro adaptaci HelenOS na běh bez MMU jednotky. Většina těchto změn byla provedena ve zdrojových kódech

nezávislých na konkrétní procesorové architektuře. Pokud se jednalo o změny závislé na zvoleném procesoru, bude to explicitně uvedeno. Většina změn je ve zdrojovém kódu vložena prostřednictvím direktiv podmíněného překladu `#if` a `#endif` s podmínkou existence symbolů začínajících `CONFIG_NOMMU`. Díky tomu není nutné vytvářet novou, samostatnou vývojovou větev jádra pouze pro účely jeho běhu na procesorech bez MMU, protože v případě, kdy nebudou při překladu symboly s prefixem `CONFIG_NOMMU` definovány, jádro bude přeloženo bez uvedených úprav.

5.2.1 Umístění jádra ve fyzické paměti

Bez podpory virtuální paměti je k dispozici pouze jeden adresový prostor reprezentovaný přímo fyzickou pamětí. Z pohledu činnosti jádra systému se nejedná o žádné omezení, protože jeho kód a data se i při podpoře virtuální paměti nachází v jediném adresovém prostoru. Rozdíl je pouze v jeho umístění a mapování do fyzické paměti. Při podpoře virtuální paměti je jádro umístěno v horní části prostoru a ve stránkovacích tabulkách je pevně zadáno mapování této oblasti na začátek fyzické paměti. Bez podpory virtuální paměti musí každá adresa používaná v jádře přímo odpovídat fyzické adrese, což se docílí změnou makra `KA2PA()` převádějící adresu jádra na fyzickou adresu a makra `PA2KA()` převádějící fyzickou adresu na adresu jádra. Obě makra, definovaná v souboru `kernel/arch/<name>/include/page.h`[1], nyní vrací identickou hodnotu:

```
#define KA2PA(x)  (x)
#define PA2KA(x)  (x)
```

Tato změna rovněž zajistí správný překlad jádra do ELF obrazu a výsledného binárního obrazu, protože linker skript používá tato makra pro definici začátku adres výsledných ELF sekcí.

Kvůli logické konzistentnosti zdrojových souborů jsou také potlačeny definice symbolů

- `KERNEL_ADDRESS_SPACE_START_ARCH`,
- `KERNEL_ADDRESS_SPACE_END_ARCH`,
- `USER_ADDRESS_SPACE_START_ARCH`,
- `USER_ADDRESS_SPACE_END_ARCH`

ze souboru `kernel/arch/<name>/include/mm/as.h`. Ty udávají rozdělení virtuálního prostoru na část uživatelskou a část jádra. Bez MMU jednotky se však tato makra vůbec nepoužívají, takže ponechání jejich definice by mohlo být matoucí.

5.2.2 Správa adresových prostorů

I přesto, že bez MMU jednotky je k dispozici jen jediný adresový prostor, bylo rozhodnuto o zachování iluze samostatného adresového prostoru pro každý proces. Důvody pro toto rozhodnutí jsou zcela pragmatické. Každý adresový prostor je popsán strukturou `as_t` definovanou v `kernel/generic/include/mm/as.h`, která zahrnuje definici paměťových oblastí (struktura `as_area_t`). Ty jsou pak předmětem alokace fyzické paměti. Pokud by tato hierarchie měla zaniknout, bylo by nutné vytvořit hierarchii novou, kde by se opět musely uchovávat informace o přiřazení paměťových oblastí procesům. To by také vyžadovalo přepis mnoha funkcí týkající se správy adresových prostorů v souboru `kernel/generic/src/mm/as.c`, který má už nyní přes dva tisíce řádků.

Zachováním stávajícího konceptu adresových prostorů nebude nutné dělat žádné velké změny, přičemž každý (virtuální) adresový prostor bude samozřejmě reprezentován celou fyzickou pamětí. Zbývá pouze vyřešit jejich vzájemnou koexistenci ve sdíleném prostoru, čímž se zaobírá správce paměťových oblastí (*address space areas manager*) popsaný v následujícím textu.

[1] *name* reprezentuje zvolenou procesorovou architekturu, v této diplomové práci se jedná o ia32.

5.2.3 Správa paměťových oblastí

Paměťové oblasti jsou části adresového prostoru, pro jejichž správu nabízí HelenOS rozhraní definované funkcemi v souboru `kernel/generic/src/mm/as.c`.

Při vytváření nové oblasti pomocí *as_area_create()* volající přímo specifikuje její bázovou virtuální adresu a pokud by se nová oblast měla překrývat s jinou oblastí ve stejném adresovém prostoru, její vytvoření není dovoleno. Typická situace však je, že volající zná rozložení adresového prostoru a specifikuje bázovou virtuální adresu tak, aby k překryvu nedošlo. Běžný příklad je alokace haldy nebo načítání ELF obrazu do adresového prostoru procesu, kde každý ELF segment má vlastní paměťovou oblast.

V případě sdíleného adresového prostoru by byl takový postup velmi problematický, a proto bylo nutné změnit rozhraní funkce *as_area_create(as, flags, size, base, attrs, backend, backend_data)* tak, že bázová adresa (parametr *base*) je vstupní i výstupní argument současně. Volající tak dostane příležitost uvést preferovanou bázovou adresu oblasti a pokud není možné ji vytvořit, bude vybrána jiná adresa a vrácena volajícímu.

Problematické je rovněž sdílení oblastí mezi různými adresovými prostory implementované funkcí *as_area_share(src_as, src_base, size, dst_as, dst_base, dst_flags_mask)*. Protože každý z těchto prostorů je přímo představován fyzickou pamětí, tak sdílení je samozřejmě triviální. Je však potřeba zajistit, aby každá ze stran podílejících se na sdílení znala bázovou adresu paměťové oblasti. To odpovídá parametrům *src_base* a *dst_base* funkce *as_area_share()*, do které byla vložena kontrola na rovnost těchto parametrů. Nejsou-li bázové adresy rovny, sdílení není uskutečněno. Při zapnutém režimu s MMU se tyto bázové adresy zpravidla nerovnají, protože každá se vztahuje k jinému virtuálnímu adresovému prostoru.

Správce paměťových oblastí

V případě oddělených adresových prostorů byly informace o paměťových oblastech uloženy přímo ve struktuře adresového prostoru `as_t`. V režimu bez MMU zůstává tento koncept nadále platný, ale díky sdílenému adresovému prostoru je nutné mít centralizovanou informaci o všech alokovaných oblastech, což se stalo úkolem *správce paměťových oblastí* (*address spaces areas manager*). Další jeho činností je práce s alokátorem fyzickým rámců, tedy alokace a uvolňování paměťových oblastí. V režimu s podporou MMU totiž funkce pro správu paměťových oblastí jako *as_area_create()* a *as_area_resize()* pouze vytvářely informaci o vzniku mapování stránek do fyzické paměti. K samotné alokaci rámců docházelo až při přístupu na tyto stránky, k čemuž by bez MMU nikdy nedošlo, a proto je nutné provádět operace (de)alokace ihned.

Implementace správce paměťových oblastí je umístěna v souboru `kernel/generic/src/nommu/nommu.c` a zahrnuje tyto funkce:

- *nommu_area_create(base, size, flags, attributes)* – pokusí se vytvořit paměťovou oblast na zadané adrese *base*. Pokud není možné vytvořit ji na této adrese, provede se průchod datovými strukturami s alokovanými oblastmi a vybere se taková bázová adresa, na které alokace uspěje (pokud taková existuje). V tom případě je tato adresa vrácena v parametru *base*. Volání této funkce bylo vloženo do implementace *as_area_create()*, která je této funkci zcela podřízena.
- *nommu_area_destroy(base)* – zruší paměťovou oblast začínající na dané adrese. Volání této funkce bylo vloženo do *as_area_destroy()*, aby bylo možné sledovat, které paměťové oblasti byly zrušeny.
- *nommu_area_resize(base, size)* – změní velikost paměťové oblasti. Volání této funkce bylo vloženo do *as_area_resize()*, aby bylo možné sledovat změny velikostí paměťových oblastí.
- *nommu_block_reserve(as, addr, size)* – rezervuje část fyzické paměti pro alokaci paměťových oblastí ze zadaného adresového prostoru *as*.

- *nommu_block_free(as, addr)* – zruší rezervaci části fyzické paměti pro daný adresový prostor *as*.

První tři funkce *nommu_area_xxx()* pracují s alokátorem fyzických rámců a představují nadstavbu příslušných funkcí *as_area_xxx()*. Jako alokátor rámců byl ponechán původní buddy alokátor využívaný v režimu s MMU jednotkou. Jeho hlavní nevýhoda je možnost alokovat pouze bloky paměti, jejichž velikost je druhou mocninou čísla 2, což má bez MMU za následek značnou interní fragmentaci alokovaných bloků.

Funkce *nommu_block_reserve()* a *nommu_block_free()* nemají svůj ekvivalent v původních funkcích správy oblastí a přinášejí zcela novou funkcionalitu. Ta je motivována způsobem načítání obrazu procesu ve formátu ELF do paměti. Pro každý ELF segment je alokována samostatná oblast, přičemž jejich rozestupy jsou pevně stanoveny. Díky výše popsanému způsobu alokací však nelze předem zaručit, na jakých adresách budou oblasti alokovány a také v jakých vzájemných rozestupech. Z tohoto důvodu je možné uzamknout část fyzické paměti funkcí *nommu_block_reserve()* pouze pro daný proces (identifikovaný strukturou adresového prostoru `as_t`), který pak při volání funkce *as_area_create()* specifikuje preferované adresy začátků oblastí, kterým je vyhověno a všechny segmenty tak mohou být v patřičných vzdálenostech od sebe. Po dobu rezervace části fyzické paměti nemá dovoleno provádět v ní alokaci žádný jiný proces, čímž by se mohlo znemožnit správné načtení segmentů. Po dokončení celé operace se zavolá funkce *nommu_block_free()* a rezervovaná část paměti se opět zpřístupní i ostatním procesům.

5.2.4 Rozhraní pro ovladače zařízení

Mapování fyzické paměti do adresového prostoru procesu zajišťuje rozhraní pro ovladače zařízení (*DDI - Device Drivers Interface*). Implementace tohoto rozhraní je uvedena v jádře systému v souboru `kernel/generic/src/ddi/ddi.c`. Proces (server), který vystupuje v roli ovladače zařízení, přes systémové volání volá funkci *ddi_physmem_map(pf, vp, pages, flags)*, která namapuje fyzickou paměť *pf* na oblast adresového prostoru *vp* o počtu stránek *pages*.

Bez virtuální paměti je provedení mapování triviální, pokud se virtuální adresa *vp* shoduje s fyzickou adresou *fp*. To je zajištěno vložením kontroly na jejich rovnost přímo do funkce *ddi_physmem_map()*. Rovněž bylo nutné provést úpravy v příslušných ovladačích zařízení tak, aby oba parametry *vp* i *fp* měly stejnou hodnotu a mapování tak vždy uspělo.

5.2.5 Sdílení paměti s IPC

Sdílení paměti mezi procesy se provádí voláním *as_area_share()* v jádře systému na základě IPC systémových volání `IPC_M_SHARE_OUT` a `IPC_M_SHARE_IN`. V uživatelském prostoru se tato systémová volání téměř výhradně provádí přes tzv. *asynchronní framework* sadou funkcí *async_share_out_xxx()* resp. *async_share_in_xxx()*.

IPC_M_SHARE_OUT

Toto systémové volání používá proces pokud chce poskytnout některou svou paměťovou oblast ke sdílení jinému procesu. Současná implementace však neumožňuje druhému procesu zjistit virtuální adresu zdrojové oblasti nabízené ke sdílení, protože oba procesy mají samostatný adresový prostor a taková informace tudíž nedává smysl. Při společném adresovém prostoru je tato informace velmi podstatná, a proto muselo být rozhraní asynchronního frameworku rozšířeno.

Rozšíření se týká pouze funkce *async_share_out_receive(callid, dst, size, flags)* definované v souboru `uspace/lib/libc/generic/async.c`, ve které byl přidán parametr *dst* vracející volajícímu fyzickou adresu paměťové oblasti nabízené druhou stranou ke sdílení. Tento parametr našel využití také v případě režimu s MMU jednotkou, jelikož dosavadní praxe byla taková, že proces si vždy

musel sám provést vyhrazení volné paměti v rámci svého adresového prostoru voláním *as_get_mappable_page()*. Příklad reálného použití původní verze před rozšířením je následující:

```
...
if (!async_share_out_receive(&callid, &size, &flags)) {
  ipc_answer_0(callid, EHANGUP);
  return;
}
fs_va = as_get_mappable_page(size);
if (fs_va == NULL) {
  ipc_answer_0(callid, EHANGUP);
  return;
}
async_share_out_finalize(callid, fs_va);
...
```

Jedná se o inicializaci blokového zařízení reprezentovaného souborem, jehož implementaci lze najít v `uspace/srv/bd/file_bd/file_bd.c`. Klient tohoto blokového zařízení navázal IPC spojení reprezentované identifikátorem `callid` a zaslal požadavek na sdílení paměťové oblasti. Funkcí *async_share_out_receive()* je tento požadavek přijmut a proces do proměnné `fs_va` ukládá virtuální adresu, na níž se má vytvořit sdílená paměťová oblast. Nakonec blokové zařízení potvrdí druhé straně souhlas se sdílením voláním *async_share_out_finalize()*, kde v druhém parametru specifikuje zmiňovanou adresu pro sdílenou oblast.

Po úpravě rozhraní došlo ke zjednodušení tohoto kódu do následující podoby:

```
...
if (!async_share_out_receive(&callid, &fs_va, &size, &flags)) {
  ipc_answer_0(callid, EHANGUP);
  return;
}
async_share_out_finalize(callid, fs_va);
...
```

Pro režim bez MMU je fyzická adresa sdílené oblasti přímo vrácena do proměnné `fs_va` při volání funkce *async_share_out_receive()*. Rovněž v případě režimu s podporou MMU je do `fs_va` vrácena virtuální adresa, kde může být sdílená oblast umístěna, jelikož volání *as_get_mappable_page()* bylo zaintegrováno dovnitř měněné funkce.

Stejné úpravy byly provedeny i ve zdrojových souborech ostatních blokových zařízení, které jako jediné používají tento způsob sdílení. Úpravy v jádře systému zde nebyly vůbec nutné, protože jak zdrojová, tak i cílová (virtuální) adresa obou paměťových oblastí pro sdílení byla vždy k dispozici.

IPC_M_SHARE_IN

Systémové volání `IPC_M_SHARE_IN` je velmi podobné volání `IPC_M_SHARE_OUT`. Rozdíl je pouze ve směru sdílení, kde nyní žádá proces iniciující spojení o nasdílení oblasti definované v druhém procesu. Princip meziprocesové komunikace je stejný jako v předchozím případě s tím rozdílem, že nyní se používají funkce *async_share_in_xxx()*. Úprava rozhraní byla nutná i zde, a to ve funkci *async_share_in_start(phoneid, dst, size, arg, flags)*, jejíž parametr *dst* byl změněn na vstupní i výstupní. Po dokončení volání této funkce dochází k vytvoření sdílené paměťové oblasti v adresovém prostoru aktuálního procesu a jeho bázová virtuální resp. fyzická adresa je vrácena právě v tomto parametru.

Analogicky i zde bylo nutno upravit zdrojové kódy některých HelenOS aplikací, jmenovitě se jednalo o knihovny `libfs`, `libc`, `net` a aplikaci `klog`. Úpravy v jádře opět nebyly nutné, jelikož adresy paměťových oblastí z obou procesů byly k dispozici.

5.2.6 Procesy

Sdílený adresový prostor

Obrazy procesů jsou v HelenOS uloženy ve formátu ELF, ve kterém je během překladu stanovena cílová virtuální adresa, na kterou bude proces před svým spuštěním načten. To znamená, že každý proces je uzpůsoben na spuštění na stejné virtuální adrese, což v případě sdíleného adresového prostoru není možné zajistit. Existují minimálně dvě jednoduchá řešení tohoto problému.

První řešení je vložit při překladu do ELF obrazu kompletní relokační informaci, kterou lze použít při načítání obrazu k opravě (relokaci) těch míst v kódu či datech, kde byla při překladu uvažována pevná adresa. Relokační informace však může zabírat velkou část výsledného obrazu – až celou jeho polovinu. Další nevýhodou je nutnost provádět samotnou relokaci před spuštěním procesu, což spotřebovává jistý strojový čas a navíc je potřeba zabudovat tuto podporu do jádra či procesu, který je zodpovědný za toto spuštění – v případě systému HelenOS se jedná jak o aplikaci *program loader*, tak o modul jádra, který je zodpovědný za spuštění iniciálních procesů systému.

Druhým přístupem je nechat překladačem vygenerovat kód nezávislý na umístění, ze kterého je spuštěn – tzv. *position-independent code* (*PIC*). V takovém případě nejsou potřeba buď žádné relokační informace anebo pouze velmi malé množství ve srovnávní s úplnou relokační informací. Navíc tento přístup je již dlouhou dobu využíván ve většině operačních systémů k implementaci dynamicky linkovaných sdílených knihoven, jejichž kód musí mít PIC vlastnost, protože každý proces využívající sdílenou knihovnu může mít jiný požadavek na umístění knihovny ve svém virtuálním adresovém prostoru.

Tato práce se přiklonila k využití druhého popsaného řešení a sestavovací skripty pro uživatelské procesy byly upraveny pro generování PIC kódu, což je zcela jistě jednodušší a transparentnější změna, než provádět při spouštění procesů relokace celého jejich obrazu. Například překladač *gcc*, využívaný pro sestavování HelenOSu na většině procesorových architektur, poskytuje pro tento účel volbu `-fPIE`.

Načítání ELF obrazů do paměti

Specifickým problémem v případě HelenOS je zajištění správného načtení ELF obrazu procesu do jednotlivých paměťových oblastí. Tento problém i s jeho řešením byl popsán v sekci 5.2.3. Úpravy v implementaci, které následovaly toto řešení, byly provedeny v souborech `kernel/generic/src/lib/elf.c` a `uspace/src/loader/elf_load.c`. V obou případech se jednalo o zamčení či rezervaci části fyzické paměti před alokací paměťových oblastí, a to funkcemi *nommu_block_reserve()* a *nommu_block_free()*.

5.2.7 Správa paměti v uživatelském prostoru

Správa paměti uvnitř procesů je zajištěna v knihovně `libc` modulem `uspace/lib/libc/generic/malloc.c`. Původní implementace vytvářela jedinou paměťovou oblast pomocí systémového volání *as_area_create()*, kde se provádí alokace libovolně velkých bloků pro potřeby procesu. Dojde-li zde k vyčerpání paměti, alokátor se bude snažit zvětšit velikost paměťové oblasti a v případě neúspěchu nebude procesu další alokace paměti umožněna.

Tento způsob práce je přípustný pokud má každý proces samostatný adresový prostor, ale v případě sdíleného fyzického prostoru takový přístup velmi brzo selže s narůstajícím počtem současně spuštěných procesů. Z tohoto důvodu byl alokátor upraven tak, aby dokázal pracovat s více paměťovými oblastmi. Tedy v případě, že se nepodaří zvětšení velikosti aktuální oblasti, bude vytvořena nová voláním *as_area_create()*, v níž může alokace pokračovat.

Je také potřeba si uvědomit, že velikost alokovaných paměťových oblastí zde hraje velkou roli. Pokud je přítomna MMU jednotka, na její velikosti příliš nezáleží, protože skutečná alokace fyzických rámců se provádí až v případě, kdy k nim proces začne přistupovat. Naproti tomu bez podpory

stránkování dochází k alokaci fyzické paměti (rámců) již při vytváření paměťové oblasti. Pokud by byla ponechána stávající velikost jedné stránky (tj. 4 KB), pak u procesů náročných na paměťovou alokaci by zřejmě velmi často docházelo k vytváření stále nových paměťových oblastí, což jednak zpomaluje běh procesu (velké množství systémových volání) a také zvětšuje datové struktury jádra. Proto byla zavedena funkce *setheapsize(size)*, v níž může proces při svém startu specifikovat minimální velikost pro alokaci paměťové oblasti.

5.2.8 IA-32 závislý kód

Pro úplnost je také nutné zmínit úpravy, které byly provedeny ve zdrojových souborech týkajících se výhradně nízkoúrovňových funkcí jádra pro architekturu Intel IA-32. Pro účely implementace prototypového řešení na této procesorové architektuře bylo zcela vypnuto stránkování, avšak byla zachována segmentace, kterou nelze vypnout. Emulace režimu bez MMU tak probíhala tím způsobem, že každý segment jak jádra, tak i uživatelských procesů pokrýval celou fyzickou paměť. Výjimku z tohoto pravidla tvořily pouze uživatelské segmenty pro přístup k TLS datům (za použití segmentového registru `gs`) a dále popsaný segment pro přístup k XFI kontextu běžícího vlákénka (za použití segmentového registru `fs`).

5.3 Softwarová izolace pomocí XFI

Technika XFI byla popsána v sekci 3.2.4 a v této sekci bude uveden detailní postup její implementace. Řada informací zde uvedených přitom není součástí odborné publikace představující techniku XFI [9], nýbrž je výsledkem této diplomové práce. V případě, kdy tomu tak není, bude to v textu explicitně uvedeno.

Princip nasazení softwarové izolace v systému HelenOS je znázorněn na obrázku 5.2. Přestože tato technika zakládá na binární instrumentaci, současná implementace vyžaduje překlad zdrojových kódů aplikace s `libc` knihovnou zahrnující modul s funkcemi pro podporu běhového prostředí XFI (*XFI run–time*). Po přeložení zdrojových kódů kompilátorem *gcc* je výsledný obraz aplikace ve formátu ELF předán nástroji *ia32binrewriter*, který zajišťuje samotný přepis binárního kódu a vložení běhových kontrol. Výsledný obraz aplikace je spustitelný v systému HelenOS zahrnujícím rozšíření v jádře a také knihovně `libc`. Těmito rozšířeními se zabývá sekce 5.4.

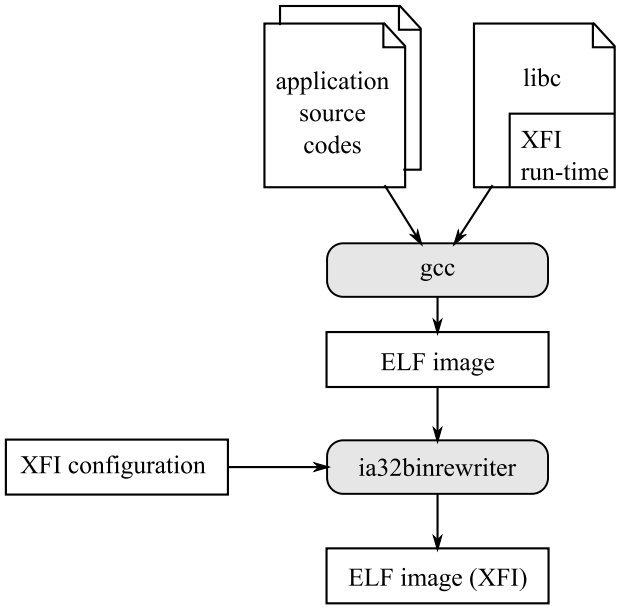
5.3.1 ia32binrewriter

ia32binrewriter je aplikace napsaná v programovacím jazyce C++ využívající *STATIF framework* (viz příloha D) pro binární editaci souborů ve formátu ELF [34]. Zdrojové kódy jsou umístěny na přiloženém CD v adresáři `Sources\ia32binrewriter`.

Princip fungování tohoto nástroje je následující. Nejprve je načten vstupní ELF soubor a všechny kódové sekce rozloženy na jednotlivé procedury a základní bloky, nad kterými jsou poté po jednom spuštěny algoritmy pro statickou analýzu a vkládání běhových kontrol. V původní terminologii XFI jsou tyto algoritmy označovány jako strážci (*guards*), čehož se drží i tato práce. Jakmile je vkládání běhových kontrol dokončeno, modifikovaný ELF obraz je uložen do původního souboru.

Jelikož binární instrumentace je poměrně náročný proces, potřebuje STATIF framework ke své správné funkci splnění několika prerekvizit:

- **Tabulka symbolů** – V instrumentovaném ELF souboru musí být přítomna tabulka symbolů zahrnující všechna návěští (procedury, cíle nepřímých skoků) vyskytující se v binárním kódu. Díky těmto symbolům může spolehlivě rozčlenit kód do jednotlivých procedur a ty pak dále na základní bloky.
- **Konstantní data mimo kódovou sekci** – Častým zvykem při překladu do binárního kódu je připojení konstantních dat aplikace na konec sekce se samotným kódem. Tento fakt působí



Obrázek 5.2: Překlad aplikací s podporou XFI techniky.

potíže při rozkladu binárního kódu na základní bloky a rovněž by mohlo dojít k aplikaci běhových kontrol na tato data, což by znamenalo jejich změnu. Z těchto důvodů musí být jakákoli data umístěna mimo sekce s kódem aplikace.

- **Konvenční podoba binárního kódu** – Strážci provádějící statickou analýzu očekávají dodržení jistých konvencí o podobě binárního kódu. Tyto konvence vyplývají z návrhu instrukční sady a pracovních registrů procesoru a jejich dodržování doporučují i samotní výrobci procesorů. Na platformě IA-32 se jedná např. o tyto:
  - Registr `esp` slouží jako ukazatel na vrchol zásobníku, pro práci se zásobníkem slouží instrukce `push`, `pop`.
  - Registr `ebp` slouží jako ukazatel na aktuální aktivační záznam, tedy přístupy na lokální proměnné a parametry procedury se provádí přes tento registr.
  - Instrukce `enter` a `leave` se používají k alokaci resp. uvolnění aktivačního záznamu.
  - Instrukce `call` a `ret` se používají k volání a návratu z procedury.

  Překladače vyšších programovacích jazyků takový kód samozřejmě generují, a proto všechny uživatelské aplikace HelenOS překládané kompilátorem *gcc* jsou vhodné pro binární instrumentaci.

5.3.2 Strážci a běhové kontroly

V rámci stávající implementace byla problematika softwarové izolace dekomponována do následujících strážců:

- **Strážce zásobníku** – provádí kontrolu přetečení a podtečení alokačního zásobníku běžícího vlákénka.
- **Strážce paměťových přístupů** – kontroluje oprávněnost zvolených typů paměťových přístupů.

```
typedef struct {
    void *begin;                /* begin address of a region */
    void *end;                  /* end address of a region */
} xfi_mregion_t;

typedef struct {
    xfi_mregion *read_regions;  /* array of regions with read access */
    xfi_mregion *write_regions; /* array of regions with write access */
    xfi_mregion *exec_regions;  /* array of regions with execute access */
    void *memory_bitmap;
} xfi_task_context;

typedef struct {
    void *shstk_ptr;            /* shadow stack pointer */
    void *shstk_top;            /* shadow stack top border */
    void *shstk_bottom;         /* shadow stack bottom border */
    void *stk_temp;             /* temporary memory within ordinary stack */
    void *stk_top;              /* ordinary stack top border */
    void *stk_bottom;           /* ordinary stack bottom border */
    void *tls;                  /* pointer to current TLS */
    int temp[4];                /* temporary storage */
} xfi_fibril_context_t;
```

Obrázek 5.3: Datové struktury využívané běhovými kontrolami.

- **Strážce přímých skoků** – ověřuje, zda cíle přímých skoků a volání náleží do kódu procesu.
- **Strážce nepřímých skoků** – ověřuje, zda cíle nepřímých skoků a volání náleží do kódu procesu.
- **Stínový zásobník** – zajišťuje integritu toku řízení při návratu z procedur a uchování hodnot některých důležitých registrů mezi voláním procedur.

Každý strážce má pro svou práci k dispozici celý binární kód vstupní aplikace, nad nímž provádí potřebné analýzy. Jeho výsledkem je zpravidla vložení nových instrukcí na stanovená místa binárního kódu sloužící jako běhová kontrola. Například strážce zásobníku vkládá kontrolu na přetečení či podtečení typicky před ty instrukce, které jistým způsobem manipulují s registrem ukazatele na vrchol zásobníku `esp`. Společnou vlastností běhových kontrol je samozřejmě zachování kontextu procesu, tj. zejména obsahu registrů a zásobníku, na nichž závisí další běh procesu. Z tohoto důvodu se před vložením běhové kontroly provádí statická analýza životnosti registrů procesoru a ty registry, které jsou analýzou označeny jako neživé[2] se použijí uvnitř těchto běhových kontrol. Pokud analýza nevrátí žádný neživý registr, avšak běhová kontrola ke své činnosti několik pomocných registrů potřebuje, některé registry jsou označeny jako pomocné a jejich hodnota je uschována a poté vyzvednuta ze zásobníku instrukcemi `push` a `pop` před, resp. po samotné běhové kontrole.

Řada běhových kontrol potřebuje ke své činnosti určité informace o kontextu běžícího procesu – např. zmíněné kontroly na přetečení a podtečení zásobníku musí mít k dispozici nejnižší a nejvyšší povolenou hodnotu registru zásobníku, vůči které budou porovnávat registr `esp`. Pro snadný přístup k těmto údajům byly vytvořeny datové struktury `xfi_task_context_t` a `xfi_fibril_context_t` popsané na obrázku 5.3. První z nich zahrnuje informace týkající se celého procesu, zatímco druhá obsahuje pouze data aktuálně běžícího vlákénka. Jejich aktuálnost zajišťuje XFI podpora v jádře

[2]Neživý registr je takový, do kterého se v bezprostředně následujících instrukcích zapisuje, ale před tímto zápisem není jeho hodnota čtena.

systému popsaná v sekci 5.4. Obě struktury jsou uloženy v paměti lineárně za sebou a přístupné pouze běhovým kontrolám přes segmentový registr `fs`, který není v HelenOS v uživatelském prostoru vyžíván a je použit i v originální implementaci XFI techniky. Bázová adresa segmentu s popisovačem v `fs` je nastavena přímo na začátek struktury `xfi_task_context_t`, takže přístup k položkám těchto struktur je opravdu velmi snadný.

V dalším textu bude podrobně rozebrána funkce a implementace každého strážce.

5.3.3 Strážce zásobníku

Strážce zásobníku zamezuje přetečení (*stack overflow*) či podtečení (*stack underflow*) alokačního zásobníku aktuálního vlákenka vkládáním běhových kontrol tam, kde dochází ke změně registru vrcholu zásobníku `esp` nebo to vyžaduje kontext situace. Původní koncept XFI rovněž zahrnuje tento typ strážce, avšak nepopisuje všechny okolnosti a situace vkládání běhových kontrol a ani jejich konkrétní podobu ve formě instrukcí procesoru.

Běhové kontroly

Strážce zásobníku implementuje pouze dvě běhové kontroly – kontrolu na přetečení v následující podobě

```
cmp   esp, [fs:stk_top]
jbe   xfi_stack_overflow_handler
```

a kontrolu na podtečení v následující formě:

```
lea   eax, [esp + Ds]
cmp   eax, [fs:stk_bottom]
jnb   xfi_stack_underflow_handler
```

Obě dvě se vkládájí až za instrukci, která mění hodnotu registru `esp`, nebo před instrukci, při jejímž vykonávání musí být korektnost ukazatele zásobníku garantována (viz další text). Dolní a horní mez paměti vymezené pro zásobník je uložena v položkách `stk_top` a `stk_bottom` struktury `xfi_fibril_context_t` (viz obrázek 5.3), na něž se obě běhové kontroly odkazují. U druhé běhové kontroly je porovnávaná hodnota `esp` zvětšena o jistou konstantu D_s, která je vypočtena na základě statické analýzy popsané v dalším textu. Díky její přítomnosti je kromě podtečení zkontrolován také přístup vlákénka do části zásobníku, v níž se obvykle nachází lokální proměnné procedury.

Pokud při porovnání hodnota v `esp` neleží v požadovaných mezích, řízení je předáno do procedury `xfi_stack_overflow_handler` resp. `xfi_stack_underflow_handler`, které zajistí ukončení procesu z důvodu chyby a předají do jádra systému informace o přetečení či podtečení zásobníku. Obě tyto procedury samozřejmě nemohou pro svou práci využívat aktuální zásobník běžícího vlákénka, jelikož `esp` ukazuje mimo paměťové regiony běžícího procesu, a proto používají dočasný zásobník rezervovaný každému vlákénku při jeho vzniku – ukazatel na jeho vrchol je v položce `xfi_fibril_context_t.stk_temp`.

Obě zmíněné kontroly mění při své činnosti 2 registry: registr `flags` obsahující stavové příznaky a registr `eax`. Jejich změna přitom může narušit další běh procesu, pokud nebyly statickou analýzou ověřeny jako neživé. Místo registru `eax` se samozřejmě použije libovolný neživý pracovní registr, pokud žádný takový není nalezen, je potřeba hodnotu `eax` před vstupem do běhové kontroly uschovat a poté obnovit. Pro tento účel opět není možné použít zásobník jako dočasné úložiště, a proto je uložen do pomocné položky `xfi_fibril_context_t.temp` následující dvojící instrukcí:

```
mov   [fs:temp], eax
...
mov   eax, [fs:temp]
```

V případě registru `flags` je situace komplikovanější, protože jeho hodnotu lze přímo získat pouze pomocí instrukce `pushf`, kterou nelze použít na stávajícím zásobníku. Z tohoto důvodu je jeho hodnota uložena do pomocného zásobníku a nakonec z něj vyzvednuta následující sekvencí instrukcí:

```
xchg  [fs:stk_temp], esp
pushf
xchg  [fs:stk_temp], esp
...
xchg  [fs:stk_temp], esp
popf
xchg  [fs:stk_temp], esp
```

Zjevně tento postup není příliš efektivní, jelikož se spotřebuje 6 procesorových instrukcí. Naštěstí situací, kdy je nutné v rámci této běhové kontroly uschovávat hodnotu `flags` je pouze malé množství.

Statická analýza

Je zřejmé, že použití běhové kontroly při každé změně ukazatele zásobníku by bylo velmi neefektivní. Proto byl navržen algoritmus pracující na úrovni procedur analyzovaného kódu, který využívá nástroje statické analýzy a snižuje počet vložených kontrol na minimum – v ideálním případě je vložena pouze jedna v rámci celé procedury.

Základem tohoto algoritmu je pro každou instrukci procedury vynucování platnosti invariantu o minimální počtu bytů, které jsou přístupné pro čtení a zápis nad i pod aktuálním ukazatelem vrcholu zásobníku. Formálně se tedy jedná o celočíselný interval adres

$$< \texttt{esp} - (K\ - D_u), \texttt{esp} + D_u + D_s >,$$

kde

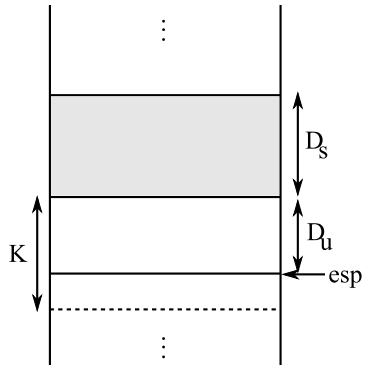
- K je kladná konstanta stanovená v době instrumentace, typicky malé číslo, např. 64,
- D_u[3] je hodnota v intervalu intervalu $< 0, K)$ počítaná v průběhu algoritmu,
- D_s[4] je kladná hodnota počítaná v průběhu algoritmu,
- `esp` je aktuální hodnota ukazatele zásobníku.

Význam D_u a D_s je nejlépe zřejmý z obrázku 5.4. D_u vyjadřuje počet bytů na zásobníku, o které byl `esp` zmenšen po provedení poslední běhové kontroly a D_s velikost intervalu ověřeného pro čtení i zápis touto poslední kontrolou. Tedy na aktuální instrukci máme jistotu, že paměťový interval $< \texttt{esp} + D_u, \texttt{esp} + D_u + D_s >$ je přístupný procesu pro čtení i zápis, jelikož byl dříve ověřen běhovou kontrolou. Rovněž interval $< \texttt{esp}, \texttt{esp} + D_u >$ je přístupný pro tytéž operace, protože D_u je vždy menší než K a běhové prostředí XFI vždy garantuje přístup alespoň ke K bytům pod hranící zkontrolovaného intervalu.

Princip statické analýzy tedy spočívá v průchodu celé procedury od jejího vstupního bodu až po všechny výstupní, detekce instrukcí pracujících s `esp` a na základě toho úprava hodnot D_u a D_s udržovaných v datové struktuře `StackGuardInfo` (viz obrázek 5.5). Tabulka 5.1 podává přehled o všech instrukcích, které jsou statickou analýzou ošetřeny a jejich typický vliv na hodnotu D_u.

Detailní popis algoritmu celé statické analýzy a vkládání běhových kontrol v rámci jediné procedury strojového kódu je uveden na obrázku 5.6. Zde jsou poznámky k jeho vybraným částem:

[3] D_u znamená *Distance Unsafe*
[4] D_s znamená *Distance Safe*



Obrázek 5.4: Invariant strážce zásobníku.

```
struct StackGuardInfo {
    int Du;
    int Ds;
    bool framePointerBound;
    int frameSize;
    ...
};
```

Obrázek 5.5: Datová struktura statické analýzy strážce zásobníku.

Instrukce	Změna hodnoty D_u
`add esp, L`	$+L$
`sub esp, L`	$-L$
`inc esp`	$+1$
`dec esp`	-1
`push`	-4
`pushf`	-4
`pusha`	-32
`pop`	$+4$
`popf`	$+4$
`popa`	$+32$

Tabulka 5.1: Vliv vybraných instrukcí na hodnotu D_u.

```
1: S = pseudotopologické uspořádání základních bloků;
2: for all b ∈ S do
3:    I = základní bloky předávající řízení do b;
4:    b.StackGuardInfo = Průnik(I);
5:    přiřaď b.StackGuardInfo do první instrukce základního bloku b;
6:    for all instrukce i ∈ b do
7:       přiřaď do i.D_u a i.D_s hodnoty z předchozí instrukce;
8:       if instrukce i přičítá nebo odečítá konstantu L od esp then
9:          i.D_u = max(0, i.D_u + L);
10:         if i.D_u = 0 and L < 0 then
11:            i.D_s = max(0, i.D_s + min(0, −L));
12:         end if
13:      else
14:         i.D_s = 0; /* způsobí vložení běhové kontroly */
15:         i.D_u = 0;
16:      end if
17:      if i.D_s = 0 or i.D_u > K then
18:         i.D_s = i.D_u + max(i.D_s, K);
19:         i.D_u = 0;
20:         vlož běhovou kontrolu před instrukci i;
21:      end if
22:   end for
23:   b.StackGuardInfo = hodnoty z první instrukce bloku b;
24:   PrověřCílovéBloky(b);
25: end for
```

Obrázek 5.6: Algoritmus strážce zásobníku.

- Jako první se provádí pseudotopologické setřídění grafu, kde vrchol je reprezentován základním blokem a hrana vede z bloku $b1$ do bloku $b2$ pouze pokud blok $b1$ podmíněně či nepodmíněně přímo předává řízení bloku $b2$. V takovém grafu samozřejmě mohou existovat cykly, proto nelze najít jeho topologické setřídění. Snahou pseudotopologického setřídění je pokusit se i přes existenci cyklů vybrat takové pořadí základních bloků, které zaručí, že pokud spuštění základního bloku $b2$ vždy musí předcházet spuštění bloku $b1$, pak je toto pořadí zachováno i v tomto setřídění. To není obecně možné, avšak kód generovaný z vyšších programovacích jazyků, kde se používají standardní způsoby řízení toku instrukcí tuto vlastnost obvykle splňuje. Standardní tok řízení představují příkazy větvení, smyčky, volání a návrat z procedur, ale již nikoli skoky napříč kódem jako je v jazyce C příkaz `goto`. Kompletní implementace algoritmu pseudotopologického setřídění se nachází v souboru `Sources\ia32binrewriter\PseudoTopologicalSort.cpp`.

- Funkce *Průnik()* vypočítá průnik struktur `StackInfoGuard` z více základních bloků, přičemž pokud daný blok ještě nebyl algoritmem zpracován, jeho struktura `StackGuardInfo` nebyla definována a v takovém případě se na průniku nepodílí. Algoritmus výpočtu je následující:

$$\begin{aligned} A &= \min_{\forall j}((D_s)_j + (D_u)_j), \\ D_u &= \min(A, \max_{\forall j}(D_u)_j), \\ D_s &= A - D_u, \end{aligned}$$

  kde j reprezentuje v uvedených formulách indexy hodnot z jednotlivých struktur `StackGuardInfo`. Průnik se v popsaném algoritmu uplatňuje v případě kdy více základních bloků, jejichž analýza již proběhla, mohou předávat řízení do jediného bloku. Takový blok musí samozřejmě převzít nejslabší předpoklady o hodnotách D_u a D_s, což je předmětem formulí popsaných výše.

- Na řádcích 9 až 11 dochází k úpravě hodnot D_u a D_s pokud i je jednou z instrukcí uvedených v tabulce 5.1. K modifikaci D_s však dochází pouze v situaci, kdy již není možné zmenšovat interval D_u, který má velikost 0. Pokud se jedná o jinou instrukci, která mění hodnotu v `esp` neznámým způsobem (např. přiřazuje mu hodnotu z jiného registru, jehož hodnotu nelze zjistit statickou analýzou), pak dochází k vynulování D_s i D_u.

- Splnění podmínky na řádku 17 je dostačující pro vložení běhové kontroly před aktuální instrukci. K tomu dochází v případě, kdy D_s je nulové, což rovněž implikuje nulovost D_u, a proto nelze bez běhové kontroly garantovat aktuální instrukci přístup k položkám na zásobníku. Druhým případem splnění podmínky je „přetečení“ hodnoty D_u, k čemuž typicky dochází při alokování datových struktur na zásobníku a vložení běhové kontroly je proto nezbytné, aby se zkontrolovalo, že na zásobníku je stále dost místa a nedošlo k jeho přetečení.

- Na konci zpracování každého základního bloku se provádí volání funkce *PrověřCílovéBloky()*. Ta kontroluje pro vstupní blok, jestli některý z nejvýše dvou cílových základních bloků, kterým může být předáno řízení, nebyl již zpracován tímto algoritmem a pokud ano, pak ověřuje jaké podmínky jsou vyžadovány na první instrukci pro hodnoty D_u a D_s. Pokud jsou tyto podmínky silnější než u vstupního bloku, je na konec vstupního bloku vložena běhová kontrola vynující jejich splnění.

Aktivační záznam

Dosud popsané algoritmy sledovaly změny pouze na registru `esp` a vkládaly běhové kontroly pro ověření jeho validity. Z pohledu paměťového strážce popsaného v následující sekci 5.3.4 je velmi

zajímavé sledovat také změny registru `ebp`. Ten se používá jako tzv. *frame pointer* nebo-li ukazatel na aktuální rámec či aktivační záznam procedury na zásobníku, ve kterém se nachází všechny lokální proměnné procedury a rovněž je přes něj odkazováno na vstupní parametry procedury. Hodnota `ebp` je typicky přiřazena v prvním základním bloku procedury a během jejího vykonávání se zpravidla nemění (s výjimkou volaných podprocedur). To lze snadno ověřit popsanou statickou analýzou a výslednou informaci udržovat v příznaku `StackGuardInfo.framePointerBound`. Jakmile je do registru `ebp` přiřazena hodnota z `esp` (např. přímo instrukcí `mov ebp,esp` nebo instrukcí `enter`), příznak je nastaven. Dojde-li k jakékoli změně `ebp`, příznak je vynulován. Velikost aktivačního záznamu (položka `StackGuardInfo.frameSize`) lze navíc snadno určit pomocí změn D_u, takže pokud je příznak `framePointerBound` nastaven, procesu lze garantovat přístup do oblasti paměti určené celočíselným intervalem adres

$$< \texttt{ebp} - \texttt{frameSize}, \texttt{ebp} + K > .$$

5.3.4 Strážce paměťových přístupů

Další důležité běhových kontroly poskytuje strážce paměťových přístupů definovaný v souboru `Sources\ia32binrewriter\MemoryAccessGuard.cpp`. Tyto kontroly ověřují u instrukcí paměťového zápisu (čtení), zda má proces oprávnění tento zápis (čtení) provést a pokud nikoli, je násilně ukončen.

Datové struktury a jejich prohledávání

Zda-li má proces právo číst nebo zapisovat na konkrétní adresu je zjištěno průchodem datových struktur ve struktuře `xfi_task_context_t` definované na obrázku 5.3. Současná implementace nabízí možnost uchovávat tyto informace dvěma způsoby.

První z nich, zmíněný také v originálním popisu XFI, představují *paměťové regiony* (struktura `xfi_mregion_t`), což jsou souvislé oblasti paměti určené počáteční a koncovou adresou, ke kterým může být stanoveno pouze jedno z těchto oprávnění přístupu: čtení, zápis, spuštění. Jelikož paměť procesů v HelenOS je typicky složena z více paměťových oblastí (`as_area_t`), paměťové regiony, které jsou přímo mapovány na některé z těchto paměťových oblastí, jsou reprezentovány polem struktur `xfi_mregion_t`, přičemž poslední prvek tohoto pole má všechny položky nulové. Průchod takovým polem je poměrně rychlý za předpokladu jeho malé velikosti, a proto i ověření, zda daná adresa spadá do některého z definovaných regionů je efektivní. Pole obsahující paměťové regiony jsou rozdělena podle přístupových práv, tedy na `read_regions`, `write_regions` a `exec_regions`.

Pro dosažení efektivní implementace byl průchod implementován přímo v instrukcích assembleru, například pro ověření práva zápisu 4 byte na adresu uloženou v registru `eax` vypadá implementace následovně:

```
        lea   ecx, [eax + 4]    # ecx points behind the last byte to write
        mov   ebx, [fs:write_regions]
                                # ebx points to list of write regions
iter:   cmp   eax, [ebx]
        jb    next              # address is below memory region
        cmp   ecx, [ebx + 4]
        jbe   success           # address is within memory region

next:   add   ebx, 8            # go to next memory region
        cmp   [ebx + 4], 0      # end of array?
        jnz   iter
        jmp   failure
```

Registr `ebx` je inicializován na adresu první položky pole `write_regions` a při jeho průchodu se kontroluje dolní interval ověřované adresy (hodnota `eax`) s počáteční adresou regionu (hodnota `[ebx]`) a horní interval (hodnota `ecx`) s koncovou adresou regionu (hodnota `[ebx + 4]`). Spadá-li ověřovaný interval adres do některého z regionů, cyklus skončí skokem na návěští `success`, v jiném případě na návěští `failure`. Uvedená část kódu samozřejmě postrádá uschování hodnot registru `ecx` a `ebx` na zásobník.

Druhý způsob správy přístupných regionů je použití paměťové bitmapy (položka `memory_bitmap`). Každý její byte popisuje práva pro přístup do paměťové buňky o pevné velikosti (typicky velikost stránky, tj. 4 KB), přičemž první bit definuje právo pro čtení, druhý bit právo pro zápis a třetí bit právo spuštění. Velikost celé bitmapy je proporciální vzhledem k celé fyzické paměti, takže např. pro 32MB fyzický prostor se bude jednat o pouhých 8 KB. Výhodou bitmapy je ověření práv u dané adresy v konstantním čase nezávislém na počtu paměťových regionů, což lze velmi snadno implementovat pouze několika strojovými instrukcemi. Například pokud by registr `eax` obsahoval adresu zápisu, pak ověření je implementováno následovně:

```
shr   eax, 12                 # memory unit size = 4KB
add   eax, [fs:memory_bitmap] # get bitmap cell
test  byte ptr [eax], 0x2     # 0x2 = write permission bit
jz    failure
```

První instrukce převádí adresu na index do paměťové bitmapy, druhá instrukce přičítá bázovou adresu bitmapy a nakonec ve třetí instrukci dochází k ověření přístupového práva pro zápis. Celá kontrola má tedy konstantní složitost a spotřebovává pouze 4 instrukce procesoru. Poslední instrukce skoku by přitom neměla narušit superskalární zpracování instrukcí, jelikož procesory rodiny IA-32 implicitně předpokládají, že dopředný podmíněný skok nebude vykonán, nemají-li o něm žádné informace z předchozích průchodů. Z tohoto důvodu je cíl tohoto skoku umístěn za ELF sekcí s běžným kódem.

Potencionálním problémem této běhové kontroly může být, že vypočtený index překročí velikost paměťové bitmapy a instrukce `test` bude přistupovat mimo povolené paměťové oblasti. Navíc v tomto případě by ověření mohlo skončit kladným výsledkem, přestože původní adresa přistupuje mimo dostupnou fyzickou paměť. Provedený pamětový přístup na tuto adresu by tak mohl např. narušit chod zařízení namapovaného do fyzického adresového prostoru. V této implementaci však takové situace neuvažujeme a pokud by bylo potřeba jim předcházet, daly by se vyřešit rozšířením běhové kontroly testem na překročení horní hranice fyzické paměti např. takto:

```
cmp   eax, MAX_PHYSMEM_ADDR
ja    failure
```

Druhý problém zde činí intervalové dotazy, kde je výhodnější použít první techniku průchodu pole s paměťovými regiony. Na druhou stranu intervalové dotazy se v běžném kódu nevyskytují příliš často, obvykle pouze při přesunech větších bloků dat implementovaných jedinou instrukcí (např. `rep movs`). Nejčastěji se používají instrukce zápisu/čtení celého slova (v případě IA-32 o velikosti 4 byte), kde lze metodu s paměťovými bitmapami uplatnit velmi dobře. Problém však představují slova, která zasahují do dvou sousedních paměťových buněk, jelikož ověření práva přístupu popsané výše uvažuje pouze tu buňku, do které patří počáteční adresa slova. To je řešeno tím, že každá paměťová oblast je při alokaci rozšířena o další stránku, která již není uvedena v paměťové bitmapě, což je opět zajištěno XFI podporou v jádře HelenOS popsanou v sekci 5.4.

Vkládání běhových kontrol

Samotnou činností strážce paměťových přístupů je pouze vkládání výše uvedených běhových kontrol či volání na ně. Strážce prochází každý základní blok aplikace a v závislosti na svém konfiguračním nastavení se zabývá pouze těmi instrukcemi, které provádějí zápis, čtení nebo spuštění paměti.

Výjimkou jsou instrukce pracující se zásobníkem, které sice provádějí zápis nebo čtení paměti zásobníku, ale ty jsou ošetřeny v rámci strážce zásobníku. Jedná se zejména o instrukce `push`, `pop`, `enter`, `leave` a `ret`.

Mějme například instrukci zápisu `mov dword [eax+4],123`, kde hodnota `123` je zapisována na adresu určenou výrazem `eax+4` a předpokládejme, že poté následuje instrukce zapisující hodnotu do registru `edx`. Oba druhy běhové kontroly formy běhové kontroly pro tento zápis vypadají následovně:

```
lea   edx, [eax+4]
call  xfi_mem_check_edx_4
mov   dword ptr [eax+4], 123
```

```
lea   edx, [eax+4]
shr   edx, 12
add   edx, [fs:memory_bitmap]
test  byte ptr [edx], 0x2
jz    failure
mov   dword ptr [eax+4], 123
```

Vlevo je uvedena varianta při použití průchodu pole paměťových regionů, zatímco napravo varianta s použitím paměťové mapy. Princip obou variant je zcela totožný. Cílová adresa zápisu je získána do registru `edx` a poté předána samotné kontrole.

V prvním případě se volá běhová kontrola `xfi_mem_check_edx_4` přijímající `edx` přímo jako vstupní argument. Pro zmenšení režie na běh i velikost kódu byla vytvořena celá sada těchto procedur lišící se pouze ve vstupních argumentech: v registru obsahující kontrolovanou adresu a v počtu kontrolovaných byte. Počet byte lze předat buď implicitně (jako 4 byty v uvedeném případě) nebo explicitně postřednictvím registru `ecx`. Jméno každé procedury pak samozřejmě odráží vstupní argumenty. Implementace těchto procedur je možné najít ve zdrojových souborech knihovny `libc` v souboru `uspace/lib/libc/arch/ia32/src/xfi.S`.

V druhém případě se jedná o kontrolu přes paměťovou bitmapu, která přímo pracuje s hodnotou v registru `edx`. Oproti první variantě je efektivnější co se režie běhu týče, avšak zabírá více procesorových instrukcí.

V případě instrukcí, které provádějí čtení či zápis větších oblastí paměti, je princip běhové kontroly stejný, registr `ecx` však není obvykle nutné ručně vyplňovat, jelikož tyto instrukce jej samy o sobě vyžadují jako vstupní argument udávající počet bytů pro zápis/čtení. Jedná se o instrukce `movs`, `outs`, `lods`, `ins`, `stos`, `cmps`, `scas`, kterým je předřazen instrukční prefix `rep`, `repz` nebo `repe`. Problémem, který se zde musí řešit je směr, v jakém instrukce paměťovou operaci provádí, tj. zda dochází k inkrementaci či dekrementaci registru s cílovou adresou. To je stanoveno příznakem `DF` (*Direction Flag)*. Při vkládání běhové kontroly se proto nejprve statickou analýzou ověřuje, zda tento příznak není nulován (instrukce `cld`) nebo nastavován (instrukce `std`) a podle toho je také upravena adresa a předána do volané procedury kontrolující práva na provedení operace. Pokud ze statické analýzy nelze hodnotu `DF` rozpoznat, je vložena sekvence instrukcí zjišťující její hodnotu dynamicky, tj. za běhu procesu, a provádějící dynamické nastavení adresy pro kontrolu.

Paměťové přístupy do TLS

Doposud popsané kontroly předpokládaly, že kontrolovaná adresa reprezentuje globální adresu v rámci fyzické paměti. Lokální úložiště dat vlákének (TLS) v systému HelenOS však využívá adresy relativní od začátku TLS pomocí adresace se segmentovým registrem `gs`. Např. adresa `[gs:0]` ukazuje na začátek TLS. Běhové kontroly při ověřování adresy směřující do TLS proto provádí přičtení bázové fyzické adresy TLS umístěné v položce `xfi_fibril_context_t.tls`.

Optimalizace běhových kontrol

Řada přístupů procesu do paměti nemusí vůbec podléhat běhovým kontrolám, jelikož jejich kontrolu lze provést staticky. Pravděpodobně největší skupinu těchto přístupů tvoří instrukce pracující

s lokálními proměnnými umístěnými na zásobníku přes registr `esp` nebo `ebp`. Nachází-li se výsledná adresa paměťového přístupu v rámci intervalu adres, který byl staticky ověřen strážcem zásobníku, běhová kontrola není vložena. Strážce zásobníku poskytuje informace o těchto intervalech ve struktuře `StackGuardInfo` pro každý analyzovaný základní blok i instrukci.

Rovněž instrukce pracující s globálními proměnnými, na které je odkazováno přímo globální adresou v rámci datového segmentu, nemusí být předmětem běhové kontroly, protože ověření adresy lze učinit statickou analýzou. V této implementaci však tato optimalizace není využita, jelikož se pracuje s pozičně nezávislým kódem, u kterého se přístup k globálním proměnným provádí výhradně přes *GOT* tabulku [34] a nikoli přímo.

5.3.5 Strážce přímých skoků

Strážce definovaný v souboru `Sources\ia32binrewriter\DirectControlFlowGuard.cpp` implementuje statickou a dynamickou kontrolu všech přímých skoků v kódu instrumentované aplikace, tak jak navrhuje XFI resp. CFI technika. Provádí sekvenční průchod všemi instrukcemi a odkazuje-li některý přímý skok mimo oblasti kódu aplikace (což lze u přímých skoků vždy staticky ověřit), je před něj vloženo volání běhové kontroly, která běžící proces vždy ukončí. Původní návrh zde bylo nutné rozšířit také o kontrolu na skoky směřující doprostřed instrukcí v rámci kódu aplikace a skoky na procedury běhových kontrol.

5.3.6 Strážce nepřímých skoků

Nepřímé skoky jsou ošetřeny strážcem ve zdrojovém souboru `Sources\ia32binrewriter\IndirectControlFlowGuard.cpp`. Jedná se o skoky (instrukce `jmp`) či volání (instrukce `call`) na adresy, které jsou dynamicky vypočítány při běhu procesu a nelze je proto ověřit staticky. Implementace tohoto strážce se řídí návrhem techniky CFI a provádí následující dvě činnosti:

- vložení instrukce návěští s pevným identifikátorem před každý potencionální cíl nepřímého skoku (odpovídá virtuální instrukci `label ID`) a
- vložení běhové kontroly před každý nepřímý skok, která ověří, že na cílové adrese je přítomna instrukce návěští s požadovaným identifikátorem (odpovídá virtuální instrukci `call ID,dest`).

Vše osvětlí následující příklad. Mějme instrukci nepřímého volání `call edi` a uvažujme, že jeho cílem je instrukce `push ebp`:

```
    prefetchnta [0x12345]
T:  push  ebp       # target of indirect call
    ...

    push  eax
    mov   eax, cs:[edi - 4]
    dec   eax
    cmp   eax, 0x12344
    jnz   failure
    pop   eax
    call  edi       # indirect call
```

Pak před instrukci `push ebp` je vložena instrukce `prefetchnta` s identifikátorem `0x12345`, která reprezentuje instrukci návěští `label ID`. Původní význam této instrukce je načtení řádku paměti do cache procesoru z adresy uvedené v argumentu. Jelikož však nemá tato instrukce žádné vedlejší efekty a nijak nemění sémantiku kódu, tak dojde-li k jejímu nechtěnému spuštění, nebude to mít na proces žádný vliv. To byl také hlavní důvod volby této instrukce autory CFI techniky.

Instrukce `call edi` je zde instrumentována více instrukcemi, které nejprve načtou 4 byty předcházející cílové adrese do registru `eax`, provedou jejich dekremetaci a poté je porovnají s hodnotou `0x12344`, což je o jedna snížená hodnota identifikátoru návěští. Jak si lze všimnout, 4bytová hodnota `0x12345` je díky instrukci `prefetchnta` předřazena přímo před instrukci `push ebp` představující v tomto příkladě cíl nepřímého skoku a porovnání tedy uspěje. Pokud by porovnání selhalo, znamenalo by to, že cílem skoku je instrukce, která nebyla opatřena instrukcí návěští, a proto procesu nemá být dovoleno předat jí řízení. V takovém případě je uskutečněn skok na návěští `failure`, ve kterém je provedeno násilné ukončení procesu.

Skutečnost, že hodnota identifikátoru v kódu běhové kontroly je snížena o jedničku je velmi podstatná: pokud by hodnota identifikátoru byla uvedena přímo ve strojovém kódu běhové kontroly, proces by teoreticky mohl z jiného místa skočit doprostřed této kontroly pomocí nepřímého skoku (jeho běhová kontrola by totiž uspěla), k čemuž nesmí dojít.

Za potencionální cíle všech nepřímých skoků jsou považovány vstupní základní bloky všech procedur a rovněž také všechny symboly v ELF souboru ukazující do kódových sekcí, což je obvykle případ návěští `case` větví příkazu `switch` v jazyce C. Tento příkaz bývá velmi často překladači optimalizován vytvořením tabulky cílových návěští, z níž je za běhu vybrána adresa návěští na základě hodnoty vstupního výrazu `switch` příkazu.

Důležitou roli hraje také volba samotné hodnoty identifikátoru, která se nesmí shodovat s žádnou 4bytovou hodnotou obsaženou v instrumentovaném kódu. Současná implementace provádí náhodný výběr identifikátoru a kontroluje jeho výskyt ve všech sekcích obrazu ELF. Pokud je přítomen, vygeneruje se další náhodný identifikátor a pokračuje se stejným způsobem do chvíle nalezení unikátního identifikátoru.

5.3.7 Stínový zásobník

Stínový zásoník je posledním prostředkem zajištění integrity toku řízení a je rovněž navržen v rámci XFI techniky. Jedná se o pomocný zásobník, který není přístupný kódu procesu a pracovat s ním mohou pouze instrukce, které jsou výsledkem instrumentace (tj. běhové kontroly). Jeho primárním účelem je uchovávání návratových adres procedur z klasického zásobníku, čímž lze detekovat, jestli návratová adresa na původním zásobníku byla procesem (náhodně či záměrně) přepsána.

Původní návrh v XFI nijak nespecifikuje jaká další data jsou předmětem správy ve stínovém zásobníku a ani nezmiňuje přesný způsob práce s ním. Implementace zde popsaná ukládá na zásobník položky popsané tabulkou 5.2. Kromě návratové adresy je totiž výhodné uchovávat také obsah registrů `esp` a `ebp`. Uchováním hodnoty `esp` se optimalizuje práce strážce zásobníku, protože ten by musel po každém volání procedury (instrukce `call`) vkládat běhovou kontrolu na přetečení a podtečení aktuálního vrcholu zásobníku v `esp`. Díky tomu, že běhová kontrola stínového zásobníku zaručí při návratu z procedury nastavení původní hodnoty `esp`, vložení kontroly strážcem zásobníku není nutné. Stejná situace platí pro registr `ebp`, na jehož korektní hodnotu se spoléhá strážce paměťových přístupů mezi voláními procedur. Jelikož tento registr je z doporučení výrobce procesoru vhodný používat jako ukazatel na aktivační záznam procedury, jeho hodnota by proto před i po volání procedury měla zůstávat stejná, což je opět zaručeno jeho uschováním ve stínovém zásobníku.

Implementace stínového zásobníku se nachází ve zdrojovém souboru `Sources\ia32binrewriter\ShadowStackProvider.cpp` a jejím jediným účelem je instrumentace instrukcí volání a návratu z procedur v každém základním bloku kódu aplikace. Tedy instrukce

```
call <proc>                    ret
```

jsou instrumentovány takto:

Hodnota	Význam
`ebp`	Ukazatel na aktivační záznam na zásobníku
`esp`	Ukazatel na vrchol zásobníku
ret adr	Návratová adresa

Tabulka 5.2: Hodnoty ukládáné na stínový zásobník.

```
call xfi_shadow_stack_func_entry_5_eax    call xfi_shadow_stack_func_exit
call <proc>                               ret
```

Implementace obou pomocných procedur jsou uvedeny v souboru `uspace/lib/libc/arch/ia32/src/xfi.S`. Procedura `xfi_shadow_stack_func_entry_5_eax` provádí uložení výše zmíněných hodnot na stínový zásobník a jako pomocný registr využívá `eax`, který byl ověřen statickou analýzou jako neživý. Obecně může být použit jakýkoli pracovní registr mimo `esp` a `ebp` pro tento účel, je-li neživý. Procedury `xfi_shadow_stack_5_xxx` jsou z tohoto důvodu implementovány ve více variantách – každá z nich využívá jeden z těchto registrů jako pomocný, jehož název tvoří poslední část jména procedury. Číslice `5` ve jméně vyjadřuje velikost instrukce `call <proc>` v bytech, což je důležitý vstupní argument, na jehož základě dochází k výpočtu návratové adresy ukládáné do stínového zásobníku. Velikost instrukce `call` na architektuře IA-32 může být v rozmezí 2 až 7 byte. Pokud není nalezen žádný neživý registr (např. z toho důvodu, že se jedná o nepřímé volání, kdy není možné staticky prověřit cílový kód) nebo je velikost instrukce `call` jiná než 5 byte, pomocný registr je stanoven až uvnitř procedury ukládající na stínový zásobník a jeho hodnota je zachována.

Procedura `xfi_shadow_stack_func_exit` provádí vyzvednutí hodnot ze stínového zásobníku a kontrolu návratové adresy ze stínového i klasického zásobníku a obnovení hodnot `esp` a `ebp`. Nesouhlasí-li vyzvednuté návratové hodnoty, proces je násilně ukončen. Obě pomocné procedury samozřejmě provádějí také kontrolu na přetečení resp. podtečení stínového zásobníku.

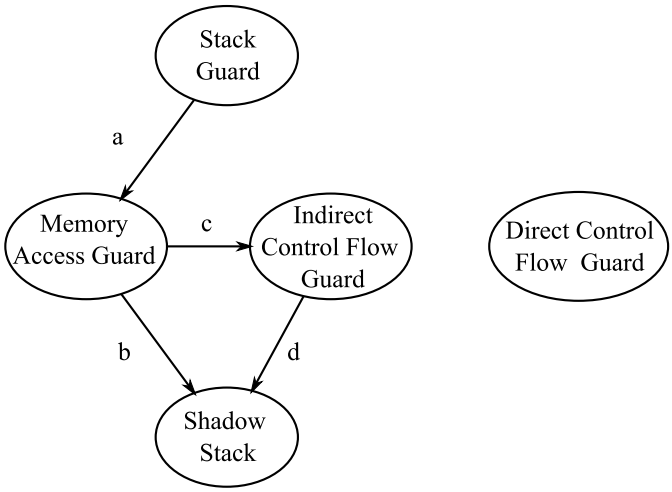
Samostatná informace o aktuálním vrcholu stínového zásobníku i jeho horní a dolní hranici je uložena ve položkách struktury `xfi_fibril_context_t` (viz obrázek 5.3).

5.3.8 Pořadí strážců

Přestože každý z popsaných strážců (včetně poskytovatele stínového zásobníku) pracuje nezávisle na ostatních, pořadí v jakém jsou spuštěny hraje významnou roli. Toto pořadí totiž určuje v jaké „blízkosti“ se bude nacházet vložená kontrola od instrukce, která byla předmětem kontroly u více strážců současně. Běhová kontrola posledního spuštěného strážce bude samozřejmě přímo předcházet této instrukci. Z tohoto důvodu bylo nutné stanovit pevné pořadí běhu strážců na základě závislostí daných vkládanými běhovými kontrolami. Pořadí je dáno topologickým setříděním grafu na obrázku 5.7, přičemž závislosti mezi strážci dle obrázku jsou následující:

a Závislost paměťového strážce na strážci zásobníku je dána zejména tím, že strážce zásobníku vyplňuje pro každou instrukci a základní blok strukturu `StackGuardInfo` s informací o aktuálním stavu ukazatele zásobníku a ukazatele na aktivační záznam. Tyto informace využívá paměťový strážce k četným optimalizacím, které snižují celkový počet vložených běhových kontrol.

b Instrukce nepřímého volání procedury (`call`) je předmětem běhové kontroly paměťového strážce (ověření spustitelnosti paměťového regionu cílové adresy) i stínového zásobníku. Kód vložený stínovým zásobníkem však musí vždy být v těsné blízkosti instrukce `call`, protože na



Obrázek 5.7: Graf udávající pořadí spuštění strážců.

základě tohoto předpokladu upravuje návratovou hodnotu vloženou na stínový zásobník. To znamená, že toto vložení musí proběhnout až po dokončení práce všech ostatních strážců.

c Tato závislost je způsobena instrukcí nepřímého skoku či volání (např. `jmp ecx`), při němž je z logického hlediska nejprve nutné ověřit, zda proces má právo spuštění kódu na cílové adrese (úloha paměťového strážce) a až poté provádět ověření na přítomnost požadovaného unikátního identifikátoru v instrukci `prefetchnta`, která musí předcházet cílové instrukci (úkol strážce nepřímých skoků). Při opačném pořadí těchto kontrol by mohlo dojít k tomu, že budou přečtena data z paměťové oblasti, do které proces nemá přístup a pokud by na dané adrese bylo namapováno některé zařízení, mohlo by to narušit jeho běh resp. jeho komunikaci s příslušným ovladačem. Při standardním nastavení běhových kontrol však proces může číst data z libovolných adres, ale pokud dojde zapnutí kontroly také na všechna paměťová čtení v paměťovém strážci, procesu by to mělo být zcela znemožněno.

d Závislost *d* je zcela identická se závislostí *b*, tedy vložený kód pro stínový zásobník musí být vždy přímo u instrukce volání `call` i v případě, kdy se aplikují běhové kontroly pro nepřímé skoky.

Zcela mimo graf závislostí stojí strážce přímých skoků, jelikož přímé skoky nejsou nikdy předmětem instrumentace u ostatních strážců. Ve výsledném pořadí byly zařazeny na předposlední místo, tj. před poskytovatele stínového zásobníku.

5.4 Systémová podpora pro XFI

Pro správný chod běhových kontrol vkládaných strážci jsou nezbytné údaje o přístupných paměťových regionech procesu a umístění a velikosti zásobníků pro každé vlákno resp. vlákénko. Tyto údaje by pravděpodobně bylo možné získat a udržovat přímo v procesu bez zásahů do jádra (o čemž se zmiňuje i původní návrh XFI), ale výhodnější je poskytovat tyto informace běhovým kontrolám přímo z jádra systému, kde jsou již k dispozici. Předmětem této části textu bude popis rozšíření jádra HelenOS i uživatelské knihovny `libc`, které umožní předávat nezbytné informace o procesu a vláknech do prostoru uživatelského procesu instrumentovaného technikou XFI. I přes popsané změny však bude stále možné provozovat v HelenOS běžné aplikace, které neprošly instrumentací.

Úpravy prováděné ve zdrojových kódech byly vždy uzavřeny mezi direktivy podmíněného překladu `#if` and `#endif` a k jejich překladu dojde pouze v případě definice symbolu `CONFIG_NOMMU_XFI`.

5.4.1 Datové struktury jádra

Běhové kontroly definovaly dvě základní datové struktury `xfi_task_context_t` a `xfi_fibril_context_t` (obrázek 5.3) sdružené do struktury `xfi_context_t`, jejíž instance je umístěna v paměti procesu. `xfi_task_context_t` popisuje stav procesu, což v současné chvíli zahrnuje pouze seznam dostupných paměťových regionů, `xfi_fibril_context_t` poskytuje informace o alokačním zásobníku a stínovém zásobníku aktuálně běžícího vlákénka. Vzhledem k tomu, že vlákénka jsou pouze doménou uživatelského prostoru (definovány a provozovány v rámci knihovny `libc`) a nejmenší jednotkou plánování v jádře je vlákno, bylo nutné zaintegrovat informace o vlákénkách také do jádra. Výsledné změny provedené na datových strukturách jádra jsou následující:

- Přidána struktura `xfi_fibril_container_t` v souboru `kernel/generic/include/nommu/xfi.h` obsahující identifikaci vlákénka a instanci `xfi_fibril_context_t`. Vlákénko je jednoznačně identifikováno pomocí ukazatele na jeho úložiště lokálních dat (*thread-local storage*).
- Struktura `task_t` v souboru `kernel/generic/include/proc/task.h` rozšířena o ukazatel na instanci struktury `xfi_context_t` přímo do paměti procesu a také rozšířena o seznam struktur `xfi_fibril_container_t` – po jedné pro každé vlákénko vytvořené uživatelským procesem.
- Struktura `thread_t` v souboru `kernel/generic/include/proc/thread.h` rozšířena o ukazatel na instanci `xfi_fibril_container_t` posledního běžícího vlákénka v rámci tohoto vlákna.

Souvislosti těchto změn jsou znázorněny na obrázku 5.8, kde je schématicky zobrazena struktura jediné úlohy se dvěma vlákny a třema vlákénky. V horní části obrázku se nachází adresový prostor procesu reprezentovaný touto úlohou. Ze struktury `task_t` vede ukazatel přímo do paměti procesu s instancí `xfi_context_t`. Dále je zde ukazatel na spojový seznam všech vlákénkek procesu, přičemž každé vlákno má přiřazeno právě jedno (běžící) vlákénko z tohoto seznamu.

5.4.2 Práce s vlákny a vlákénky

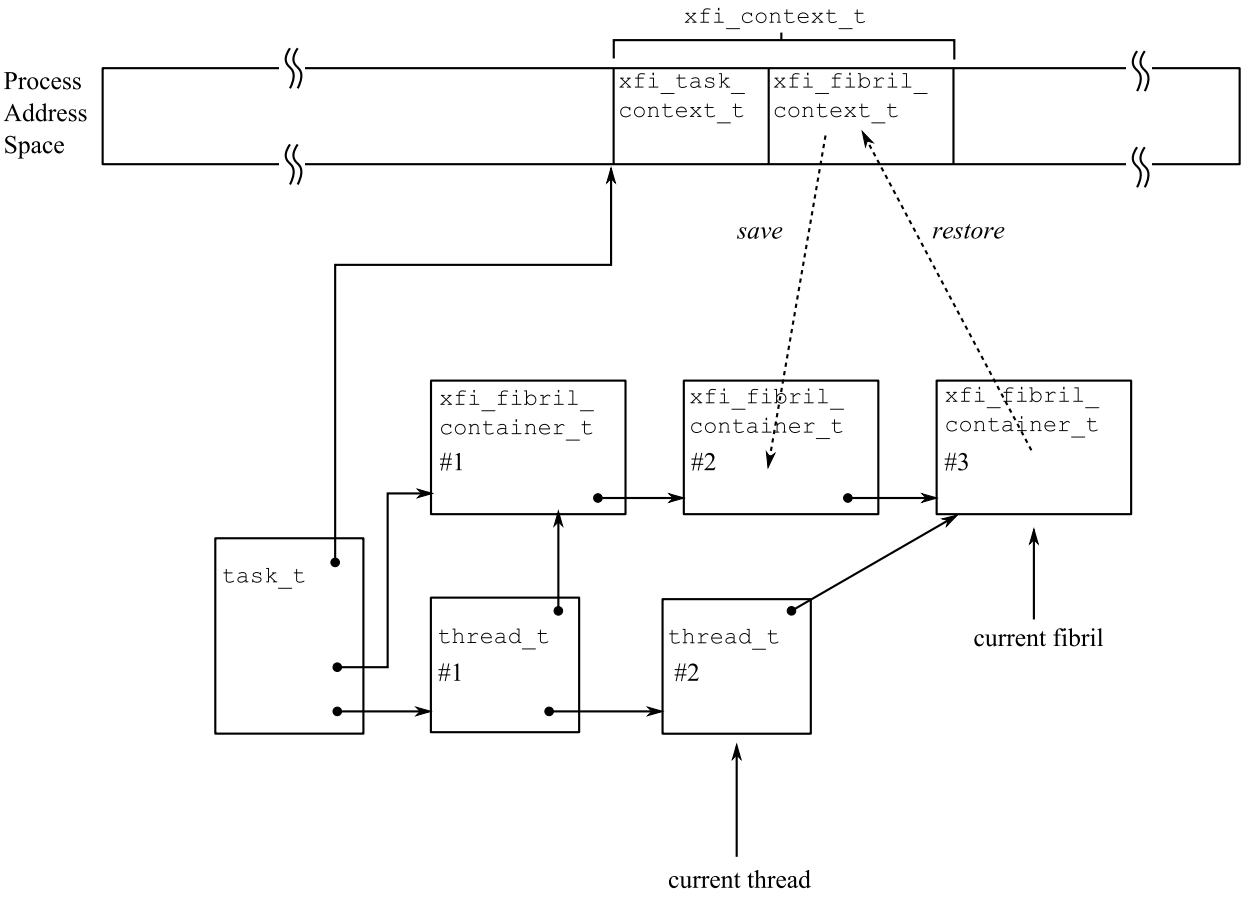
Jelikož správu vlákének zajišťuje knihovna `libc` v uživatelském prostoru, byla zavedena dvě nová systémová volání, kterými lze jádro informovat o vzniku resp. zrušení každého vlákénka:

- *sys_xfi_notify_fibril_created(tls_ptr, stack_ptr, stack_size)* – informuje o vytvoření vlákénka s daným ukazatelem na lokální úložiště dat *tls_ptr* a s ukazatelem na jeho zásobník, v jádře provede vytvoření instance `xfi_fibril_container_t` (včetně alokace stínového zásobníku) zatím bez přiřazení konkrétnímu vláknu,
- *sys_xfi_notify_fibril_terminated(tls_ptr)* – informuje o zrušení vlákénka, provede odstranění instance `xfi_fibril_container_t` v jádře (včetně uvolnění stínového zásobníku).

Tato volání byla přidána přímo do příslušných funkcí vytvoření a zrušení vlákénka v knihovně `libc`.

Další nezbytnou informací pro jádro je notifikace o přepínání vlákének, což je opět zcela pod kontrolou funkcí `libc`. Při této události je totiž nutné provést nejprve uložení stávající instance `xfi_fibril_context_t` aktuálně běžícího vlákénka (dále jen XFI kontext) do datových struktur jádra a poté nastavit nové hodnoty této instance odpovídající vlákénku, na které má být přepnuto řízení. Údaje v této instanci intenzivně využívají běhové kontroly strážce zásobníku a stínového zásobníku. Na obrázku 5.8 odpovídají tomuto postupu šipky s popisky *save* u vlákénka #2 a *restore* u vlákénka #3 – tj. dochází k přepnutí z #2 na #3.

Je evidentní, že jednodušším postupem by bylo nastavení ukazatele v paměti procesu na instanci `xfi_fibril_container_t`, která se nachází v jádře systému. Jelikož jak jádro, tak i všechny procesy sdílejí společný adresový prostor, proces by bez problémů mohl přes takový ukazatel přistupovat na



Obrázek 5.8: Schéma podpory XFI v jádře HelenOS.

Systémové volání	Obalující/notifikační volání	Význam
sys_as_area_create()	*xfi_sys_as_area_create()*	Vytvoří novou oblast
sys_as_area_resize()	*xfi_sys_as_area_resize()*	Změní velikost oblasti
sys_as_area_change_flags()	*xfi_sys_as_area_change_flags()*	Změní příznaky oblasti
sys_as_area_destroy()	*xfi_sys_as_area_destroy()*	Zruší oblast
sys_physmem_map()	*xfi_notify_physmem_map()*	Namapuje část fyzické paměti do oblasti
IPC_M_SHARE_OUT	*xfi_notify_as_area_shared()*	Sdílení oblasti přes IPC volání
IPC_M_SHARE_IN	*xfi_notify_as_area_shared()*	Sdílení oblasti přes IPC volání

Tabulka 5.3: Systémová volání vztažená k paměťovým oblastem a k nim příslušející obalující resp. notifikační funkce.

kontext vlákénka. Z pohledu běhových kontrol by to však znamenalo minimálně o jednu strojovou instrukci navíc, ve které by musely přečíst hodnotu tohoto ukazatele a provést jeho dereferenci. Proto je efektivnější kopírovat obsah struktury `xfi_fibril_context_t`, která zabírá pouze desítky byte.

Samotné přepínání mezi vlákénky je také zcela v režii knihovny `libc`, avšak jeho součástí je vždy systémové volání *sys_tls_set()* nastavující ukazatel *thread-local storage* na hodnotu odpovídající vlákénku, na které je přepnuto. Toho bylo využito k zavolání funkce *sys_xfi_tls_set()*, v níž se nejprve uloží XFI kontext původního vlákénka, zvolí nové vlákénko (identifikované ukazatelem na *thread-local storage*) a provede obnovení jeho XFI kontextu.

Stejný postup uložení a obnovení XFI kontextu vlákének je použit i při přepínání vláken v jádře, protože každému vláknu je přiřazeno právě jedno vlákénko. Technicky je to ošetřeno ve funkcích jádra *before_thread_runs()* a *after_thread_ran()*, které plánovač volá před a po běhu zadaného vlákna.

5.4.3 Správa paměti

Další podporou implementovanou jádrem je poskytování informací o paměťových regionech přístupných procesu, což využívají běhové kontroly strážce paměťových přístupů, které předpokládají dostupnost těchto údajů v instanci struktury `xfi_task_context_t`. Pro aktualizaci této struktury byly v jádře vytvořeny funkce popsané v druhém sloupci tabulky 5.3. Prvním typem jsou funkce s prefixem `xfi_sys_`, jejich účelem je obalení původních systémových volání pracujících s paměťovými oblastmi. Druhým typem jsou funkce s prefixem `xfi_notify_`, které jsou volány z příslušných implementací systémových volání. Oba typy funkcí přitom pracují stejným způsobem, což je zprostředkování získaných informací o změnách přístupných paměťových oblastí do datové struktury `xfi_task_context_t`, kde jsou definovány jak paměťové regiony pro každé přístupové právo (čtení, zápis spuštění), tak i volitelná paměťová bitmapa. Dojde-li tedy např. k vytvoření nové paměťové oblasti procesem pomocí volání *sys_as_area_create()*, obalující funkce *xfi_sys_as_area_create()* propíše informace o této oblasti jak do seznamu paměťových regionů, tak do paměťové bitmapy.

Zavedení obalujících systémových volání bylo motivováno také faktem, že v některých případech potřebuje i samo jádro vytvořit paměťovou oblast v rámci adresového prostoru procesu, která by však neměla být přístupná samotnému procesu – tj. běhová kontrola paměťových přístupů na ní selže. Tedy pokud bude funkce *as_area_create()* volána přímo z funkcí jádra a nikoli z obaleného systémového volání *sys_as_area_create()*, pak vytvořená oblast nebude propsána do datových struktur běhových kontrol. To je momentálně využito při alokaci stínového zásobníku pro každé vlákénko procesu a také pro samotné seznamy paměťových regionů, na něž vede ze struktury `xfi_task_context_t` ukazatel.

5.4.4 Výjimky

Součástí podpory běhových kontrol je také systémové volání *sys_xfi_exception(exc_type, where, arg1, arg2, arg3)* používané k ukončení procesu při neúspěchu běhové kontroly. Jedná se o tzv. XFI výjimku (*XFI exception*) specifikovanou svým typem (argument *exc_type*) a adresou výskytu v kódu (argument *where*) a případně dalšími volitelnými argumenty. Tyto informace jsou v případě jejího výskytu vypsány také na kernel konzoli HelenOS pro snažší diagnostiku.

Seznam stávajících typů všech výjimek je uveden v tabulce 5.4 i s popisem jejich významu. Typ výjimek je specifikován výčtovým typem `xfi_exception_t` definovaným v hlavičkovém souboru `kernel/generic/include/nommu/xfi_context.h`.

5.4.5 Spouštění procesů

Spouštění procesů v rámci uživatelského prostoru zajišťuje serverový proces *loader*, který bylo nutné upravit za účelem korektního spouštění procesů instrumentovaných XFI technikou. Úprava spočívala pouze v přidání systémového volání *sys_xfi_switch_task()* přímo před předání řízení spouštěnému procesu, jehož obraz je načten přímo do adresového prostoru loaderu. Toto systémové volání zajistí zkopírování datové struktury `xfi_context_t` z loaderu do nově spuštěného procesu a provede smazání identifikace aktuálně běžícího vlákénka, jelikož nový proces si po startu vytvoří zcela nové iniciální vlákénko.

5.5 Nedostatky v implementaci XFI

Popsaná implementace běhových kontrol i systémová podpora XFI zahrnuje určité drobné nedostatky, díky nimž tato implementace nemůže zajistit absolutní paměťovou ochranu a softwarovou izolaci ve všech situacích, ve kterých se proces může vyskytnout. Nejedná se však o principielní nedostatky, nýbrž pouze malé nedodělky v samotné implementaci.

5.5.1 Přepínání kontextu vlákének

Kontext vlákének je přepínán v knihovně `libc` funkcemi *context_save()* a *context_restore()*, které jako jediné nejsou předmětem vkládání běhových kontrol. To znamená, že představují potencionální nebezpečí v situaci, kdy by jim jako argument byl předán ukazatel na chybný kontext či ukazatel do paměti nepřístupné procesu. Důvodem vypnutí běhových kontrol je stávající implementace stínového zásobníku, která by rovněž musela počítat s konceptem uložení aktuálního stavu a jeho obnovení. To by bylo možné implementovat doplněním struktury `xfi_fibril_context_t` o položku s poslední hodnotou ukazatele na vrchol stínového zásobníku před uložením kontextu vlákénka. Při obnovení kontextu by vrchol stínového zásobníku byl získán z této položky.

5.5.2 Provedení nepovoleného paměťového přístupu

Při souběhu určitých okolností může ve stávající implementaci dojít k situaci, kdy běhová kontrola strážce paměťových přístupů uspěje a povolí jednomu vláknu procesu zapsat data do určité paměťové oblasti, zatímco druhé vlákno ve stejný okamžik způsobí uvolnění této paměťové oblasti a provedení zápisu bude tudíž chybná operace. To je možné jen při proložení těchto akcí následujícím způsobem:

- vlákno 1: spuštěna běhová kontrola pro ověření zápisu na adresu A, kontrola uspěje,
- přeplánování na vlákno 2,
- vlákno 2: uvolněna paměťová oblast, do níž náleží i adresa A,
- přeplánování na vlákno 1,

Typ výjimky	Popis
DirectControlFlowIntegrityViolation	Porušení integrity toku řízení skokem mimo oblast kódu procesu
IndirectControlFlowIntegrityViolation	Porušení integrity toku řízení nepřímým skokem na nepovolenou instrukci
ShadowStackOverflow	Přetečení stínového zásobníku, obvykle způsobeno příliš hlubokou či nekonečnou rekurzí volání funkcí
ShadowStackUnderflow	Podtečení stínového zásobníku značící, že se proces pokusil provést více návratů z procedury než volání
ShadowStackControlFlowIntegrityViolation	Porušení integrity toku řízení při pokusu o návrat z procedury, obvykle způsobeno poškozením alokačního zásobníku
MemoryDataWriteViolation	Pokus o zápis dat mimo povolené paměťové oblasti
MemoryDataReadViolation	Pokus o čtení dat mimo povolené paměťové oblasti
MemoryCodeExecuteViolation	Pokus o spuštění kódu mimo paměťové oblasti obsahující kód procesu
StackOverflow	Přetečení alokačního zásobníku
StackUnderflow	Podtečení alokačního zásobníku

Tabulka 5.4: Typy XFI výjimek.

- vlákno 1: zápis na adresu A.

Celý problém tkví v tom, že ověření adresy zápisu běhovou kontrolou a samotný zápis do paměti není atomická operace. Možným řešením by bylo docílení atomičnosti vložením zámků, avšak režie na práci se zámky by byla velmi značná. Přijatelnější alternativou by bylo odložené uvolnění paměťové oblasti – tj. i přesto, že proces provedl systémové volání, k samotnému uvolnění paměťové oblasti by došlo až ve chvíli, kdy by všechna ostatní nezablokovaná vlákna procesu běžela alespoň jedno časové kvantum od chvíle systémového volání. Vlákna blokovaná na synchronizačních primitivech není nutné do této podmínky započítávat, protože nemohla být přerušena mezi běhovou kontrolou a zápisem do paměti právě proto, že jsou zablokována jako důsledek systémového volání některého synchronizačního primitiva.

Toto řešení samozřejmě není ideální, protože dovoluje zapsat procesu do paměťové oblasti, která již procesu logicky nenáleží, avšak takový zápis bude zcela neškodný, jelikož paměťová oblast nemohla být přiřazena jinému procesu.

5.6 Srovnání s původní implementací XFI

Jediným zdrojem informací týkající se původní implementace jsou odborné články [9] a [1] popisující XFI a CFI techniky, na jejichž základě probíhal návrh a implementace XFI v této diplomové práci. Zatímco technika CFI odpovídající strážci nepřímých skoků je zde popsána velmi detailně včetně použitých procesorových instrukcí, u ostatních XFI strážců je popsán pouze princip fungování. Z tohoto důvodu nelze stanovit přesné rozdíly mezi implementací uvedenou zde a originální implementací. Následující odstavce stručně shrnou alespoň ty rozdíly vyplývající z obecného popisu.

Strážce zásobníku

Originální implementace zajišťuje ochranu přetečení zásobníku rovněž porovnáním s horní hranicí paměti vymezené pro dané vlákno a zajišťuje mimo to také K dalších bytů dostupných nad aktuálním vrcholem zásobníku, což bylo využito i v této diplomové práci. Její popis však vůbec neuvádí, zda se kontrola vztahuje i na podtečení zásobníku.

Co se týče statické analýzy zjišťující přístupnost procesu do oblasti pod vrcholem zásobníku, ta je v origiální implementaci zobecněna do formy tzv. verifikační stavů popsaných na konci sekce 3.2.4. Tato implementace používá jednodušší přístup a implementuje algoritmus založený na změnách hodnoty `esp` (viz obrázek 5.6), který je však detailně rozepsán.

Strážce paměťových přístupů

Původní implementace poskytovala 2 typy běhových kontrol: *fastpath* a *slowpath*. *fastpath* nalezla využití hlavně tam, kde je předpoklad, že instrumentovaný modul dělá většinu paměťových přístupů do jediné paměťové oblasti, kterou mu poskytl jeho rodičovský proces (v případě, že modul je plugin rodičovského procesu). *slowpath* kontrola se pak využila u ostatních paměťových přístupech. Zde uvedená implementace neklade na paměťové přístupy žádné předpoklady, protože nasazení XFI se provádí na libovolný uživatelský proces. Z tohoto důvodu implementuje jen obdobu *slowpath* kontroly a nově pak kontrolu pomocí paměťové bitmapy, což je dokonce efektivnější ověření paměťového přístupu než původní *fastpath* kontrola.

Co se týče dalších optimalizací, původní implementace těží z přítomnosti verifikačních stavů, a proto nevkládá běhové kontroly tam, kde to není potřeba. Tato implementace nepracuje s verifikačními stavy a využívá pro tyto optimalizace pouze informací strážce zásobníku, což eliminuje pouze část běhových kontrol, které jsou nadbytečné. I přesto je výsledná režie na běh procesů velmi příznivá, jak je popsáno v další kapitole v sekci 6.6.

Strážce přímých a nepřímých skoků

Implementace těchto strážců je v této práci v souladu s původním návrhem XFI a CFI technik.

Stínový zásobník

Původní návrh zmiňuje stínový zásobník pouze v obecné podobě a neuvádí způsob jeho implementace ani kompletní výčet registrů či dalších hodnot na něj ukládaných. Zde popsaná implementace naproti tomu přesně stanoví, jaký je obsah stínového zásobníku a kde dochází ke vložení kódu pro práci s ním.

Systémová podpora

Systémová podpora navržená a implementovaná v této práci je samozřejmě přizpůsobena architektuře systému HelenOS. Původní návrh zahrnoval obecný popis požadavků na tuto podporu – přítomnost správce tabulek s paměťovými přístupy, nutnost alokovat prostor pro stínový zásobník při vytvoření nového vlákna, přechod mezi XFI kódem a rodičovským kódem pomocí tzv. softwarových bran v případě, kdy je XFI modul pluginem v rámci jednoho procesu.

Shrnutí

Ze srovnání plyne, že stávající implementace je jednodušší než její původní návrh a v některých ohledech také odlišná. Zjednodušení však nemá vliv na úroveň dosažené softwarové izolace, nýbrž její efektivity, protože původní návrh zahrnoval řadu optimalizací založených na verifikačních stavech, které zde nebyly implementovány. Je potřeba také zdůraznit, že autoři implementovali XFI pod systémem Windows, kde měli k dispozici nástroj *Vulcan* [30] pro statickou analýzu binárního kódu a binární instrumentaci EXE souborů. V rámci této diplomové práce nebyl žádný takový nástroj pro formát ELF srovnatelné kvality k dispozici, a proto byl implementován vlastní nástroj pojmenovaný STATIF framework (viz příloha D).

Kapitola 6

Vyhodnocení

Tato kapitola je věnována stručnému vyhodnocení implementace, jehož součástí je i změření výkonnosti běhu aplikací v režimu bez MMU jednotky s podporou softwarové izolace.

Vyhodnocení je provedeno podle jednotlivých kritérií stanovených při návrhu v sekci 5.1.1, kterým také odpovídají názvy většiny následujících sekcí.

6.1 Současný stav

Stávající implementace popsaná předchozí kapitolou je spustitelná na architektuře IA-32 v emulátorech QEMU [25] či VMware Player [37] a rovněž přímo na reálném hardware s touto procesorovou architekturou. Pro ověření, zda se zachovaly základní funkce jádra i uživatelského prostoru, byla spuštena sada testů jádra z kernel konzole i sada testů v uživatelském prostoru (viz aplikace `uspace/app/tester`). Testy byly provedeny v režimu bez MMU jednotky s vypnutou i zapnutou podporou softwarové izolace XFI a v obou případech dopadly úspěšně. Rovněž funkcionalita uživatelských aplikací a serverů zůstala zachována, což se ověřilo zejména během vývoje jejich používáním.

6.2 Zachování funkcí MMU

Ideální řešení by zachovalo všechny funkce MMU využívané jádrem HelenOS. To by však znamenalo provádět emulaci MMU jednotky, což by byl možný přístup, ale jistě velmi neefektivní a zřejmě i implementačně poměrně náročný. Tato práce se vydala jiným směrem a zachovává proto pouze některé funkce, u nichž je velmi dobrý předpoklad pro využití i na vestavných systémech s velmi omezenými zdroji.

Jmenovitě se jedná o následující funkce:

- **Ochrana a izolace** procesů je dosažena pomocí sofwarové izolace uživatelských procesů technikou XFI, jejíž granularita ochrany jsou pamětové oblasti stejně jako je tomu v případě přítomnosti MMU. Úroveň ochrany je zde tedy srovnatelná, avšak cena za její dosažení je určitá režie na běh i velikost kódu uživatelských aplikací. Měřením této režie se zabývá sekce 6.6 uvedená dále v této kapitole.
- **Sdílení paměti** je obzvlášť v systému HelenOS velmi důležitá funkce intenzivně využívaná při meziprocesové komunikaci. Bez přítomnosti MMU se jedná o snadnou operaci, která vyžaduje splnění jediné podmínky, aby obě paměťové oblasti, které mají být sdíleny, byly umístěny na stejné fyzické adrese. Díky softwarové izolaci je navíc stále možné stanovit pro obě tyto oblasti rozdílná přístupová práva.

- **Přetečení zásobníku** není v původní verzi HelenOS nijak detekováno zatímco s podporou softwarové izolace je tato detekce automatická, jelikož se jedná o součást integrity toku řízení, která je předmětem běhových kontrol.
- **Detekce nulového ukazatele** je opět docílena prostřednictvím softwarové izolace jako nepřímý důsledek rozvržení zón fyzické paměti na architektuře IA-32. V tomto rozvržení je počáteční oblast paměti přímo v jádře označena jako rezervovaná, a proto na ní nemůže dojít k alokaci paměťové oblasti pro žádný proces. Každý proces s podporou softwarové izolace smí přistupovat (pro zápis a spuštění) pouze do vyjmenovaných paměťových oblastí, a proto je mu takový přístup na adresu nula vždy znemožněn.

Zbývající funkce popsané níže implementovány nebyly a mohly by se proto stát námětem na případné budoucí rozšíření této práce.

- **Mapování a výměna stránek** jsou základními prvky virtuální paměti a bez přítomnosti MMU by bylo komplikované implementovat je ryze softwarovými prostředky.
- **Zamezení fragmentace** dostupné paměti není momentálně nijak zajištěno, a proto dochází jak k interní, tak i externí fragmentaci paměti. Externí fragmentace je důsledkem sdílení jediného adresového prostoru všemi procesy včetně jádra systému, zatímco interní fragmentace je způsobena alokátorem fyzických rámců. Tento alokátor využívá buddy systém dovolující alokovat pouze po sobě jdoucí stránky, jejichž počet je roven mocnině čísla dvě. Např. při alokaci 33 kB paměti dojde k alokaci celých 64 kB při velikosti stránky 4 kB. Řešením je zjevně použití jiného alokátoru rámců, resp. alokátoru fyzické paměti, mluvíme-li o systému bez MMU jednotky.

6.3 Minimální změny zdrojových kódů

Důležitým kritériem při implementaci bylo zachování stávající funkčnosti jádra a uživatelských procesů při minimálním počtu změn v jejich zdrojových kódech. Za účelem vyhodnocení tohoto kritéria byly spočítány řádky, které byly ve zdrojových kódech změněny, připsány nebo přidány do nově vytvořených zdrojových souborů. Do zdrojových souborů byly počítány také hlavičkové soubory a rovněž se započítávaly i komentáře. Přibližné výsledky jsou znázorněny v tabulkách 6.1 a 6.2. Každá tabulka podává samostatný přehled o změnách v jádře systému a prostoru uživatelských aplikací. Navíc každá z těchto oblastí je dále rozdělena na procesorově závislou část (označena jako ia32 kód) a na procesorově nezávislou část (označena jako obecný kód), což umožňuje udělat si dobrou představu o povaze provedených zásahů.

První tabulka 6.1 se věnuje režimu bez MMU jednotky, kde byl proveden relativně malý počet změn v existujících zdrojových souborech, které se týkaly zejména rozšíření parametrů některých funkcí pracujících s paměťovými oblastmi. Nově přidané řádky se pak týkaly práce s paměťovými bloky, které v režimu s MMU jednotkou nebyly potřeba, a také práce s ELF relokacemi při načítání ELF obrazu aplikace v jádře systému či v uživatelském prostoru pomocí serveru *program loader*. Téměř polovina nových či změněných řádků v procesorově nezávislé části uživatelského prostoru se vztahuje k implementaci funkce *malloc()* s podporou alokace paměti z více různých paměťových oblastí současně. Konečně změny procesorově závislého kódu jádra systému se týkaly hlavně vypnutí stránkování a zamezení práce s datovými strukturami pro údržbu stránkovacích tabulek.

Tabulka 6.2 se zabývá implementací softwarové izolace XFI. Zde je zajímavý první řádek tabulky, z něhož je vidět, že nebylo nutné dělat žádné změny v rozhraní stávajících definic funkcí jádra ani uživatelského prostoru a XFI byla tedy implementována jako nepovinné rozšíření celého systému. Rovněž je patrné, že těžiště implementace se nachází v procesorově nezávislé částí jádra, zatímco v uživatelské části se nachází implementaci XFI run-time funkcí v rámci knihovny `libc`.

	Prostor jádra		Uživatelský prostor	
	ia32 kód	obecný kód	ia32 kód	obecný kód
Změněné řádky	0	25	0	180
Nové řádky (stávající soubory)	100	350	4	400
Nové řádky (nové soubory)	160	1 050	100	320

Tabulka 6.1: Počet nových a změněných řádek zdrojového kódu jádra a uživatelských procesů HelenOS v implementaci režimu bez MMU jednotky.

	Prostor jádra		Uživatelský prostor	
	ia32 kód	obecný kód	ia32 kód	obecný kód
Změněné řádky	0	0	0	0
Nové řádky (stávající soubory)	60	400	0	80
Nové řádky (nové soubory)	0	2 170	540	450

Tabulka 6.2: Počet nových a změněných řádek zdrojového kódu jádra a uživatelských procesů HelenOS v implementaci softwarové izolace XFI.

6.4 Transparentnost

Z hlediska aplikačních programátorů i vývojářů jádra bylo transparentnosti dosaženo, a to v obou typech implementace.

Při implementaci režimu bez MMU došlo sice k malým změnám rozhraní funkcí pro práci s paměťovými oblastmi, jednalo se však o velmi malé změny, které byly nutné pro unifikaci přístupu k paměťovým oblastem jak v režimu s MMU jednotkou, tak i bez ní. Navíc, jak je patrno z první řádky tabulky 6.1, počet míst, kde se tyto změny projevily, je poměrně malý – jedná se řádově o desítky řádek ve všech zdrojových souborech.

Transparentnost techniky XFI je téměř stoprocentní, protože její nasazení spočívá pouze v binární instrumentaci a není nutná žádná asistence programátora při tomto procesu na rozdíl od jiných technik softwarové izolace, jako např. při *source-to-source* překladu zdrojových kódů aplikací. Jediným omezením je nutnost poskytovat v binárních souborech pomocné informace a některá omezení na ELF sekce v binárním kódu. Ta jsou však dodržena automaticky díky změnám provedeným v sestavovacích skriptech (linker skript a příslušné `Makefile` soubory).

6.5 Přenositelnost

Ve vývoji HelenOS je kladen velký důraz na přenositelnost, což se snaží respektovat i tato diplomová práce.

Přenositelnost implementace režimu bez MMU jednotky by měla být velmi vysoká, protože úpravy originálních zdrojových kódů se týkaly hlavně procesorově nezávislých částí (viz tabulka 6.1). Změny v kódu architektury IA-32 se vztahovaly k vypnutí stránkování a zpracování ELF relokací, což bylo samozřejmě nezbytně nutné a bude potřeba dělat vždy při přechodu na jinou procesorovou architekturu. Z těchto důvodů by měl být přechod na jinou architekturu (např. ARM) zcela bezproblémový.

Složitější situace je u implementace softwarové izolace XFI. Infrastruktura s podporou běhu instrumentovaných aplikací v jádře systému je sice téměř nezávislá na použitém procesoru, avšak většina run-time funkcí v knihovně `libc` využívaných vkládanými běhovými kontrolami je zcela závislá na konkrétní instrukční sadě. Stejný problém je u nástroje *ia32binrewriter*, jehož běhové kontroly jsou principielně přenositelné na jiné architektury, avšak při samotném přepisování binárního kódu nakonec tento nástroj musí znát dokonale cílovou instrukční sadu. Přenositelnost je tedy pouze částečná, což je však vyváženo kvalitní, efektivní a transparentní softwarovou izolací pro-

cesů. Na druhou stranu v případě přenosu na jinou architekturu není napsání nástroje pro binární instrumentaci XFI tak problematické.

6.6 Efektivita

V této sekci budou prezentovány výsledky z řady měření, jejichž účelem bylo zjistit efektivitu implementace z pohledu režie na běh uživatelských aplikací a na velikost jejich binárního kódu. Pro získání ucelené představy o efektivitě by bylo vhodné změřit také paměťovou náročnost uživatelských aplikací, potažmo i jádra systému. Toto měření by však v současné implementaci bylo zkresleno faktem, že alokátor rámců režimu bez MMU jednotky stále používá buddy systém a tudíž dochází k velké interní fragmentaci alokované paměti, která může činit v nejhorším případě až 50 % z celkové spotřeby paměti. Namísto přesného měření proto bude dále uveden jen odhad spotřeby paměti při předpokladu použití alokátoru rámců s granularitou alokace jedné stránky.

6.6.1 Srovnávací testy

Pro měření bylo vybráno několik srovnávacích testů ze sady SPEC CPU2006 [31] doplněných o testy adaptivní diferenciální pulsní kódové modulace [32], JPEG komprese [15] a výpočtu md5 [26]. Detailnější popis těchto testů je uveden v tabulce 6.3. Důvod, proč byly ze sady SPEC CPU2006 vybrány pouze 3 testy spočívá v jejich snadné přenositelnosti na systém HelenOS bez nutnosti rozšiřování stávající verze knihovny `libc`. I přesto bylo nutné udělat určité změny ve zdrojových kódech všech testů pro jejich úspěšný překlad pro systém HelenOS. Finální verze těchto zdrojových souborů lze nalézt v adresáři `uspace/app/benchmarks`.

Název testu	Stručný popis
adpcm encode	Kodér algoritmu adaptivní diferenciální puslní kódové modulace pro kompresi audio dat
adpcm decode	Dekodér algoritmu adaptivní diferenciální puslní kódové modulace pro kompresi audio dat
bzip2	SPEC CPU2006 – Kompresní algoritmus
jpeg	Komprese obrazu do formátu JPEG
md5	Výpočet md5 hashe
mcf	SPEC CPU2006 – Kombinatorická optimalizace plánovaní pohybu vozidel v městské hromadné dopravě
sjeng	SPEC CPU2006 – Algoritmy umělé inteligence hrající šachovou partii

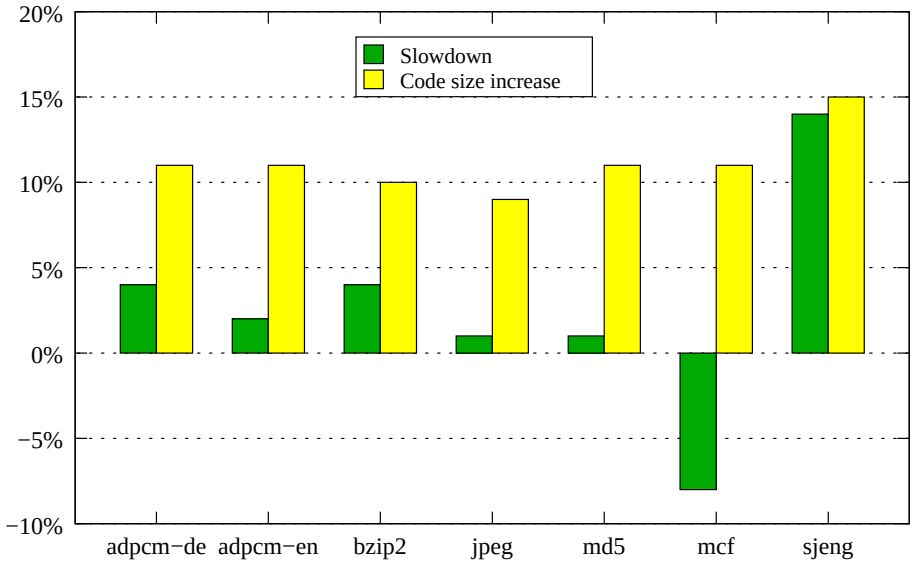
Tabulka 6.3: Srovnávací testy použité pro měření efektivity implementace softwarové izolace a režimu bez MMU jednotky.

Měření se provádělo na stroji s procesorem Intel Pentium 4 o výkonu 2,4 GHz a pamětí 1,5 GB RAM. Předmětem měření bylo 6 různých konfigurací se všemi zmíněnými srovnávacími testy. Přehled konfigurací je uveden v tabulce 6.4. Jednotlivé konfigurace se odlišovaly v režimu běhu systému HelenOS – tj. zda se jednalo o režim s MMU jednotkou či bez ní a nastavením softwarové izolace XFI, kde byly postupně zapínáni strážce zásobníku (*stack guard*), strážce integrity toku řízení (*CFI guard*) skládajícího se ze strážce (ne)přímých skoků a stínového zásobníku, strážce paměťových přístupů (*memory guard*). U strážce paměťových přístupů se v posledních 2 konfiguracích lišilo nastavení, zda mají být použity klasické běhové kontroly procházející seznam paměťových oblastí anebo také běhové kontroly využívající pomocnou paměťovou bitmapu. V obou případech běhové kontroly testovaly pouze přístupy pro zápis do paměti a spuštění, přístupy pro čtení testovány nebyly.

Každá z konfigurací byla součástí speciálního sestavení HelenOS, ve kterém se při startu spouštěly pouze procesy nutné k chodu systému a testů. Při měření běžel každý test celkově nejméně

Konfigurace	Režim	Stack guard	CFI guard	Memory guard
1	MMU	–	–	–
2	bez MMU	–	–	–
3	bez MMU	Ano	–	–
4	bez MMU	Ano	Ano	–
5	bez MMU	Ano	Ano	Ano
6	bez MMU	Ano	Ano	Ano

Tabulka 6.4: Konfigurace měření.



Obrázek 6.1: Porovnání režimu bez MMU a režimu s MMU jednotkou.

10 vteřin, přičemž výsledná délka jeho běhu se měřila v počtech tiků procesoru, které test strávil v uživatelském prostoru. Každý test byl spuštěn 5krát a výsledný počet tiků jeho běhu byl získán jako aritmetický průměr těchto 5 měření. Směrodatná odchylka chyby při měření byla obvykle menší než 1 %, avšak v některých případech mírně překročila hodnotu 3 %.

ISO obrazy spustitelného HelenOS s jednotlivými konfiguracemi testů i výsledky testů jsou k dispozici na přiloženém CD disku v adresáři `Benchmarks`.

6.6.2 Režim bez MMU

Výsledky testů porovnávajícího efektivitu režimu bez MMU jednotky s režimem s MMU se nachází na obrázku 6.1. Levý sloupec u každého testu znázorňuje zpomalení aplikace v porovnání běhu v režimu s MMU zatímco pravý sloupec reprezentuje nárůst velikosti binárního kódu. Skutečnost, že aplikace v režimu bez MMU jsou pomalejší a zabírají více procesorových instrukcí je důsledek použití pozičně nezávislého kódu (tzv. *PIC* kód), pro něhož nejsou instrukce architektury IA-32 přizpůsobeny, jelikož není možné provádět adresování relativně k hodnotě registru `eip`. Z tohoto důvodu jsou přístupy ke globálním proměnným pomalejší, protože se musí zjistit aktuální hodnota registru `eip` a přistoupit do *GOT* tabulky, kde je teprve uložena adresa globální proměnné. Tento problém se týká i veškerých globálních dat používaných v aplikaci, tzn. jak polí s konstantami tak i pomocných tabulek generovaných překladačem. Jeden z testů se však této úvaze vymyká, a to test `mcf`, u kterého se při výpočtech používají globální proměnné jen velmi zřídka, jak se ověřilo průchodem zdrojových souborů, a celkový čas potřebný k běhu testu byl dokonce o 8 % nižší než u konfigurace s MMU jednotkou. Naopak výsledky testu `sjeng` jen potvrzují, že zpomalení souvisí

s používáním globálních proměnných, protože ve zdrojovém kódu se používá velké množství globálně definovaných proměnných a tabulek, což má za následek 14% zpomalení.

Co se týče paměťové spotřeby, ta by měla být teoreticky nižší než při přítomnosti MMU jednotky, protože není potřeba alokovat paměť pro stránkovací tabulky a při implementaci režimu bez MMU nebyly použity žádné nové pomocné datové struktury, nýbrž došlo pouze k malému rozšíření datové struktury popisující paměťovou oblast. V současné verzi je však paměťová spotřeba v nejhorším případě až dvojnásobně vyšší kvůli použití buddy alokátoru pro alokaci rámců fyzické paměti. Při použití alokátoru s granularitou jednoho rámce by celková spotřeba paměti systému byla měřitelná i v provedených srovnávacích testech.

6.6.3 Softwarová izolace s XFI

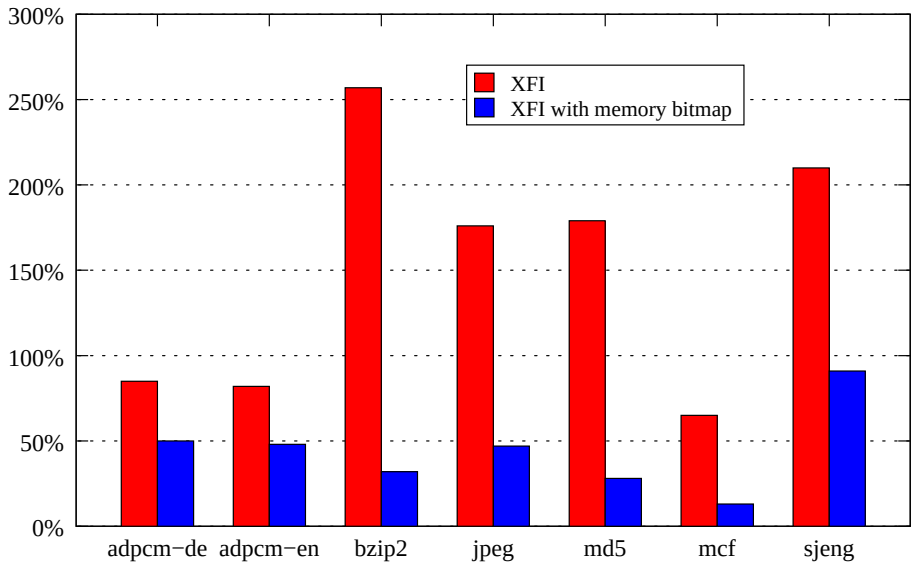
Na grafech 6.2 a 6.3 jsou zobrazeny režie testů instrumentovaných XFI technikou v režimu bez MMU vzhledem k testům bez této instrumentace. První graf se týká běhové režie, druhý pak nárůstu velikosti binárního kódu testu. V obou měřeních byly v XFI technice nasazeny běhové kontroly všech strážců. V levých sloupcích (červená barva) byly použity u paměťového strážce klasické běhové kontroly procházející seznam paměťových oblastí (první varianta XFI), v pravých sloupcích (modrá barva) byly použity u těchto strážců přednostně paměťové bitmapy (druhá varianta XFI). Režie je vyjádřena v procentech relativně k neinstrumentované verzi testu, takže např. hodnota 50 % u testu `adpcm_de` v prvním grafu značí 50procentní zpomalení oproti její původní verzi.

Z prvního grafu je zřejmé, že běhově efektivnější instrumentaci poskytuje XFI ve variantě s paměťovou bitmapou – rozdíl je v některých případech až několikanásobný, např. režie testu `bzip2` bez paměťové bitmapy činí 257 %, zatímco s paměťovou bitmapou pouze 32 %. V případě nárůstu binárního kódu po instrumentaci je situace opačná, i když rozdíl není tak markantní a činí nejvýše 25 %. Příčinou je rozdílnost používaných běhových kontrol paměťového strážce. Při použití klasické běhové kontroly je každý paměťový přístup (přesněji zápis či spuštění) ověřován zavoláním podprocedury, která provádí průchod seznamem dostupných paměťových oblastí. Volání procedury a průchod smyčkou mohou zabrat až desítky instrukcí, než je kontrola úspěšně dokončena. Naproti tomu při použití běhových kontrol využívajících paměťové bitmapy stačí řádově jednotky instrukcí pro získání indexu do paměťové bitmapy a otestování nenulovosti bitu přístupových práv. Cenou za tento přístup je větší počet instrukcí na místě dotyčného paměťového přístupu.

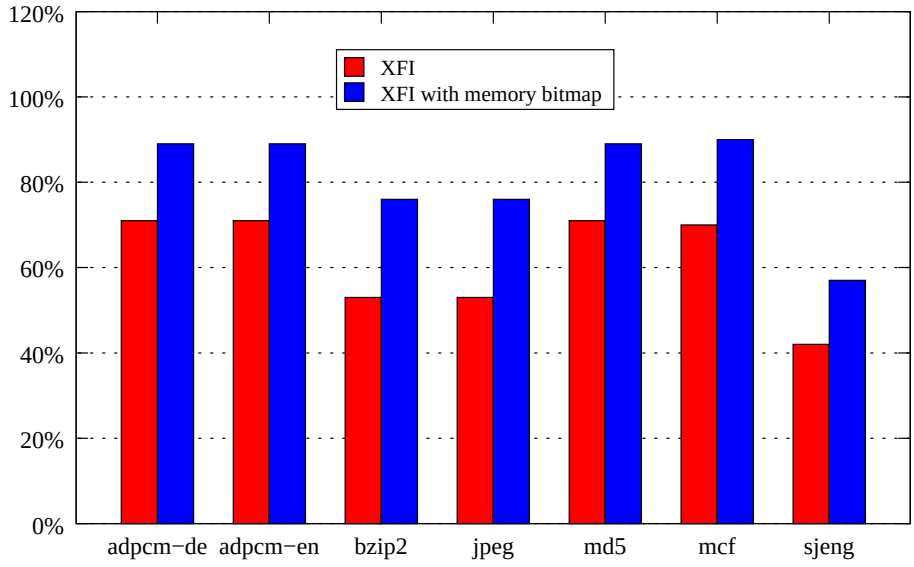
Nejnižší běhovou režii zaznamenal test `mcf` o velikosti 13 %, naopak nejvyšší měl již zmiňovaný `bzip2` bez paměťové bitmapy, která činila 257 %. V průměru se jedná o režii 151 % bez paměťové bitmapy a 44 % s použitím paměťové bitmapy. U nárůstu binárního kódu jsou průměrné hodnoty 62 % resp. 81 % pro každou variantu.

Zajímavé informace podávají také tabulky 6.5 a 6.6, které zobrazují podíl běhových kontrol každého strážce na celkové režii běhu resp. nárůstu binárního kódu. Tabulky jsou rozděleny do dvou částí, v horní jsou uvedeny informace pro režim s klasickým paměťovým strážcem, zatímco v dolní části s implementací paměťovou bitmapou. Podle těchto informací lze vyčíst povahu jednotlivých testů – např. `bzip2`, `md5` a `mcf` soustředí své výpočty pravděpodobně dovnitř dlouhých smyček, kde se provádí intenzivní zápisy dat a minimální počet volání podprocedur. Vypovídá o tom skutečnost, že největší podíl na běhové režii má paměťový strážce, přičemž běhové kontroly dalších dvou strážců, které se provádí typicky při vstupu do podprocedury, se vykonávají velmi zřídka. Naproti tomu testy `adpcm_decode`, `adpcm_encode` a `jpeg` vykazují poměrně rovnoměrné rozložení režie, což naznačuje, že výpočet může být strukturován do více nezávislých funkcí, které se navzájem často volají. Při provedení intenzivnějších optimalizací běhových kontrol paměťového strážce by tedy bylo možné značně snížit běhovou režii u testů, jejichž režie je z velké části tvořena právě těmito kontrolami.

Implementace XFI se rovněž neobejde bez určité paměťové režie. Podpora XFI v jádře i uživatelském prostoru systému potřebuje pro udržení seznamu přístupných paměťových oblastí pro každý typ přístupu (čtení, zápis, spuštění) jeden alokovaný rámec, dále jeden rámec pro stínový zásobník každého vlákénka a rovněž datové struktury v řádech desítek byte pro každé vlákénko. Paměťová



Obrázek 6.2: Zpomalení běhu aplikací instrumentovaných technikou XFI.



Obrázek 6.3: Zvětšení kódu aplikací instrumentovaných technikou XFI.

spotřeba je tedy přímo úměrná počtu vytvořených vlákének. V případě použití paměťového strážce s paměťovou bitmapou je navíc nutná alokace této bitmapy, jejíž velikost je úměrná dostupné fyzické paměti nebo může být i vyšší pokud má bitmapa pokrývat i např. paměťově mapovaná zařízení umístěná na adresách nad koncem fyzické paměti. Budeme-li uvažovat bitmapu pokrývající pouze fyzickou paměť, pak tato bitmapa bude zabírat $N/page\_size$ byte, kde N je celková velikost fyzické paměti a $page\_size$ velikost stránky, oba údaje vyjádřeny v byte.

6.7 Závěr

Lze konstatovat, že podle provedeného vyhodnocení splnila implementace režimu bez MMU a softwarové izolace XFI kritéria, která byla stanovena při návrhu. Implementace zachovává řadu funkcí nabízených hardwarovou MMU jednotkou pomocí běhových kontrol XFI techniky. Provedené změny v jádře i uživatelských procesech přitom nebyly zásadního charakteru a většinou byly do stávajícího kódu vloženy v rámci bloků podmíněného překladu a jen velmi malá skupina změn byla provedena mimo tyto bloky. Celá implementace je navíc transparentní jak pro vývojáře jádra (díky blokům podmíněného překladu), tak pro vývojáře aplikací uživatelského prostoru, kde bylo provedeno jen velmi málo změn. Implementace režimu bez MMU by měla být rovněž velmi dobře přenositelná s tím, že u softwarové izolace technikou XFI je přenositelnost pouze částečná, jelikož je použita technika binární instrumentace. Poslední vyhodnocení se týkalo efektivity, kde měření ukázala, že jak režie na velikost kódu, tak režie na běh aplikací je akceptovatelná.

	Stack guard	CFI guard	Memory guard
XFI			
adpcm_de	20 %	28 %	52 %
adpcm_en	24 %	27 %	49 %
bzip2	0 %	1 %	99 %
jpeg	8 %	9 %	83 %
md5	0 %	1 %	99 %
mcf	0 %	3 %	97 %
sjeng	7 %	14 %	79 %
XFI with memory bitmap			
adpcm_de	35 %	47 %	18 %
adpcm_en	41 %	47 %	12 %
bzip2	0 %	5 %	95 %
jpeg	32 %	32 %	36 %
md5	-2 %	8 %	95 %
mcf	-1 %	14 %	87 %
sjeng	16 %	33 %	51 %

Tabulka 6.5: Podíl jednotlivých strážců na režii běhu aplikací instrumentovaných XFI technikou.

	Stack guard	CFI guard	Memory guard
XFI			
adpcm_de	55 %	23 %	22 %
adpcm_en	55 %	23 %	22 %
bzip2	44 %	20 %	36 %
jpeg	35 %	30 %	35 %
md5	55 %	23 %	22 %
mcf	54 %	22 %	24 %
sjeng	48 %	24 %	28 %
XFI with memory bitmap			
adpcm_de	44 %	18 %	38 %
adpcm_en	44 %	18 %	38 %
bzip2	31 %	14 %	56 %
jpeg	24 %	21 %	55 %
md5	44 %	18 %	38 %
mcf	41 %	17 %	42 %
sjeng	36 %	17 %	47 %

Tabulka 6.6: Podíl jednotlivých strážců na zvětšení binárního kódu aplikací instrumentovaných XFI technikou.

Kapitola 7

Možnosti rozšíření

V této kapitole budou popsány některé návrhy či doporučení, jak by se dala současná implementace režimu bez MMU jednotky a softwarové izolace rozšířit či zdokonalit, aby se dosáhlo lepší praktické použitelnosti. Některé z těchto návrhů byly původně zamýšleny jako součást této diplomové práce, avšak z časových důvodů se nepodařilo je implementovat.

7.1 Režim bez MMU

7.1.1 Alokátor fyzické paměti

Stávající implementace používá pro alokaci rámců fyzické paměti buddy alokátor, který je používán i původní verzí HelenOS s podporou MMU jednotky. Tento alokátor sice dokáže alokovat dostupnou paměť velmi efektivně, avšak počet alokovaných rámců musí být vždy roven mocnině čísla 2. V režimu s MMU jednotkou toto omezení nehraje příliš velkou roli, protože alokace pro uživatelské procesy se provádí pouze po jediném rámci díky tomu, že stránky z virtuálního adresového prostoru procesu jsou mapovány na rámce až když proces na tuto stránku provede přístup. V režimu bez MMU jednotky je nutné již při alokaci paměti procesem provést vyhrazení fyzické paměti zavoláním alokátoru rámců.

Možným řešením je výměna alokátoru rámců za nový alokátor, který dokáže alokovat paměť o velikosti libovolného počtu po sobě jdoucích rámců. Tento přístup zvolili také např. autoři uClinuxu [36], popsaného v následující kapitole, kde se potýkali s naprosto stejným problémem. Nový alokátor rámců by mohl, podobně jako buddy alokátor, pracovat nad polem struktur `frame_t` popisujících jednotlivé rámce, kde by si mohl vést pomocné informace o alokovaných blocích fyzické paměti.

7.1.2 Provázané zásobníky

Nepříjemným omezením uživatelských aplikací v HelenOS je nutnost znát velikost zásobníku, která jednotlivá vlákénka potřebují ke své činnosti. Vzhledem k tomu, že se jedná o objekty spravované knihovnou `libc`, nemohou využít techniky stránkování k dynamické změně jejich velikosti v případě potřeby. Využít tuto techniku mohou pouze vlákna spravovaná jádrem, v nichž neběží žádné vlákénko. Bez MMU není k dispozici ani tato možnost.

Řešením může být technika provázaných zásobníků (*linked stacks*) použitá např. v operačním systému Singularity [13] popsaného v následující kapitole. Principem je dynamická alokace nových částí zásobníku, které nemusí navazovat na aktuální vrchol zásobníku, nýbrž mohou být alokovány na jiných (fyzických) adresách. V kódu aplikace je pak nutné zajistit transparentní přechod mezi těmtito částmi zásobníku a rovněž zajistit jejich uvolňování v případě, kdy již nebudou potřeba. V systému Singularity řeší tuto problematiku překladač, který po statické analýze rozhodne, před která volání funkcí je vhodné umístit kontroly na přetečení zásobníku, jejichž důsledkem je alokace

nové části v případě potřeby. V systému HelenOS by pro tyto účely mohla posloužit statická analýza binárního kódu a binární instrumentace, která by zajistila vložení potřebných kontrol přetečení a alokace nové části zásobníku.

7.1.3 Optimalizace alokátoru paměti v uživatelském prostoru

Původní i současná verze funkce *free()* v knihovně `libc` uvolňuje alokované bloky paměti a umožňuje uživatelské aplikaci znovu takto uvolněnou paměť použít, ale již neprovádí uvolnění paměťových oblastí, na kterých se alokace bloků provádí. Tedy z pohledu jádra systému a ostatních uživatelských aplikací je tato paměť stále alokována pro tuto aplikaci, i když ji nevyužívá.

7.1.4 Paměťová optimalizace spouštění procesů

Spouštění uživatelských procesů zajišťuje proces *program loader*, který nejprve do svého adresového prostoru načte obraz nově spouštěného procesu a poté provede skok na jeho vstupní bod. Samotný obraz procesu *program loader* je přitom ve fyzické paměti umístěn pouze v jedné instanci, jelikož se o mapování stránek jeho obrazu do fyzické paměti stará ELF backend. Bez MMU jednotky dochází k jeho načtení do fyzické paměti vždy znovu s každým novým spouštěným procesem, ale již nedochází k uvolnění jeho obrazu, když je řízení předáno spouštěnému procesu. K uvolnění dochází až při ukončení spouštěného procesu.

Optimalizace by zde spočívala v uvolnění obrazu *program loaderu* ještě před předáním řízení cílovému procesu, což lze udělat ve dvou krocích. V prvním kroku se provede alokace pomocné paměťové oblasti, do níž se umístí kód provádějící uvolnění ELF obrazu a následný skok na cílový proces. V druhém kroku se provede skok do této pomocné oblasti. Paměťová režie se tak sníží pouze na velikost této pomocné oblasti, což může být pouze jedna stránka.

7.1.5 Dynamicky linkované sdílené knihovny

Sdílené knihovny jsou velmi efektivním prostředkem sdílení kódu i snížení paměťové spotřeby, což může být obzvlášť výhodné na vestavných systémech s malou paměťovou kapacitou. Naneštěstí sdílení knihoven je běžně možné jen za přítomnosti MMU jednotky, jelikož se při něm plně využívá mechanimus *copy-on-write*. Bez podpory MMU jednotky lze sdílení knihoven také docílit, obvykle je k tomu potřeba upravit generování binárního kódu přímo v překladači. Inspirací pro implementaci sdílených knihoven v režimu bez MMU v HelenOS může být uClinux [36] nebo např. odborný článek [24] detailně popisující implementaci sdílených knihoven na uClinuxu s procesorem ARM.

7.2 Softwarová izolace s XFI

7.2.1 Nástroj ia32binrewriter

Nástroj *ia32binrewriter* i STATIF framework provádějící binární instrumentaci by se mohl stát předmětem řady vylepšení.

Doposud je nutné, aby instrumentovaný binární kód ve formátu ELF zahrnoval tabulku symbolů obsahující všechna návěští použitá v kódu, což STATIF framework využívá pro rozpoznání začátku všech procedur. I bez tabulky symbolů by mělo být možné určit hranice procedur pomocí kvalitní statické analýzy binárního kódu (na základě toku řízení základních bloků). Rovněž cíle nepřímých skoků lze rozpoznat analýzou relokační tabulky, která musí být u pozičně nezávislého kódu vždy přítomna.

Další možné vylepšení spočívá v přímém vkládání (přilinkování) kódu pro podporu běhových kontrol. Tento kód je v současné chvíli součástí knihovny `libc` a uživatel může snadno změnit jeho podobu editací příslušných zdrojových souborů. Pokud by se tento kód vkládal až nástrojem *ia32binrewriter*, byl by to krok k většímu zabezpečení a spolehlivosti provozování XFI implementace.

V současné chvíli však STATIF framework nemá dostatek prostředků pro vložení sekcí ELF souboru do jiného ELF souboru.

Přínosem by rovněž bylo přesunout běh nástroje *ia32binrewriter* z fáze překladu do fáze spouštění procesu. To by znamenalo zprovoznění tohoto nástroje přímo pod systémem HelenOS a jeho spouštění procesem *program loader*, což by vyžadovalo portaci C++ kódu do prostředí systému HelenOS spolu s knihovnou `libelf`, která je tímto nástrojem používaná pro editaci ELF souboru.

7.2.2 Ověření spouštěného XFI procesu

I v případě, kdy by bylo možné provádět binární instrumentaci přímo před samotným spuštěním procesu v systému HelenOS, stále je výhodné mít možnost provádět instrumentaci v čase překladu zejména z důvodů její časové náročnosti, která se pohybuje v řádech sekund. Aby měl systém jistotu, že do binárního kódu procesu se od provedení instrumentace neprováděly žádné zásahy ovlivňující správnou činnost běhových kontrol, je nutné ověřit jeho důvěryhodnost. Existují minimálně dva používané způsoby, jak tuto důvěryhodnost ověřit.

První způsob je používán přímo původní implementací techniky XFI a je založen na statické analýze binárního kódu. Tato analýza prověří, zda běhové kontroly jsou na všech potřebných místech. Druhá možnost používaná např. v technice Harbor spočívá v nasazení digitálního podpisu. Po dokončení binární instrumentace je celý binární kód digitálně podepsán a před jeho spuštěním podpis ověřen. Oba způsoby by bylo možné implementovat i v systému HelenOS.

7.2.3 Optimalizace běhových kontrol

Značný prostor pro optimalizace skýtají běhové kontroly XFI techniky popsané v kapitole věnované návrhu a implementaci. Jak naznačily výsledky srovnávacích testů v sekci 6.6, až v polovině případů převažuje režie běhových kontrol paměťového strážce, kde by bylo možné provést několik významných optimalizací.

V současném stavu se při statické analýze nezohledňuje tok řízení, díky kterému by bylo možné odstranit duplicitní běhové kontroly. Pokud např. proces zapisuje dvakrát po sobě do stejné proměnné s mírnými rozestupy těchto zápisů, pak za určitých okolností je možné odstranit běhovou kontrolu u druhého zápisu. Statickou analýzou toku řízení by také bylo možné zjistit, zda se po sobě prováděné zápisy netýkají stále stejného objektu (např. proměnné datového typu `struct` v jazyce C) a pokud ano, pak by stačilo vložit běhovou kontrolu ověřující přístup do intervalu adres (začátek a konec struktury) namísto samostatných běhových kontrol před každým zápisem. Tato optimalizace by se uplatila v řadě případů, kdy dochází k inicializacím či kopírování datových struktur. Další vylepšení založené na statické analýze toku instrukcí by spočívalo v předřazení běhové kontroly paměťového zápisu před cyklus, v němž adresa zápisu závisí na iterační proměnné cyklu. Běhová kontrola by provedla ověření intervalu adres, přičemž začátek a konec tohoto intervalu by závisel na iniciální hodnotě iterační proměnné a její maximální hodnotě.

7.2.4 Podpora sdílených knihoven

Sdílené knihovny se principem víceméně neliší od standardních spustitelných aplikací z hlediska binárního kódu ve formátu ELF. Jejich přepis by tudíž zajišťoval nástroj *ia32binrewriter* a běhová podpora XFI v jádře a uživatelském prostoru systému HelenOS by rovněž byla aplikovatelná tak jako u stávajících procesů linkovaných pouze staticky.

7.2.5 Podpora SMP

Symetric multiprocessing (*SMP*) je v systému HelenOS podporován, a proto by byla vhodná i jeho podpora v rámci XFI techniky. V současné implementaci je samozřejmě podporováno vícevláknové zpracování založené na předpokladu, že v daný okamžik může běžet pouze jediné vlákénko procesu.

Tomuto předpokladu odpovídá i fakt, že datová struktura `xfi_fibril_context_t` používaná XFI běhovými kontrolami je umístěna na pevné fyzické adrese a je měněna jádrem při každém přepnutí na jiné vlákénko. Běhové kontroly přistupují k této struktuře přes registr `fs`, který je momentálně pevně nastaven při startu procesu. Při podpoře SMP by tento registr mohl být měněn v závislosti na tom, na kterém procesoru k přepnutí vlákénka došlo, takže daný proces by současně mohl běžet na všech dostupných procesorech za logického předpokladu, že každé vlákénko běží maximálně na jediném z nich.

Kapitola 8

Podobné projekty

V této kapitole bude uveden stručný přehled některých jiných operačních systémů, které řeší podobnou problematiku jako tato diplomová práce. Z dostupných řešení byl vybrán pouze malý vzorek, jelikož projektů, které se jistým způsobem zabývají paměťovou ochranou v operačních systémech, ať už na vestavných zařízeních nebo desktopových systémech, je velká škála. Podrobnější přehled těchto projektů lze najít např. v odborných článcích, které popisují níže zmíněné projekty.

8.1 uClinux

uClinux [36] [22] je populární derivát Linuxu adaptovaný na běh na mikroprocesorech postrádajících MMU jednotku jako je např. Coldfire, Blackfin nebo ARM. Podobně jako v případě této diplomové práce jeho autoři museli čelit problémům spojeným s nepřítomností MMU jednotky, na kterou se řada funkcí původního jádra Linuxu i uživatelských procesů spoléhá.

uClinux umisťuje všechny uživatelské procesy do společného adresového prostoru, z čehož plynou nevýhody jako je nemožnost zvětšit paměť procesu za jeho běhu, na což jsou linuxové procesy za normálních okolností zvyklé, a hlavně vzájemná ztráta ochrany jádra a uživatelských procesů. V uClinuxu tedy může každý proces číst a zapisovat na libovolné adresy fyzické paměti, pokud není k dispozici alespoň jednoduchá hardwarová ochrana na bázi regionů.

Samotné jádro se od původní verze Linuxu výrazně neliší, jisté rozdíly zde však jsou. Např. podpora paměťově mapovaných souborů systémovým voláním *mmap()* byla velmi omezena – lze ji použít pouze v režimu pro čtení a pouze na souborech ze souborového systému, který ukládá data souborů sekvenčně, čemuž vyhovuje např. souborový systém *romfs*. Také alokace fyzických rámců byla změněna, resp. rozšířena o nový alokátor nazvaný *kmalloc2*, zabraňující interní fragmentaci tím, že dovoluje alokovat fyzickou paměť s granularitou 4 kB narozdíl od klasického alokátoru, který používá z důvodu efektivity buddy systém.

Další významnou změnou v jádře je nedostupnost standardních formátů spustitelných souborů jako je např. ELF formát. Namísto toho je k dispozici pouze tzv. plochý formát (*flat format*) umožňující uchovávat spustitelný kód ve dvou variantách: s plnou relokační informací nebo ve variantě pozičně nezávislého kódu (PIC kód) s malým počtem relokací týkajících se pouze dat. Výhodou je také možnost využití tzv. XIP (*execute-in-place*) režimu, kdy je proces spuštěn přímo z flash nebo ROM paměti. S XIP režimem také úzce souvisí problematika sdílených knihoven, které jsou opět podporovány pouze ve *flat* formátu. Aby bylo docíleno skutečného sdílení kódu knihovny, musí být přeložena do pozičně nezávislého kódu a rovněž využívat XIP režim.

Dalším omezením v uClinuxu je způsob vytváření nových procesů. Ve světě Linuxu je běžné, že rodičovský proces vytváří syna naklonováním sama sebe přes systémové volání *fork()*. Synovský proces poté sdílí s rodičovským všechny otevřené soubory a další vlastnosti. Jedinou výjimkou je zásobník, na kterém nový proces pracuje. Klonování je umožněno technikou *copy-on-write* navázanou na stránkování, které není v uClinuxu k dispozici. Místo toho nabízí uClinux systémové volání

vfork(), které pozastaví běh rodičovského procesu do chvíle, než synovský proces zavolá funkci *exec()* startující nový proces a klonování tudíž není potřeba.

8.2 Singularity

Singularity [13] je experimentální operační systém vyvinutý v Microsoft Research. Jedná se o systém napsaný kompletně od základů s použitím programovacího jazyka Sing#, což je rozšíření dobře známého jazyka C#. Pro nízkoúrovňové funkce jsou použity také jazyky C nebo přímo assembler, avšak podle autorů je více než 90 % kódu napsáno pouze v Sing#.

Singularity je postaven na konceptu tzv. softwarově izolovaných procesů (*Software Isolated Process*) nebo-li SIP, jejichž kód je generován překladačem zajišťujícím vkládání běhových kontrol pro zajištění paměťové ochrany a izolace. Režie těchto kontrol je však velmi malá díky tomu, že Sing# je moderní, silně typovaný jazyk, kde se nepoužívají přímé paměťové přístupy jako např. v jazyce C, nýbrž je založen na objektovém přístupu s podporou garbage collectoru. Díky softwarové izolaci umožňuje Singularity běh více SIP procesů v rámci jediného adresového prostoru i bez MMU jednotky při zachování maximální úrovně ochrany a vzájemné izolace. Izolace je zajištěna faktem, že SIP proces smí přistupovat pouze do své vlastní haldy nebo do sdílené haldy nazývané *výměnná halda* (*exchange heap*), která se používá při vzájemné komunikaci SIP procesů.

SIP proces může komunikovat buď s jádrem systému přes ABI (*Application Binary Interface*) rozhraní poskytující přes 100 různých funkcí nebo s jiným SIP procesem přes komunikační kanál založený na předem daném kontraktu, což je formální definice komunikačního rozhraní pomocí sady funkcí a komunikačních stavů. Samotná komunikace se provádí zasíláním a příjmem zpráv obsahujících data, v nichž mohou být přítomny i ukazatele směřující pouze do výše zmiňované výměnné haldy.

V základní konfiguraci systém Singularity umisťuje jádro, uživatelské SIP procesy i výměnnou haldu do jediného adresového prostoru. Pokud je však přítomná MMU jednotka, Singularity umí využít i její funkce hardwarové ochrany a za tímto účelem zavádí tzv. hardwarově izolované procesy nebo-li HIP (*Hardware Isolated Process*) a princip domén ochrany. HIP je tedy proces, který má svou samostatnou doménu ochrany – samostatný virtuální adresový prostor. Hardwarovou a softwarovou izolaci lze přitom kombinovat, takže např. více SIP procesů lze umístit do společné domény ochrany, kde bude také umístěna jejich společná výměnná halda. Hardwarová ochrana je v systému Singularity implementována spíše jako dodatečná funkcionalita, která autorům umožnila provést řadu měření za účelem porovnání efektivity softwarové a hardwarové ochrany. Dosažené výsledky jasně hovoří ve prospěch softwarové ochrany, která byla v Singularity až několikanásobně efektivnější.

8.3 Mondrix

Mondrix je upravená verze linuxového jádra s vestavěnou podporou Mondrianské ochrany paměti [40] popsanou v sekci 3.1.2. Jedná se o kombinaci softwarové a hardwarové ochrany za účelem vzájemné izolace modulů linuxového jádra tak, aby jejich chybná funkčnost nenarušila běh jádra ani ostatních modulů. Pro izolaci uživatelských procesů se přitom stále využívá hardwarové ochrany nabízené MMU jednotkou.

8.4 VINO

VINO [28] je experimentální operační systém vyvinutý na Harvardské univerzitě v rámci disertační práce týkající se výzkumu v oblasti rozšiřitelných operačních systémů. Jeho hlavní myšlenkou je umožnit uživatelským procesům resp. modulům rozšiřujícím jádro převzít kontrolu nad řízením různých strategií jádra – např. jaká oblast fyzické paměti bude procesu namapována do virtuálního

adresového prostoru. Převzetí této kontroly však musí být provedeno bezpečným způsobem, aby nedošlo k narušení funkce jádra a rovněž cena za tuto bezpečnost musí být přiměřeně nízká.

VINO používá pro dosažení těchto cílů vlastní softwarové izolace nazývané *MiSFIT* [27] založené na technice SFI [38] a rozšířené o podporu na běh na procesorech řady x86. Pro zajištění paměťové ochrany se používá SFI technika *sandboxing*, avšak pro zachování integrity toku řízení navrhuje MiSFIT vlastní prostředky: tabulku povolených návěští, na které je možné předat z kódu řízení z nepřímého skoku či volání a dále pomocný zásobník pro uchování návratových adres procedur podobně jako u XFI techniky.

Kapitola 9

Závěr

Předmětem této práce bylo rozšíření operačního systému HelenOS umožňující běh na procesorech bez MMU jednotky. Značná část práce byla věnována přehledu technik používaných v současné době pro nahrazení části funkcí MMU jinými prostředky; tyto techniky byly poté vyhodnoceny a srovnány. Na základě tohoto srovnání byla zvolena technika XFI [9] založená na instrumentaci binárního kódu, která poskytuje potřebnou paměťovou ochranu a izolaci současně běžících procesů i bez přítomnosti MMU pomocí tzv. softwarové izolace. Implementace této techniky v kontextu systému HelenOS byla podrobně popsána a vyhodnocena také spolu s dalšími úpravami nezbytnými pro chod systému HelenOS v prostředí bez MMU jednotky. Na závěr bylo uvedeno několik jiných operačních systémů běžících na procesorech bez této jednotky a způsob, jakým byla řešena náhrada funkcí této jednotky v jejich případě.

9.1 Splnění cílů

Lze konstatovat, že cíle stanovené v sekci 1.2 byly splněny, což také dokládá funkční prototypová implementace běžící na procesoru IA-32 s vypnutou podporou stránkování, což téměř věrně simuluje prostředí bez MMU jednotky. Nahrazení paměťové ochrany a izolace procesů je zajištěno softwarově technikou XFI, takže běžící procesy i přes skutečnost, že jsou umístěny ve společném adresovém prostoru, nemají o sobě žádné tušení a nemohou se ani navzájem ovlivňovat. Jak dokládají výsledky z kapitoly 6, implementované řešení mění rozhraní a datové struktury jádra systému pouze v malé míře a většina úprav byla dodána ve formě volitelné a rozšiřující funkcionality, což velmi snižuje pravděpodobnost zanesení chyb do původních zdrojových souborů. Díky malému počtu úprav a použití binární instrumentace na výsledný binární kód aplikací je výsledné řešení také transparentní jak pro samotné aplikace, tak i vývojáře, kteří stále mohou pro svou práci předpokládat existenci samostatného virtuálního prostoru a izolaci více běžících procesů jako při přítomnosti MMU jednotky. Podařilo se splnit i poslední cíl týkající se efektivity navrženého řešení, jelikož výsledná režie na běh procesů, u nichž je XFI nasazena, dosahuje podle srovnávacích testů v kapitole 6 v průměru 44 % resp. 151 % v závislosti na použité variantě softwarové izolace a nárůst binárního kódu 81 % resp. 62 %, což je v obou případech poměrně akceptovatelná režie.

9.2 Přínosy práce

Režim běhu bez MMU, v současné době provozován pouze na platformě IA-32, umožní operačnímu systému HelenOS rozšířit v budoucnu portfolio podporovaných procesorů zejména o tzv. vestavěné procesory, jejichž typickým rysem je právě absence MMU jednotky a použití ve vestavěných zařízeních nejrůznějších typů.

Dalším přínosem je pravděpodobně skutečnost, že se jedná o zcela první nasazení techniky binární instrumentace pro dosažení softwarové izolace na obecném mikrojádrovém operačním systému.

Dosavadní výzkum se zaměřoval zejména na softwarovou izolaci v rámci monolitických operačních systémů, kde byl požadavek na vzájemnou izolaci modulů jádra či pluginů aplikací softwarovými prostředky, nebo na izolaci procesů v rámci vestavných zařízení, kde byl typicky nasazen některý ze specializovaných operačních systémů (např. systémy reálného času). Žádný ze studovaných odborných článků však nereferoval o nasazení binární instrumentace na všechny procesy v rámci mikrojádrového operačního systému.

Za přínos lze považovat také detailní popis implementace techniky XFI a volnou šiřitelnost jejich zdrojových kódů. Přestože technika XFI nebyla navržena v rámci této diplomové práce, ale vytvořena před několika lety v rámci výzkumu v Microsoft Research, její autoři v publikovaných článcích neuvádí detailní popis implementace a pravděpodobně z důvodu autorských práv nemohou poskytnout její zdrojové soubory[1].

Jako neplánovaný vedlejší produkt této diplomové práce vznikl framework STATIF (viz příloha D) pro statickou binární instrumentaci nezávislý na konkrétní instrukční sadě, který na rozdíl od jiných dostupných nástrojů pro spustitelný formát ELF umožňuje provádět velmi jemnou a cílenou instrumentaci – tj. např. vložení libovolné sekvence instrukcí na přesně zadanou pozici v binárním kódu aplikace. V současnosti zahrnuje pouze podporu pro procesory z rodiny IA-32, avšak jeho rozšíření o další instrukční sady znamená hlavně přidání kodéru a dekodéru pro tyto sady. Existence a provozuschopnost tohoto frameworku na binárních kódech aplikací systému HelenOS může v budoucnu také posloužit např. pro vylepšení ladícího systému, kde lze uplatit binární instrumentaci pro profiling aplikací.

9.3 Navazující práce

Řadu možných vylepšení a témat pro budoucí rozšíření práce uvedla kapitola 7. Z těch nejpodstatnějších uveďme např. optimalizaci alokátoru rámců fyzické paměti, který je v současné verzi stále založen na buddy alokátoru s velkou interní fragmentací.

Prostor pro optimalizace skýtá i současná implementace běhových kontrol XFI, kde např. v původním návrhu této techniky autoři zmiňují použití tzv. verifikačních stavů v rámci celého binárního kódu, zatímco implementace v této diplomové práci používá pouze její podmnožinu v podobě statické analýzy práce s registrem zásobníku a ukazatelem na aktivační záznam procedur. Z tohoto důvodu je stávající implementace sice jednodušší, ale generuje méně efektivní kód.

Přirozeným pokračováním této práce by také bylo vytvoření XFI implementace na jiném procesoru bez MMU jednotky a portace HelenOS na tento systém v režimu bez MMU. Tím by se skutečně ověřila přenositelnost současné implementace.

Díky frameworku pro binární instrumentaci si lze také představit další práci na poli výzkumu v oblasti softwarové izolace v systému HelenOS.

[1] Autor této diplomové práce byl v kontaktu s jedním z autorů článku [9], který sdělil, že zdrojové kódy nemohou být poskytnuty.

Literatura

[1] Abadi M., Budiu M., Erlingsson U., Ligatti J.: *Control-flow integrity: Principles, implementations, and applications, ACM Computer and Communications Security Confefence*, 2005.

[2] AMD64 procesory: `http://www.amd.com/us-en/Processors/`.

[3] ARM Limited: `http://www.arm.com/`.

[4] Austin T. M., Breach S. E., Sohi G. S.: *Efficient detection of all pointer and array access errors, Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, strany 290 – 301, 1994.

[5] Blackfin procesory: `http://www.analog.com/en/embedded-processing-dsp/blackfin/content/index.html`.

[6] Budiu M., Erlingsson U., Abadi M.: *Architectural Support for Software Based Protection, Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, strany 42 – 51, 2006.

[7] Condit J., Harren M., McPeak S., Necula G. C. , Weimer W.: *CCured in the real world, Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, strany 232 – 244, 2003.

[8] Cyclone programovací jazyk: `http://cyclone.thelanguage.org/`.

[9] Erlingsson U., Abadi M., Vrable M., Budiu M., Necula G. C.: *XFI: Software Guards for System Address Spaces, OSDI*, 2006.

[10] GNU Compiler Collection, `http://gcc.gnu.org/`.

[11] HelenOS 0.2.0 Design Documentation, `http://www.helenos.eu/doc/design.pdf`.

[12] HelenOS stránky, `http://www.helenos.eu`.

[13] Hunt G. C., Larus J.R.: *Singularity: Rethinking the Software Stack*, ACM SIGOPS Operating Systems Review, 2007.

[14] Choudhuri S., Givaris T.: *Software Virtual Memory Management for MMU-less Embedded Systems*, technická zpráva Kalifornské univerzity, 2005.

[15] Independent JPEG Group, `http://www.ijg.org/`.

[16] Intel IA-32 procesory: `http://www.intel.com/`.

[17] Kára A.: *Optimization of binary machine code on Intel IA-32*, diplomová práce, Univerzita Karlova v Praze, 2006.

[18] Kowshik S., Dhurjati D., Adve V.: *Ensuring code safety without runtime checks for real-time control systems, Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, strany 288 – 297, 2002.

[19] Kumar R., Kohler E., Srivastava M.: *Harbor: Softwarebased Memory Protection For Sensor Nodes, Proceedings of the 6th international conference on Information processing in sensor networks*, strany 340 – 349, 2007.

[20] Kumar R., Singhania A., Castner A., Kohler E., Srivastava M.: *A system for coarse grained memory protection in tiny embedded processors, Proceedings of the 44th annual Design Automation Conference* , strany 218 – 223, 2007.

[21] Leontie E., Bloom G., Narahari B., Simha R., Zambreno J.: *Hardware-enforced Fine-grained Isolation of Untrusted Code, Proceedings of the first ACM workshop on Secure execution of untrusted code*, strany 11 – 18, 2009.

[22] McCullough, D.: *uClinux for Linux programmers*, Linux Journal, 2004.

[23] MIPS32 architektura: `http://www.mips.com/products/architectures/mips32/`.

[24] Park J., Lee J., Kim S., Hong S.: *Quasistatic shared libraries and XIP for memory footprint reduction in MMU-less embedded systems*, Transactions on Embedded Computing Systems, Volume 8, 2008.

[25] QEMU emulátor a virtualizér, `http://wiki.qemu.org/Main_Page`.

[26] Rivest R.:*The MD5 Message-Digest Algorithm*, RFC 1321, IETF, 1992.

[27] Small C., Seltzer M. I.: *MiSFIT: Constructing safe extensible systems, IEEE Concurrency: Parallel, Distributed and Mobile Computing*, 6(3), 1998.

[28] Small C.: *Building An Extensible Operating System, dizertační práce, Harvardská univerzita*, 1998.

[29] SPARC architektura: `http://www.sparc.com/standards/SPARCV9.pdf`.

[30] Srivastava A., Edwards A., Vo H.: *Vulcan: Binary transformation in a distributed environment*, technická zpráva MSR-TR-2001-50, Microsoft Research, 2001.

[31] Standard Performance Evaluation Corporation: *SPEC CPU2006 benchmark suite*, 2006, `http://www.spec.org/osg/cpu2006/`.

[32] Sun Microsystems, Inc.: *Adaptive differential pulse-code modulation*, `http://www.data-compression.com/G711_G721_G723.tar.gz`.

[33] Svoboda J.: *Dynamic Linker and Debugging/Tracing Interface for HelenOS*, diplomová práce, Univerzita Karlova v Praze, 2008.

[34] System V Application Binary Interface, Edition 4.1, `http://www.sco.com/developers/devspecs/gabi41.pdf`.

[35] Tanenbaum A. S.: *Moderní operační systémy*, Prentice Hall, 2008.

[36] uClinux, `http://www.uclinux.org/`.

[37] VMware Player, VMware Inc., `http://www.vmware.com`.

[38] Wahbe R., Lucco S., Anderson T. E., Graham S. L.: *Efficient software-based fault isolation, Proceedings of the Fourteenth Symposium on Operating Systems Principles*, strany 203 – 216, 1993.

[39] Weerasinghe N., Coulson G.: *Lightweight module isolation for sensor nodes, Proceedings of the First Workshop on Virtualization in Mobile Computing*, strany 24 – 29, 2008.

[40] Witchel E., Cates J., Asanovic K.: *Mondrian memory protection, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, strany 304 – 316, 2002.

[41] Witchel E., Rhee J., Asanovic K.: *Mondrix: Memory Isolation for Linux using Mondriaan Memory Protection*, Proceedings of the twentieth ACM symposium on Operating systems principles, 2005.

[42] Zhivich M., Leek T., Lippmann R.: *Dynamic Buffer Overflow Detection, Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

Příloha A

Obsah přiloženého CD

Součástí práce je CD disk obsahující samotný text práce v elektronické podobě, zdrojové soubory, výsledky srovnávacích testů a spustitelné ISO obrazy systému HelenOS vytvořené překladem různých konfigurací zdrojových souborů.

Samotné CD lze rovněž použít pro nabootování operačního systému HelenOS v režimu bez MMU jednotky s nasazením techniky XFI v každé aplikaci (paměťový strážce byl použit v klasické variantě bez paměťové bitmapy).

V téměř každém adresáři na CD disku se nachází soubor `readme.txt`, který blíže popisuje obsah adresáře, takže ho lze vždy použít pro získání přesných informací.

Přehled obsahu CD disku je následující:

Adresář	Obsah
`Benchmarks/`	Výsledky srovnávacích testů v podobě CSV souborů a ISO obrazy systému HelenOS použité pro jednotlivá měření srovnávacích testů
`Documentation/`	Text diplomové práce a technická dokumentace vygenerovaná ze zdrojových souborů
`ISO-Images/`	Spustitelné ISO obrazy různých verzí systému HelenOS
`Sources/`	Zdrojové soubory původní i změněné verze systému HelenOS 0.4.2, zdrojové soubory nástroje *ia32binrewriter*

Příloha B

Sestavení a spuštění HelenOS

B.1 Sestavení ze zdrojových souborů

Zdrojové soubory upravené verze operačního systému HelenOS jsou umístěny v archivu `Sources\helenos-0.4.2-nommu-xfi.tgz`. Po jejich rozbalení např. do adresáře `~/helenos` je autory HelenOS doporučováno nainstalování toolchainu s cross-kompilátorem `gcc` pro architekturu IA-32. To je zajištěno spuštěním skriptu:

```
$ cd ~/helenos/tools
$ ./toolchain.sh ia32
```

Přestože použití nativního kompilátoru `gcc` pro překlad není doporučováno, při vývoji implementace této diplomové práce byl bez problémů běžně používán nativní kompilátor `gcc` verze 4.4.4.

Před samotným překladem je nutné provést konfiguraci týkající se nastavení cílové procesorové architektury a podporovaných vlastností výsledného obrazu systému. Konfiguraci lze provést spuštěním příkazu

```
$ cd ~/helenos
$ make config
```

který zobrazí grafické konfigurační menu, kde je možné nastavovat požadované vlastnosti. Pro snažší překlad verze systému, která je výsledkem této diplomové práce, je možné použít již připravené konfigurace – na přiloženém CD v adresáři `Sources\helenos-0.4.2-nommu-xfi-config\` jsou pro každou variantu reprezentovanou samostatným adresářem uvedeny potřebné konfigurační soubory. Tyto soubory stačí nakopírovat do kořenového adresáře `~/helenos` se zdrojovými soubory.

Varianty konfigurací jsou následující:

- **classic** – klasická varianta, režim bez MMU ani XFI podpora nejsou součástí překladu, funkcionalita tohoto obrazu jádra je tedy stejná jako při použití oficiální verze 0.4.2.
- **nommu** – varianta s podporou režimu bez MMU jednotky, kdy je vypnuto stránkování, ale podpora XFI není součástí překladu.
- **nommu-xfi** – podpora režimu bez MMU jednotky včetně softwarové izolace XFI technikou (paměťový strážce nepoužívá paměťovou bitmapu, nýbrž pouze seznamy regionů).

Nakonec se překlad celého systému provede zavoláním příkazu

```
$ make
```

Skončí-li překlad bez chyby, výsledkem je bootovatelný ISO obraz `image.iso` v kořenovém adresáři, odkud byl spouštěn příkaz `make`.

B.2 Spuštění

Nabootování operačního systému HelenOS lze uskutečnit vypálením výsledného ISO obrazu na CD disk a poté nabootování z tohoto disku přímo na reálném hardware.

Při vývoji však byly využívány hlavně emulátory resp. virtualizéry procesoru IA-32 a to `QEMU` [25] a `VMware Player` [37], ve kterých bylo rovněž možné provést nabootování z ISO obrazu systému.

Spuštění v emulátoru `QEMU` vypadá následovně:

```
$ qemu -m 64 -cdrom image.iso -boot d
```

Uvedené parametry zajistí pro emulovaný stroj fyzickou paměť velikosti 64 MB a nabootování ISO obrazu ze souboru `image.iso`.

Spuštění ve virtualizéru `VMware Player` lze provést vložením ISO obrazu do virtuální CD-ROM mechaniky a restartem virtuálního stroje.

Příloha C

Nástroj ia32binrewriter

C.1 Sestavení ze zdrojových souborů

Sestavení a spuštění nástroje *ia32binrewriter* vyžaduje operační systém unixového typu s instalovanými nástroji autotools a překladačem gcc a rovněž dynamicky linkovanou knihovnou libelf. Rozbalení zdrojových souborů (z přiloženého CD disku v adresáři `Sources`) a překlad lze provést spuštěním těchto příkazů:

```
tar -xzf ia32binrewriter.tgz ia32binrewriter
cd ia32binrewriter
./configure && make
```

Spustitelný soubor bude vytvořen v adresáři `src`.

Při vývoji byl překlad úspěšně prováděn na operačních systémech Gentoo Linux 2.3 a Fedora 12 s překladačem gcc 4.4.4, nástroji binutils 2.20.1 a knihovnou libelf v rámci sady nástrojů elfutils verze 0.146.

Přeložená podoba *ia32binrewriter* je rovněž součástí zdrojových souborů HelenOS, takže při překladu celého operačního systému ze zdrojových souborů není nutné dělat překlad nástroje ručně.

C.2 Příkazová řádka

Parametry příkazové řádky s krátkým popisem jsou uvedeny v tabulce na následující straně. Kromě těchto parametrů je nutné specifikovat název souboru, který bude předmětem binární instrumentace technikou XFI. Výsledek je uložen do stejného souboru.

Užitečným parametrem, zejména při diagnostice různých problémů, je volba `-i` způsobující vygenerování disassembler výstupu obsahujícího základní bloky s instrukcemi jak původního, tak i generované kódu. Generované instrukce jsou navíc označeny v komentáři písmenem G. Parametry pro zapínání (`-e`) či vypínání (`-d`) jednotlivých strážců je pak možné použít v případech, kdy je podezření na chybu v běhových kontrolách některých strážců. Rovněž byly využity při srovnávacích testech, kdy se měřila režie jednotlivých strážců a bylo nutné zapínat je postupně.

C.3 Příklad použití

Následující příklad použití se aplikuje také při sestavování všech aplikací HelenOS s podporou XFI:

```
ia32binrewriter -q -s context_save -s context_restore main
```

Obecné parametry	
`-s <name>, --skip <name>`	Vynechá zadanou ELF sekci či funkci z instrumentačího procesu
`-p, --pic`	Generování pozičně nezávislého kódu (implicitní volba)
`-np, --no-pic`	Generování absolutní pozicovaného kódu
`-i`	Generování disassembler výstupu s instrumentovaným kódem
`-q, --quiet`	Tichý režim, pouze chyby a varování jsou vypisovány na konzoli
Parametry povolení/zakázání strážců	
`-eM, --enable-mem-guard`	Zapnutí paměťového strážce
`-eS, --enable-stack-guard`	Zapnutí strážce zásobníku
`-eC, --enable-control-flow-guard`	Zapnutí strážce integrity toku řízení
`-dM, --disable-mem-guard`	Vypnutí paměťového strážce
`-dS, --disable-stack-guard`	Vypnutí strážce zásobníku
`-dC, --disable-control-flow-guard`	Vypnutí strážce integrity toku řízení
Parametry strážce zásobníku	
`-Sk <const>, --stack-K <const>`	Nastavení konstanty K na hodnotu `<const>`, implicitní hodnota je 64
Parametry paměťového strážce	
`-M[r][w][x]`	Určuje paměťové operace chráněné paměťovým strážcem: `r` – čtení, `w` – zápis, `x` – spuštění, implicitní je kombinace `wx`
`-Msafe`	Vynucuje uchovávání originálních hodnot registrů použitých v běhových kontrolách
`-Mbitmap <size>`	Ověřování paměťových přístupů se bude dít prostřednictvím paměťové bitmapy, `<size>` udává celkovou velikost paměti pokrytou bitmapou v jednotkách MB

Tabulka C.1: Parametry příkazové řádky nástroje ia32binrewriter.

V binárním kódu aplikace `main` dochází ke vložení všech běhových kontrol na všechna místa kódu kromě procedur *context_save()* a *context_restore()*, což bylo zdůvodněno v sekci 5.5.1.

Zavolání nástroje v následující podobě

```
ia32binrewriter -q -s -Mbitmap 1024 context_save -s context_restore main
```

má stejný výsledek jako v prvním příkladě až na běhové kontroly paměťového strážce, které v tomto případě pro svou činnost budou využívat paměťovou bitmapu pokrývající fyzickou paměť velikosti 1 GB.

Příloha D

STATIF framework

STATIF (*STATic binary Instrumentation Framework*) je framework pro binární instrumentaci aplikací ve formátu ELF. Technicky se jedná o knihovnu napsanou v jazyce C a C++, která umožňuje načíst do paměti binární kód daného ELF souboru, převést tento kód do interní reprezentace (instrukce, základní bloky, procedury), provádět modifikace v této reprezentaci (vkládat instrukce, mazat základní bloky atd.) a výsledek nakonec zpět uložit ve formě binárního kódu do ELF souboru.

STATIF framework byl vyvinut v rámci této diplomové práce, avšak jeho velká část – jmenovitě kodér a dekodér instrukcí procesorů IA-32 – byla převzata z jiné diplomové práce [17], která se zabývala binární optimalizací. Hlavní motivací pro vznik této knihovny byl fakt, že podobné a propracovanější volně dostupné nástroje buď neumí pracovat s formátem ELF nebo nedokáží provádět jemnou binární instrumentaci na úrovni jednotlivých instrukcí.

Technická dokumentace knihovny STATIF framework je uvedena na přiloženém CD v HTML formátu v souboru `Documentation/STATIF/index.html`.